An experimental tool for checking probabilistic program refinement

Carlos J. Gonzalía

cjg@cs.uns.edu.ar
Departamento de Ciencias e Ingeniería en Computación,
Universidad Nacional del Sur,
Avenida Alem 1253, 8000 Bahía Blanca,
Provincia de Buenos Aires, Argentina

Abstract. We discuss the features and current status of a software tool developed for checking refinement properties of a particular (though still quite general) class of probabilistic nondeterministic programs. The tool has been used successfully for several interesting examples, and its development is currently being expanded to encompass even more classes of probabilistic programs. The novelty of the tool lies in its expressing the refinement checking as a linear satisfiability problem, and also in producing evidence of lack of refinement by a problem of the same kind.
Keywords: *formal methods, probabilistic programs, computer verification, refinement checking, trace generation, linear satisfiability checking.*

1 Background

We are interested in the use of some versions of pGCL (probabilistic Guarded Command Language) for the purpose of machine-assisted formal methods in establishing properties of interest about programs. pGCL is a generalization of Dijkstra's *guarded command language* ([6]) by adding an operator for probabilistic choice (and retaining the demonic nondeterministic choice operator, that is to say, a choice which we cannot affect and which is not made with respect to any probability). This results in a quite non-trivial addition, since the behaviour of programs that contain both nondeterministic (as in the original language) and probabilistic choice can exhibit complex behaviour, difficult to figure out intuitively. Our long term goal is to be able to solve some interesting problems about web services, that can be expressed by probabilistic programs and their properties of interest, using computer-assisted formal methods ([4]).

The semantic model for pGCL we use is the one in which a program is seen operationally as a *function or transition mapping initial states to sets of probability distributions over final states.* More exactly, the formal semantics uses discrete distributions, and the final sets of those distributions that are the meaning of programs are restricted to the convex, up-closed, and Cauchy closed ones. This is not a problem for the tool design and use, since such complex sets of distributions are always implied by a finite set of discrete distributions over final states. If the initial states are finite, so are the final ones too, and that is the context in which we work for the time being. Technical details and examples of pGCL can be found in previously published results by McIver and Morgan (see [11] for a comprehensive reference to all aspects of pGCL).

With this semantics, if we call S to our set of states, we can define what it means for a program p to be refined (usually denoted by the symbol $\sqsubseteq$) by another program q in pGCL:

$$p \sqsubseteq q \text{ iff } \forall s \in S : p.s \supseteq q.s$$

where $p.s$ denotes the set of discrete probability distributions (satisfying the convexity and closure properties already mentioned) over final states that is the meaning of program p.

We have called the tool **pGCLd** , as it fundamentally leans on this operational view of the semantics of probabilistic programs to generate both final distributions and traces of the program behaviour. The tool outputs files that can be processed by the linear satisfiability solver Yices ([17]) to test refinement between such programs. A linear satisfaction problem is similar to a linear program, formed by several linear arithmetic constraints, but instead of solving for a maximum or minimum of some function subjected to those constraints, we only need to know if there exists values for all the variables such that all the constraints are satisfied.

2 The input language: a subset of pGCL

Our working language is a subset of pGCL: we don't yet deal with iteration as present in the full language. The tool's syntax is case-sensitive, and all variables are implicitly declared with (implicit) type integer. It is possible to have arrays of such variables too. A program consists of a sequential composition of program fragments, where each fragment can be either an assignment, a conditional, a demonic choice, or a probabilistic choice. Conditionals must have both branches, demonic choice is always binary (not a limitation due to its associativity), and probabilistic choice can have as many branches as desired.

A (simplified) BNF description of the input syntax follows:

variable ::= letter (letter | digit)*

integer ::= [$\sim$] digit$^+$

real ::= digit$^+$. digit$^+$

numop ::= + | - | * | / | %

numexp ::= variable | integer | (numexp) |
    numexp numop numexp | $\sim$ numexp

relop ::= < | <= | > | >= | = | <>

boolexp ::= TT | FF | (boolexp) | numexp relop numexp |
  boolexp & boolexp | boolexp `|` boolexp | ! boolexp

prog ::= variable `:=` numexp | (prog) | prog ; prog |
  if boolexp then prog else prog endif | prog `^` prog |
  @{ real : prog [, real : prog]* }

`^` denotes demonic choice, @ followed by a list of branches between braces denotes probabilistic choice, conditionals have their condition between square brackets (the "then" part to the left of the condition, the "else" part to the right), composition is denoted by semicolon. In Boolean expressions, & denotes and, `|` or, ! not. In integer expressions, % is the remainder operation. Notice that negative integers and sign inversion are denoted by a $\sim$, following ML syntax. Note again that we don't deal (for the time being) with loops in our tool, but that this still allows us to investigate many interesting small programs and their properties, due to the already mentioned complex behaviour of such programs when probabilistic and nondeterministic choice interact freely.

There is a default priority for the operators and constructions of pGCL programs, though the reader will probably feel more comfortable parenthesizing fully all the program fragments that may cause ambiguity. For Boolean expressions, or is lowest, and comes next, and not is highest. Boolean operators have lower precedence than comparison relations, which are all ranked at the same precedence level. Arithmetic expressions have higher precedence than comparison relations and follow the usual rules of precedence for such expressions. For program constructions, conditionals are the strongest, then demonic choice, and finally sequencing as the lowest precedence one.

An input file should contain only one pGCL program, well-formed according to the preceding grammatical rules. Syntax errors are reported to the user with a brief indication of the position where the first such error was detected during parsing.

3 Some simple examples of tool use

To illustrate how the tool works and the kind of information about refinement we can obtain through its use, we show in this section some simple examples to give the reader an idea of how things work in practice.

In all the following examples, suppose program $p1$ is the following one:

```
x:= 0 ^ x := 1
```

This program's meaning is that the value of `x` is chosen nondeterministically to be either 0 or 1.

Now suppose program $p2$ is:

```
@{0.5: x := 0, 0.5: x:=1}
```

And this program's meaning is that the value of `x` is chosen to be either 0 or 1, randomly and with the same probability of choosing one of the two possible values.

3.1 Generating final distributions

Executing **pGCLd** we can get the corresponding distribution tables for both programs, respectively for $p1$:

```
[<{x=0}, 1.0>]
[<{x=1}, 1.0>]
```

and for $p2$:

```
[<{x=0}, 0.5><{x=1}, 0.5>]
```

The results are shown in this notation: each distribution is between matching brackets, and each point of the distribution (consisting of a final state and a probability for that state to be reached as final) is between angular brackets.

This is implemented as the straightforward following of the operational semantics of pGCL mentioned before, that is, transforming a current set of distributions over states along the way as the program sentences are found. Some additional details can be found in [5]. In particular, the reader might find the rewriting rules for the used fragment of pGCL of interest, as they present some complexities any implementation of a semantic evaluator for pGCL needs to deal with. While our first implementations used the rewriting rules just cited, currently we have a smarter traversal of the input program's syntax tree in which they are implicitly used to generate the resulting probability distributions, improving the efficiency.

We show now a program that uses array variables, and how their indices can be affected by probabilistic choices too, which illustrates how complex and hard it could be for a person to figure out the behaviour of a pGCL program without the help of a tool generating the final distributions. Consider the program $p3$:

```
@{1/3: a := 1, 2/3: a:=2};
n := 2*a;
if n>2 then x[a-1] := 2 else x[a-2] := 1 endif
```

After processing it with our tool and asking for the result distributions, we obtain:

```
Distribution list for program:

There are 1 distributions.

Distribution with 2 component(s):
<{a=1, n=2, x[0]=2}, 0.333333333333>
<{a=2, n=4, x[1]=2}, 0.666666666667>
```

We should point out that undefined variables don't show up on the final state that is part of a result distribution, for instance `x[1]` in the first component of the result for $p3$ just shown.

3.2 Testing refinement between two programs

We can also use the tool to test the first program against the second one for containment. The execution of Yices on the result file will look like this, assuming we have stored the representations of $p1$ and $p2$ in the respective files `ex1.p` and `ex2.p`:

```
Containment check for ex1.p against ex2.p:

L1 is [<{x=0}, 1.0>]
L2 is [<{x=1}, 1.0>]

Testing against (0.5, 0.5): sat
(= L1 1/2)
(= L2 1/2)
```

There is a reason for the shape of L1 and L2, and it has to do with being able to understand the evidence the solver Yices can present to us when checking the refinement problem as a satisfiability one. Abstracting from the Yices syntax for the file generated by our tool, which by the way follows the conventions of LISP (this explains why the values for L1 and L2 are shown in such a way) the solver is asked to verify the satisfiability of the following problem:

$$1 \times L1 + 0 \times L2 \geq 0.5$$

$$0 \times L1 + 1 \times L2 \geq 0.5$$

subject to $L1 + L2 > 0$ and $L1 + L2 = 1$ ($L1, L2 \in [0, 1]$). As informed by the Yices solver, the problem has as an (obvious!) solution the values $L1 = 0.5$ and $L2 = 0.5$.

The reason for this translation to that particular linear satisfiability problem is explained fully in our previous work [12]. Basically the idea is that we express a program as a convex set, determined by the final distributions. Then the refinement problem $p \sqsubseteq q$ reduces (nontrivially!) to a verification that each distribution of the refinement q is contained in the convex set of the specification p, and this can be expressed as a linear satisfiability problem solvable by a tool like Yices. If all the points of p are contained in the convex set of q, then p is refined by q. In the previous example, the results obtained via Yices mean that $p1$ is refined by $p2$.

If we test the containment in the other direction, the result will be negative and look like this:

```
Containment check for ex2.p against ex1.p:
```

```
L1 is [<{x=0}, 0.5><{x=1}, 0.5>]

Testing against (1.0, 0.0): unsat
unsat core ids: 3

Testing against (0.0, 1.0): unsat
unsat core ids: 4
```

There are two separate tests, since $p1$ has two final distributions that the whole of $p2$ needs to be checked for containment.

Here the ids mentioned in the result are mentions to which constraints failed to be satisfied in each test. If you inspect the Yices file generated by our tool, the numbering of the ids refers to the position of the failing constraint, the first listed being `id 1`. In this case, then, we have that $p2$ does not refine $p1$. Again, abstracting from Yices syntax, the first test would be the satisfiability problem (without solutions):

$$0.5 \times L1 \geq 1$$

$$0.5 \times L1 \geq 0$$

subject to $L1 = 1$ (and of course to $L1 > 0$). The first constraint already fails, and is the `id 3` that the Yices solver is talking about.

3.3 Finding evidence that refutes a refinement

Ideally, once we know a refinement doesn't hold, we would like evidence for it, that is, a *certificate* for refuting the proposed refinement. To see how the preceding success and failure relate to results that give us such certificate, we show now how those results look like:

```
Separation check for ex1.p against ex2.p:

L1 is {x=0}
L2 is {x=1}

Testing against (0.5, 0.5): unsat
unsat core ids: 1 2
```

Not surprisingly, since as we have seen before there is refinement of $p1$ by $p2$, no refuting certificate is found.

On the other hand, let's see what happens when asking for a certificate that $p2$ isn't refined by $p1$, which we also know from before:

```
Separation check for ex2.p against ex1.p:

L1 is {x=0}
L2 is {x=1}
```

```
Testing against (1.0, 0.0): sat
(= L1 0)
(= L2 2)

Testing against (0.0, 1.0): sat
(= L1 2)
(= L2 0)
```

We use the term separation, as we can say that the certificate separates $p2$ from $p1$ regarding the proposed refinement in geometric terms. The technical details are somewhat involved so we refer the interested reader to our previous work [12]. Here we will just say that if there is indeed a final distribution that falls outside the convex set of the refined program, then there is an *hyperplane* separating it from the convex set of the refinement. In the particular convex sets involved in the refinement problem, the search for such a separation can be achieved by another linear satisfiability problem. The solution to this new problem describes such a hyperplane, which is the certificate that we have been talking about (it's actually the hyperplane's *normal*).

3.4 Obtaining traces of the full execution

In many situations it is useful to see the traces that lead to a final distribution in a probabilistic program. Our tool also offers that facility, which we illustrate with another example.

Consider the following program:

```
x:=1;
@{1/2: y:=0, 1/2: y:=1};
y:=0
```

We can ask our tool to provide the final distributions, as always, but also to accompany them with the corresponding traces that ended in such distributions. The traces begin by the final state and list backwards all the program statements that were executed to reach the final state, along of course with the probability that this particular execution took place:

```
Trace list for program ex4.p:

There are 1 distributions.

Distribution with 2 component(s):
<After y := 0: {x=1, y=0}
After y := 0: {x=1, y=0}
After x := 1: {x=1}
, 0.5>
<After y := 0: {x=1, y=0}
After y := 1: {x=1, y=1}
After x := 1: {x=1}
, 0.5>
```

4 Results and some remarks on implementation

Besides examples like the preceding ones, that serve to illustrate the correctness of the methods used by our tool in conjunction with the linear solver, we have managed to apply the tool to more complex cases, even in situations in which the tool isn't fully comprehensive of the pGCL constructs used. In particular, the Tank Monitoring problem [16] was explored in a useful way by the refinement checking allowed by our tool (for details and results we refer to our work in [5]). Much more remains to be done, of course, and in particular the application of the tool to situations involving properties of web services.

As the nature of the problem turns out to be mainly the semantic evaluation of a probabilistic program of a reduced class, we decided to choose a functional programming language for the implementation. The expressive style of the functional paradigm greatly helped to make the implementation of such an experimental tool a manageable effort. We used for this task the final available version of Moscow ML ([13]) (a no longer maintained or updated version of ML that is notable for its portability, small size, and lack of complexities or bugs as an implementation environment).

Since Moscow ML is hardly the choice most readers would have picked for implementing a tool of this nature, we should remark that there were contextual reasons too for the choice. It was expected to tie the tool to other efforts using the proof assistant HOL [8], in particular to some successful formalizations of pGCL ([9]). HOL is implemented in Moscow ML, hence the original requirement for its use as the implementation environment of our tool. While those interested could probably and without much effort still try to use our tool code to work along HOL, we are currently rewriting the tool in Haskell [7] to enjoy the many powerful features of this modern functional language. In particular we are eager to extend the tool towards a version of pGCL with hidden variables and refinements that should satisfy safety properties about such secrets [10]. The rough results so far are very encouraging.

Ideally, we would also like to implement clearer, better interfaces for the non-expert user. Providing a full GUI would be the natural goal, but even working inside the command line there is some room to improve the pre- and postprocessing of results and input files communicated to the Yices solver. A particular avenue to explore is using Yices as a library to be called inside a Haskell program instead of through a chain of separate programs. This will demand, however, a careful study of how the Yices libraries can be safely combined with the runtime environment of Haskell.

5 Conclusions, other work and future work

We feel confident that our tool is already useful and promising, but also aware of its limitations and implementation quirks. A change of implementation language, most likely to Haskell, will open new possibilities of integration with other tools and libraries. We also look forward towards expanding the language of probabilistic programs accepted by the tool. Of particular interest are programs with hidden information, in which properties of interest such as refinement must also verify certain security properties (see [10]).

There are other tools for model exploration and even refinement checking for probabilistic formalisms, but as far as we are aware they also lack the ability to check for refinements in general and do it (like us) in restricted settings, like for instance PRISM [14]. Our ability to produce certificates for refuting refinement is something that a general model checker finds very awkward to do at present. Another reason for choosing to try an experimental tool of our own was that using model checkers that are based on the idea of Markov Decision Processes (for another example see RAPTURE [15]) might force us to express the program in a shape that ends up obscuring much of the pGCL formal structure. As work on probabilistic programs with hidden information was anticipated, we can say that for our purposes starting with a small tool of our own was the safest plan. An interesting possibility, now that we have a clearer view of the advantages and limitations of our experimental tool, would be combining one such probabilistic model checker with our tool to analyze properties, with the intention of making use of the particular strengths of each piece of formal methods technology according to the situation.

Finally, and in the same way that our original efforts could be integrated with related formalizations in the HOL proof assistant, the move to a Haskell-based implementation would mean the possibility to try and integrate our tool with the proof assistant Agda ([1]), also based on Haskell and using a powerful constructive logic that allows for a very high level computer formalization of programs and their properties. The possibility of successfully formalizing a programming language with random choice in a constructive logical framework has been demonstrated by the work of [2] in the proof assistant Coq ([3], also based on constructive logic), which makes us quite hopeful of a similar effort for pGCL in Agda.

Acknowledgments: The author thanks the support of ANPCyT (Agencia Nacional de Promoción Científica y Tecnológica), Ministerio de Ciencia, Tecnología e Innovación Productiva, Presidencia de la Nación, Argentina. This work has been funded through the author's project with that institution under the reference PICT PRH No. 100, and which is part of the RRHH Project No. 37 held by the Universidad Nacional del Sur as agreed with ANPCyT.

References

1. *Agda 2.* Homepage. Programming Logic Group, Chalmers University of Technology, Sweden. `http://wiki.portal.chalmers.se/agda/`

2. P. Audebaud and C. Paulin-Mohring: Proofs of randomized algorithms in Coq. In: *Science of Computer Programming*, 74(8), pages 568-589, 2009.
3. *The Coq Proof Assistant.* Homepage, INRIA, France.
4. C. Gonzalía: Métodos Relacionales para la Espacificación, Verificación y Composición de Servicios Semánticos en la Web. In: *WICC 2010. XII Workshop de Investigadores en Ciencias de la Computación, 5 y 6 de mayo de 2010, Calafate, Santa Cruz, Argentina.* RedUNCI y Universidad Nacional de la Patagonia San Juan Bosco, 2010.
5. C. Gonzalía and A. McIver: Automating Refinement Checking in Probabilistic System Design. In *Formal Methods and Software Engineering: 9th International Conference on Formal Engineering Methods, ICFEM 2007, Boca Raton, FL, USA, November 14-15, 2007. Proceedings.* Lecture Notes in Computer Science, Volume 4789, Springer, 2007.
6. E.W. Dijkstra: *A Discipline of Programming.* Prentice-Hall, 1976.
7. *The Haskell programming language.* Homepage. `http://www.haskell.org/`
8. *HOL proof assistant.* Homepage at the Sourceforge repository. `https://www.cl.cam.ac.uk/research/hvg/HOL/`
9. J. Hurd, A. McIver, C. Morgan: Probabilistic guarded commands mechanized in HOL. In: *Theoretical Computer Science.* Volume 346, Issue 1, November 2005, pages 96-112.
10. A. McIver, L. Meinicke and C.Morgan: Compositional closure for Bayes Risk in probabilistic noninterference. In: *Automata, Languages and Programming. 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II.* Lecture Notes in Computer Science, Volume 6199, Springer, 2010.
11. A. McIver and C. Morgan: *Abstraction, Refinement and Proof for Probabilistic Systems.* Monographs in Computer Science, Springer, 2005.
12. A. McIver, C. Morgan and C. Gonzalía: Proofs and refutations for probabilistic systems. In: *FM 2008: Formal Methods - 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008.* Lecture Notes in Computer Science, Volume 5014, Springer, 2008.
13. *Moscow ML.* Homepage mantained by Peter Sestoft, IT University of Copenhaguen, Denmark. `http://www.itu.dk/ sestoft/mosml.html`
14. *PRISM - Probabilistic Symbolic Model Checker.* Homepage at Department of Computer Science, University of Oxford, United Kingdom. `http://www.prismmodelchecker.org/`
15. *The RAPTURE verification tool.* Homepage maintained by Bertrand Jeannet. `http://pop-art.inrialpes.fr/people/bjeannet/rapture/rapture.html`
16. Schneider, S., Hoang, T.S., Robinson, K.A., Treharne, H.: Tank monitoring: a case study in pAMN. In: *Formal Aspects of Computing*, 18(3), pages 308-328, 2006.
17. *Yices: an SMT solver.* Homepage at Computer Science Laboratory, SRI International. `http://yices.csl.sri.com/`