A Reflector for the PRISM Probabilistic Model Checker

Stephen Gilmore
Laboratory for Foundations of Computer Science

25th November 2002

The DEGAS project connects tools for the UML software modelling language to model-checkers, static analysers and solvers. *Reflectors* are used in the DEGAS project to convert the output from a verification or analysis tool back into a format which can be used by a UML modelling tool.

This paper documents a reflector for PRISM [KNP02], a probabilistic model checker for reactive modules and the PEPA stochastic process algebra.

This reflector assumes that the PEPA Extractor and the PEPA compiler have already been run. The former has extracted a `.pepa` file from an `.xmi` file. The latter has extracted a `.sm` file from the `.pepa` file and has written a log file (`.log`) mapping PEPA local state identifiers onto the numeric constants used in the reactive modules notation. The output from the PRISM tool onto standard out has been captured and saved in a `.pres` (PRISM results) file. The PRISM Extractor reads the `.log` file and the `.pres` file and writes an `.xml` file which can be read by the PEPA Reflector.

This reflector is written in Standard ML and can be compiled with a pure Standard ML compiler such as Moscow ML or Standard ML of New Jersey or can be compiled with the MLj compiler, which writes its results as a zipped archive of Java class files suitable for running on the Java Virtual Machine.

The MLj wrapper for the Reflector is listed in Appendix A. The PRISM Reflector is listed in Appendix B.

Acknowledgements: The author is supported by the DEGAS (Design Environments for Global ApplicationS) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing.

References

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, April 2002.

A MLj wrapper for the PRISM reflector

```
(*
   File: reflector.sml

   This is the root file for the MLj compilation process and
   refers the compiler to the PRISM Standard ML structure.

*)

structure reflector = struct

   fun error() =
       (TextIO.output(TextIO.stdErr,
            "usage: java -cp reflector.zip reflector filename\n");
        TextIO.flushOut(TextIO.stdErr))

_public _classtype T
{

  _public _static _final _method "main" (env : Java.String option
                                                Java.array option) =
      case env of
        NONE =>
           error ()
      | SOME env' =>
        let
          val array = Java.toArray env'
        in
          if Array.length array = 0
          then error ()
          else
            case Array.sub(array, 0) of
              NONE => error ()
            | SOME jstr =>
              PRISM.main (Java.toString jstr)
        end
}


end;
```

B The PRISM reflector

```
(*
   File: PRISM.sml

   This is the PRISM reflector.  It accepts as input a PRISM output
   file (the transcript of messages printed to the standard output
   stream) and converts this into the XML results format written
   by the PEPA Workbench.  This results file is then reflected using
   the PEPA Reflector to produce a modified UML model in the XMI file
   format.

*)

structure PRISM =
struct

  val jobname = ref ""

  fun member x [] = false
    | member x (h as (_, x') :: t) = x = x' orelse member x t

  exception Getcode
  fun getcode x [] = raise Getcode
    | getcode x ((h as (n, x')) :: t) = if x = x' then h else getcode x t

  fun gensym seen _ [] = rev seen
    | gensym seen n (h::t) =
          if member h seen
          then gensym ((getcode h seen) :: seen) n t
          else gensym ((n, h) :: seen) (n + 1) t

  exception LogToVector and LookupDef
  fun logToVector log =
    case rev log of
      [] => raise LogToVector
    | ("System" :: sys)::t =>
            let val sysCoded = gensym [] 0 sys
             in (Vector.fromList (map #1 sysCoded), logToVector' sysCoded t)
            end
    | _::t => raise LogToVector
  and logToVector' sysCoded defs = Vector.fromList (makeList 0 sysCoded defs)
  and makeList n sysCoded defs =
    case lookup n sysCoded of
        NONE => []
    | SOME a => (lookupDef a defs) :: makeList (n + 1) sysCoded defs
  and lookup n [] = NONE
    | lookup n (h as (n', x) :: t) = if n = n' then SOME x else lookup n t
  and lookupDef a ((h as (a' :: localStates)) :: t) =
          if a = a' then Vector.fromList localStates else lookupDef a t
    | lookupDef a _ = raise LookupDef

  fun sep #" " = true
    | sep #"=" = true
    | sep #"(" = true
    | sep #")" = true
    | sep #"," = true
    | sep #"{" = true
    | sep #"}" = true
    | sep #"|" = true
    | sep #"\n" = true
    | sep _ = false

  exception fatalInputOutputError
  fun error s = (TextIO.output (TextIO.stdErr, ">>> Error: " ^ s ^ "\n");
                 TextIO.flushOut TextIO.stdErr;
                 raise fatalInputOutputError)

  fun tryOpenIn filename =
     let val is = TextIO.openIn filename
     in is
     end handle _ => error ("Could not open file named: " ^ filename)

  fun first is =
     String.tokens sep (TextIO.inputLine is)

  fun parseExtractorLog is =
     if TextIO.endOfStream is
     then []
     else first is :: parseExtractorLog is

  fun readExtractorLog () =
     let val is = tryOpenIn (!jobname ^ ".log")
         val result = parseExtractorLog is
     in
         TextIO.closeIn is;
         logToVector result
     end

  datatype localStates = None
    | Archive of localStates ref * (int * real) ref * localStates ref

  fun addStateProbability
        (datum as  (stateNumber, probability))
        (tree  as  ref None) =
           tree := Archive (ref None, ref datum, ref None)
    | addStateProbability
        (datum as (stateNumber, probability))
        (tree  as ref (Archive (left, value as ref (s', p'), right))) =
           if stateNumber = s'
           then value := (s', p' + probability)
           else if stateNumber < s'
                then addStateProbability datum left
                else addStateProbability datum right

  exception GetStateProbability
  fun getStateProbability
        stateNumber
        (tree  as  ref None) =
           raise GetStateProbability
    | getStateProbability
        stateNumber
        (tree  as ref (Archive (left, value as ref (s', p'), right))) =
           if stateNumber = s'
           then p'
           else if stateNumber < s'
```

```
              then getStateProbability stateNumber left
              else getStateProbability stateNumber right

datatype resultsTree = Empty
  | Node of resultsTree ref * (int * localStates ref) ref * resultsTree ref

fun addResult
      (result as  (componentNum, datum as (stateNumber, probability)))
      (tree  as  ref Empty) =
         let val newResult = ref None
          in addStateProbability datum newResult;
             tree := Node (ref Empty, ref (componentNum, newResult), ref Empty)
         end
  | addResult
      (result as  (componentNumber, datum as (stateNumber, probability)))
      (tree  as ref (Node (left, value as ref (c', r'), right))) =
         if componentNumber = c'
         then addStateProbability datum r'
         else if componentNumber < c'
              then addResult result left
              else addResult result right

exception GetResult
fun getResult
        (query as  (componentNumber, stateNumber))
        (tree  as  ref Empty) =
           raise GetResult
  | getResult
        (query as  (componentNumber, stateNumber))
        (tree  as ref (Node (left, value as ref (c', r'), right))) =
           if componentNumber = c'
           then getStateProbability stateNumber r'
           else if componentNumber < c'
                then getResult query left
                else getResult query right

val results = ref Empty

fun printXMLresults os =
  let
     val (offsets, defns) = readExtractorLog()
     fun printProbability component stateNumber =
         let
            val prob = getResult (component, stateNumber) results
            fun unML #"~" = "-" | unML c = str c
            val formattedProb = String.translate unML (Real.toString prob)
         in
            TextIO.output (os, "      <Probability>");
            TextIO.output (os, formattedProb);
            TextIO.output (os, "</Probability>\n")
         end

     fun printStates component stateNumber localStates =
         if stateNumber = Vector.length localStates
         then () (* vectors number from zero *)
         else let
              in
                 TextIO.output (os, "    <State Name=\"" ^
                     Vector.sub(localStates, stateNumber) ^ "\">\n");
                 printProbability component stateNumber;
                 TextIO.output (os, "     </State>\n");
                 printStates component (stateNumber + 1) localStates
              end

     fun printComponents n =
         if n = Vector.length offsets then () (* vectors number from zero *)
         else let
                 val componentNumber = Vector.sub(offsets, n)
                 val localStates = Vector.sub(defns, componentNumber)
                 val ns = Int.toString n
              in
                 TextIO.output (os, "  <Component Name=\"" ^ ns ^ "\">\n");
                 printStates n 0 localStates;
                 TextIO.output (os, "  </Component>\n");
                 printComponents (n + 1)
              end
  in
     TextIO.output (os, "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n");
     TextIO.output (os, "<PEPA_Workbench_Results>\n");
     printComponents 0;
     TextIO.output (os, "</PEPA_Workbench_Results>\n")
  end

fun printInt  os NONE      = TextIO.output (os, ">>> ERROR: parsing integer\n")
  | printInt  os (SOME i)  = TextIO.output (os, Int.toString i ^ "\n")
fun printReal os NONE      = TextIO.output (os, ">>> ERROR: parsing real\n")
  | printReal os (SOME r)  = TextIO.output (os, Real.toString r ^ "\n")

exception Record
fun record componentNumber [] p = ()
  | record componentNumber ((SOME h)::t) p =
    (addResult (componentNumber, (h, p)) results;
     record (componentNumber+1) t p)
  | record _ _ _ = raise Record

exception Accumulate
fun accumulate (stateVector, SOME probability) = record 0 stateVector probability
  | accumulate (_, NONE) = raise Accumulate

fun separator #":" = true
  | separator #"(" = true
  | separator #"," = true
  | separator #")" = true
  | separator #"=" = true
  | separator #"\n" = true
  | separator _ = false

exception Format

fun format acc [last] = (rev acc, Real.fromString last)
  | format acc (h::t) = format (Int.fromString h :: acc) t
  | format _ [] = raise Format

fun parse line = format [] (tl (String.tokens separator line))

fun startsWithNumeral [] = false
```

```
    | startsWithNumeral (h::t) = Char.isDigit h

  fun state line = startsWithNumeral (explode line)

  fun process is os =
    let
    in  while not (TextIO.endOfStream is) do
           let val line = TextIO.inputLine is
           in if state line
              then accumulate (parse line)
              else ()
           end;
         printXMLresults os
    end

  fun main basename =
    let val inputStream = tryOpenIn (basename ^ ".pres")
        val outputStream = TextIO.openOut (basename ^ ".xml")
    in  jobname := basename;
        process inputStream outputStream;
        TextIO.closeIn inputStream;
        TextIO.closeOut outputStream
    end

end;
```