# On Automated Verification of Probabilistic Programs

Axel Legay[1], Andrzej S. Murawski[2], Joël Ouaknine[2], and James Worrell[2]

[1] Institut Montefiore, Université de Liège, Belgium
[2] Oxford University Computing Laboratory, UK

**Abstract.** We introduce a simple procedural probabilistic programming language which is suitable for coding a wide variety of randomised algorithms and protocols. This language is interpreted over finite datatypes and has a decidable equivalence problem. We have implemented an automated equivalence checker, which we call APEX, for this language, based on game semantics. We illustrate our approach with three non-trivial case studies: (i) Herman's self-stabilisation algorithm; (ii) an analysis of the average shape of binary search trees obtained by certain sequences of random insertions and deletions; and (iii) the problem of anonymity in the Dining Cryptographers protocol. In particular, we record an exponential speed-up in the latter over state-of-the-art competing approaches.

## 1 Introduction

Ever since Michael Rabin's seminal paper on probabilistic algorithms [23], it has been widely recognised that introducing randomisation in the design of algorithms can yield substantial benefits. Unfortunately, randomised algorithms and protocols are notoriously difficult to get right, let alone to analyse and prove correct. In this paper, we propose a simple prototype programming language which we believe is suitable for coding a wide variety of algorithms, systems, and protocols that make use of probability.

Our language incorporates several high-level programming constructs, such as procedures (with local scoping) and arrays, but is predicated on finite datatypes and enjoys some key decidability properties. From our perspective the most important of these is *probabilistic contextual equivalence*, which can be used to express a broad range of interesting specifications on various systems.

We have developed an automated equivalence checker, APEX, for our probabilistic programming language. Our approach is based on game semantics, and enables us to verify *open* programs (i.e., programs with undefined components), which is often essential for the modular analysis of complex systems. Game semantics itself has a strong compositional flavour, which we have exploited by incorporating a number of state-space reduction procedures that are invoked throughout a verification task.

We illustrate our framework with three non-trivial case studies. The first is Herman's algorithm, a randomised self-stabilising protocol. The second is a problem about the average shape of binary search trees obtained by certain sequences of random insertions and deletions. Finally, our third case study is an analysis of anonymity in the Dining Cryptographers protocol. In the latter, we record an exponential speed-up over state-of-the-art competing approaches.

**Main Contributions.**[3] Our main contributions are twofold: First, we define a simple imperative, non-recursive call-by-name procedural probabilistic language, interpreted over finite datatypes, and show that it has a decidable contextual equivalence problem. Our language is related to second-order Probabilistic Idealized Algol, as studied in [19], and our decidability proof relies in important ways on results from [6, 19]. Among the new ingredients are direct automata constructions (rather than reliance on abstract theoretical results, as was done in [19]), in particular with respect to epsilon-elimination.

Our second—and arguably most significant—contribution lies in the novel application of our framework in the treatment of the case studies. In particular, we use contextual equivalence for *open* programs in a key way in two of the three instances. As discussed in greater details below and in Section 6, our use of contextual equivalence in the Dining Cryptographers protocol results in a dramatic improvement in both verification time and memory consumption over current alternative approaches.

**Related Work.** Proposals for imperative probabilistic programming languages, along with associated semantics, go back several decades (see, e.g., [16]). As noted in [8], most of the semantic models in the literature are variants of Markov chains, where the states of the Markov chain are determined by the program counter and the values of the variables.

While such treatments are perfectly adequate for model checking closed (monolithic) programs, they are usually ill-suited to handle open programs, in which certain variables or even procedures are left undefined. Moreover, such semantic approaches are also generally of no help in establishing (probabilistic) contextual equivalence: the indistinguishability of two open programs by any (program) context. Contextual equivalence, in turn, is arguably one of the most natural and efficient ways to specify various properties such as anonymity—see Section 6 for further details and background on this point.

As we explain in Section 3, our approach, in contrast, is based on game semantics, and differs radically from the various 'probabilistic state-transformer' semantics discussed above. The main benefit we derive is an algorithm for deciding contextual equivalence.

We note that many probabilistic model checkers, such as PRISM [11] and LiQuor [4], have been reported upon in the literature—see [18] for a partial survey. Most of these tools use probabilistic and continuous-time variants of Computation Tree Logic, although Linear Temporal Logic is also occasionally supported.

[3] A full version of this paper, which will include all the formal definitions, constructions, and proofs that have been omitted here, is currently in preparation [18].

## 2 A Probabilistic Programming Language

Code fragments accepted by APEX are written in a probabilistic procedural language with call-by-name evaluation, whose full syntax is given below.

```
const ::= [0-9]+
id ::= [a-z]+
gr_type ::= 'void' | 'int%' const | 'var%' const
gr_list ::= gr_type {',' gr_list }
type ::= { gr_type '->' | '(' gr_list ') ->' } gr_type
rand_dist ::= const ':' const '/' const {',' rand_dist }
gr_params ::= gr_type id {',' gr_params }
params ::= gr_type id { '(' gr_list ')' } {',' params }
program_list ::= program {',' program_list }
binop ::= '+' | '-' | '*' | '/' | 'and' | 'or' | '<=' | '<' | '='
unop ::= 'not'
typable_val ::= const | 'coin' | 'rand[' rand_dist ']'

program ::=
  'skip' | typable_val | '( int%'const typable_val ')' |
  id | id '(' { program_list } ')' | id '[' program ']' |
  'int%'const id | 'int%'const id '[' const ']' |
  'if' program 'then' program { 'else'  program } |
  'case' '(' program ')' '[' program_list ']' |
  program ';' program | 'while' program 'do' program |
  program ':=' program | program binop program | unop program |
  gr_type id '(' {gr_params} ') {' program '}' program |
  '(' program ')' | '{' program '}'

input ::= type 'main(' { params } ') {' program '}'
```

This language has two simple mechanisms for specifying random values. First, the 'probabilistic' constant `coin` represents the fair coin: it returns value 0 or 1, each with probability $\frac{1}{2}$. More generally, one can specify arbitrary finite distributions using `rand`, e.g., `rand[1:1/3, 2:1/3, 3:1/3]` stands for the fair three-sided die. Other syntactic elements are intended to resemble C in order to make it easier to analyse pieces of code; for example, blocks are delimited by braces ({...}). The language is predicated on finite integer datatypes, which support modulo arithmetic (`+`,`-`,`*`,`/`). Local variables can be declared using statements of the form `int%`$n$ `i`, where $n$ indicates the modulus and `i` is a variable name. Similarly, arrays are defined using declarations such as `int%`$n$ `a[`$m$`]`, where $m$ represents the size of the array. `int%2` can double as the Boolean type with the associated logical operations (`and`, `or`, `not`). Procedures can be introduced with syntax such as

```
void procname(int%4 i, var%7 j) {...}
```

where `i` is a $\{0, 1, 2, 3\}$-valued parameter (modulo 4) and `j` is a reference parameter (analogous to '`int &j`' in C++) modulo 7. Functions are defined in the same way except that `int%`$n$ should be used instead of `void` for some $n \in \mathbb{N}^+$. Procedures/functions can be declared locally within other procedures/functions, but recursive calls are not allowed. Iteration is provided in the form of `while` loops.

Our framework also supports *open* code with undefined variables, procedures or functions (also known as undefined parameters). Their names together with types (e.g., `var%6 x`, or `void f(int%3,var%7)`) have to be declared as part of the input statement whose general shape is as follows:

*type* `main(`*undefined_parameters*`)` $\{$*open_code*$\}$.

## 3 Contextual Equivalence

As a result of randomisation, closed programs of type `void` terminate with some probability, whereas closed programs of type `int%`$m$ generate a sub-distribution on $\{0, \ldots, m-1\}$. In addition to closed programs, we also consider open program fragments in which some parts are not specified and are represented by undefined parameters, as discussed earlier. Open programs cannot be executed on their own and become executable only when put in a program context that makes them closed, i.e., provides instantiations for the undefined parts. Note that open programs can alternatively be viewed as higher-order procedures.

We say that two (open or closed) programs $P_1$ and $P_2$ are *equivalent* iff they behave the same way inside all program contexts, i.e., for any context $C$ such that $C[P_1]$, $C[P_2]$ are closed programs of type `void`, $C[P_1]$ and $C[P_2]$ terminate with the same probability[4]. Thus equivalent programs exhibit the same observable behaviour. The observable behaviour of closed programs is determined simply by the sub-distribution generated by termination. That of open programs can be said to correspond to the ways the program can use its undefined components. This intuition is made precise by game semantics [1, 12, 6, 19], which we briefly examine below.

Intuitively, probabilistic contextual equivalence is a linear-time (as opposed to branching-time) and statistical (as opposed to possibilistic) notion of program equivalence. We remark that it is an especially powerful instrument in the case of open programs, which we make full use of in the case studies presented in Sections 5 and 6.

The main theoretical result underpinning the work we present here is the following.

**Theorem 1.** *Probabilistic contextual equivalence is decidable for the programming language given in Section 2.*

[4] The probability of termination is formally computed using an operational semantics; we refer the reader to [18] for the precise details.

The proof of Theorem 1 is based on game semantics and relies heavily on the results of [6, 19]. Full details will appear in [18].

Game semantics is a modelling theory for a wide range of programming paradigms. It associates to any given (probabilistic) open program a (probabilistic) *strategy*, which in turn gives rise to a set of (probabilistic) *complete plays*. *Full abstraction* is then the assertion that two open programs are contextually equivalent iff they exhibit precisely the same set of complete plays. Theorem 1 is established via a full abstraction result, in which moreover the relevant sets of complete plays can be represented using probabilistic automata [22]. Probabilistic program equivalence therefore reduces to language equivalence for probabilistic automata, which can be decided in polynomial time [25].

We note that the probabilistic automata arising from game semantics are radically different from the Markov chains that arise in the various probabilistic state-transformer semantics discussed in Section 1. Whereas the latter essentially correspond to an operational unwinding of a program, the game-semantical probabilistic automata capture the ways in which a program can interact with its environment, i.e., the broader context in which the program lies. For a more detailed account of game semantics as used in this paper, we refer the reader to [18].

Our tool APEX generates the probabilistic automata representing open programs in a compositional manner, by executing bespoke automata operations for each of the syntactic constructs of our language. The state spaces of the resultant intermediate automata are reduced using a variety of algorithmic techniques, including reachability analysis, decomposition into strongly connected components, and quotienting by probabilistic bisimulation [17].

We remark that closed programs always give rise to single-state automata, whereas open programs yield non-trivial automata. Of course, in both cases the intermediate automata produced can be arbitrarily large[5] and complex, hence the need for efficient implementations of the constructions and the use of state-space reduction techniques.

As an example, consider the following open program $P_1$:

```
void main(var%2 x) { x:=rand[0:1/3, 1:2/3]; x:=coin }
```

$P_1$ has a single free identifier, `x`, which is a variable ranging over $\{0, 1\}$. In the program code, `x` is first assigned 0 or 1 with respective probabilities of $\frac{1}{3}$ and $\frac{2}{3}$, and is then again assigned 0 or 1, but with equal probability.

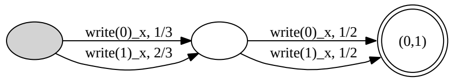
The open program $P_2$, below, is similar to $P_1$ except that the two assignments to `x` are both made by a fair coin:

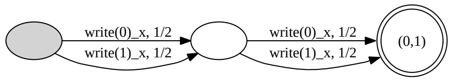
```
void main(var%2 x) { x:=coin; x:=coin }
```

$P_1$ and $P_2$, it turns out, are not equivalent; indeed, it is possible to manufacture a context which (probabilistically) distinguishes them by observing the

[5] More precisely, the automata can have size exponential in the size of the code fragment.

sequence of assignments to `x`.[6] By full abstraction, the complete plays of $P_1$ and $P_2$, captured respectively by the two probabilistic automata depicted below, must therefore differ:





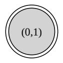
Each automaton consists of three states. Initial states are shaded, and accepting states are doubly circled. Transitions are labelled by the corresponding assignments to `x` together with the associated probabilities.

The probabilistic languages of the two automata are plainly different; for instance, the word $\langle write(0)\_x, write(0)\_x\rangle$ is accepted by the first automaton with probability $\frac{1}{6}$ and by the second automaton with probability $\frac{1}{4}$.

$P_1$ and $P_2$ both contain a free identifier `x`, and are therefore not closed programs. It is of course possible to declare `x` as a local variable instead, as in the following:

```
void main() { int%2 x; x:=coin[0:1/3, 1:2/3]; x:=coin }
```

The above program is closed, and terminates with probability 1; it is therefore equivalent to `void main() { skip }`, and its set of complete plays is captured by the following probabilistic automaton:



The second component of the label on the accepting state, the number '1', represents the probability of termination. This automaton therefore accepts the empty word with probability 1 (and all other words with probability 0).

It should be clear that, if $P_1$ and $P_2$ are modified by making `x` a local variable rather than a free identifier, then no context can possibly distinguish them and they therefore become equivalent (in accordance with one's intuition). Hence the way in which variables are declared, whether as free identifiers or locally, intuitively corresponds to whether they are visible or not to the outside world. We will make use of this idea when we consider the notion of anonymity in Section 6.

The current version of APEX relies on manipulating text files with a library of automata routines. APEX takes as input a text file containing a description of an open probabilistic program which comprises the program type, the list of its free identifiers, and the program code. The output is a probabilistic automaton. After the automata have been generated they can be inspected immediately, or fed to other automata-theoretic procedures such as Tzeng's equivalence-checking algorithm [25].

[6] An instance of such a context could, for example, instantiate the free occurrences of `x` with the sequential composition of a command followed by a variable; in effect, assignments to `x` then induce side-effects, which can be detected by the context.

## 4 Herman's Self-Stabilisation Algorithm

Self-stabilisation is an important area of research in distributed systems that originated with Dijkstra's seminal 1974 paper [7]. Roughly speaking, a self-stabilising system is one that always eventually recovers in finite time from transient faults and operates correctly.

Herman's algorithm is a classical example of a randomised self-stabilisation protocol [9]. Imagine a network of processes, arranged in a ring, with each process possibly holding a token. 'Legitimate' configurations are those in which a token is held by exactly one process. The aim of a self-stabilisation protocol is to guide the network towards legitimate configurations.

Let us assume that each process possesses a distinguished two-valued variable, and let us adopt the convention that a process is deemed to hold a token if its distinguished variable has the same value as that of its immediate right-hand neighbour. (Note that in order for this representation scheme to make sense, there must be an odd number of processes in the network.)

The algorithm works as follows. At every time step, each process determines whether or not it holds a token. If it does, it flips its distinguished variable with probability $\frac{1}{2}$, and otherwise sets its distinguished variable equal to that of its right-hand neighbour. We assume that processes execute synchronously.

What we would like to show is that such a protocol is *correct*, i.e., that it always eventually leads the system to a legitimate, single-token configuration.

To this end, we implemented Herman's algorithm in our probabilistic programming language for various numbers of processes in the network. The code for a 15-process network is given below.

```
void main() {
  int%2 x[15]; int%2 z; int%3 token; int%15 i;
  token:=2;
  while(not (token=1)) do {
    token:=0;
    i:=0;
    z:=x[0];
    while (i+1) do {
      if (x[i]=x[i+1]) then
        x[i]:=coin else x[i]:=x[i+1];
      if (i>0) and (x[i-1]=x[i]) then
        token:=case(token)[1,2,2];
      i:=i+1
    };
    if (x[i]=z) then x[i]:=coin else x[i]:=z;
    if (x[i-1]=x[i]) then token:=case(token)[1,2,2];
    if (x[i]=x[0]) then token:=case(token)[1,2,2]
  }
}
```

Most of the syntax is self-explanatory, perhaps with the exception of the statement `token:=case(token)[1,2,2];`. This is similar to the `switch` construct in C, and is equivalent to

```
if token=0 then token:=1 else
  if token=1 then token:=2 else
    if token=2 then token:=2;
```

In the program, the distinguished variables of processes are held in a 15-element array `x` of two-valued variables. The inner `while` loop simulates the synchronous execution of the network over a single time step. In this loop, the variable `token` is used to count the total number of tokens present in the network, with the value 2 representing 'two or more'. Recall the use of modulo arithmetic so that variables that overflow simply cycle through 0. The outer `while` loop ensures that the code is executed until the network contains just a single token.

Note that our implementation is a closed program of type `void`. It should be clear that the correctness of Herman's algorithm (a single-token configuration is always eventually reached) corresponds to the assertion that our program terminates with probability 1. And indeed, when running APEX, the output is the one-state automaton corresponding to `void main() { skip }` already depicted in Section 3.

We remark that it is trivial to modify the code to model networks with different numbers of processes: it suffices to replace the two occurrences of the number '15' in the first line by whatever other value is desired.

Although all instances of our program ultimately give rise to the same single-state automaton, the computation times of APEX increase with the sizes of the networks modelled. This is not surprising since an $n$-process network has $2^n$ distinct configurations (ignoring symmetries). The intermediate automata generated by APEX reflect this growth, although this is mitigated to some extent by the use of state-space reduction techniques throughout the computation.

## 5 Hibbard's Algorithm and Random Trees

APEX makes it possible to compare various finite-state distribution generators. For instance, one can easily verify that the standard iterative algorithm for simulating a six-sided die using a fair coin is correct.

In this section we analyse a more complicated example, having to do with the average shape of binary search trees generated by sequences of random insertions and deletions. This is a classical problem in the theory of algorithms, in which a central concern is to ensure that the random trees generated within a particular scheme have low average height.

Binary search trees have been used and studied by computer scientists since the 1950s. In 1962, Hibbard proposed a simple algorithm to dynamically delete an element from a binary tree [10]. Moreover, he also proved that a random deletion from a random tree, using his algorithm, leaves a random tree. Although the statement might seem self-evident, we will see shortly that this is not quite the case. More precisely, Hibbard's theorem can be stated as follows: "If $n + 1$ items are inserted into an initially empty binary tree, in random order, and if one of those items (selected at random) is deleted, the probability that the resulting binary tree has a given shape is the same as the probability that this tree shape would be obtained by inserting $n$ items into an initially empty tree, in random order."

Hibbard's paper was remarkable in that it contained one of the first formal theorems about algorithms. Furthermore, the proof was not simple. Interestingly, for more than a decade it was subsequently believed that Hibbard's theorem in fact proved that trees obtained through arbitrary sequences of random insertions and deletions are automatically random, i.e., have shapes whose distribution is the same as if the trees had been generated directly using random insertions only; see [10, 15].

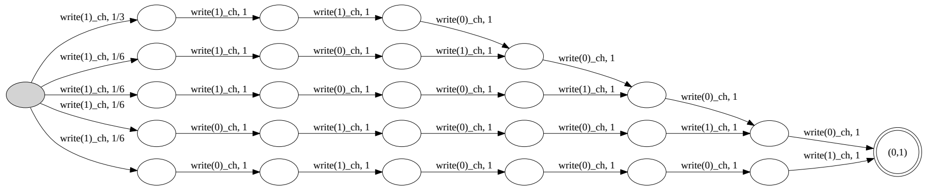
Quite surprisingly, it turns out that this intuition was wrong. In 1975, Knott showed that, although Hibbard's theorem establishes that $n + 1$ random insertions followed by a deletion yield the same distribution on tree shapes as $n$ insertions, we cannot conclude that a subsequent random insertion yields a tree whose shape has the same distribution as that obtained through $n + 1$ random insertions [14].

As Jonassen and Knuth point out, this result came as a shock. In [13], they gave a careful counterexample (based on Knott's work) using trees having size no greater than three. More precisely, they showed that three insertions, followed by a deletion and a subsequent insertion (all random) give rise to different tree shapes from those obtained by three random insertions. Despite the small sizes of the trees involved and the small number of random operations performed, their presentation showed that the analysis at this stage is already quite intricate. This suggests a possible reason as to why an erroneous belief was held for so long: carrying out even small-scale experiments on discrete distributions is inherently difficult and error-prone. For example, it would be virtually impossible to carry out by hand Jonassen and Knuth's analysis for trees of size no greater than five (i.e., five insertions differ from five insertions followed by a deletion and then another insertion), and even if one used a computer it would be quite tricky to correctly set up a bespoke exhaustive search.

Our goal here is to show that such analyses can be carried out almost effortlessly with APEX. It suffices to write programs that implement the relevant operations and subsequently print the shape of the resultant tree, and then ask whether the programs are equivalent or not.

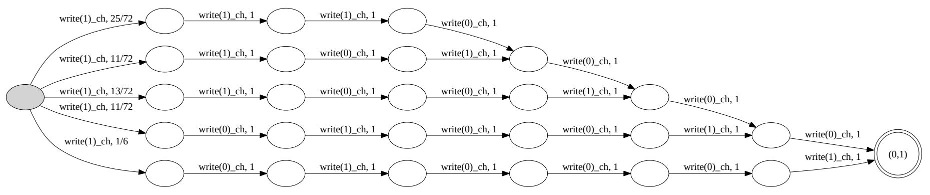
As an example, we describe how to use APEX to reproduce Jonassen and Knuth's counterexample, i.e., three insertions differ from three insertions followed by a deletion and an insertion. Since APEX does not at present support pointers, we represent binary trees of size $n$ using arrays of size $2^n - 1$, following a standard encoding (see, e.g., [5]): the left and right children of an $i$-indexed array entry are stored in the array at indices $2i + 1$ and $2i + 2$ respectively.

It is then possible to write a short program that inserts three elements at random into a tree, then sequentially prints out the tree shape in breadth-first manner into a free identifier `ch`. The actual code is omitted here for lack of space, and can be found in [18]. From this open program, APEX generates the following probabilistic automaton:



The upper path in this automaton, for example, represents the balanced three-element tree shape $\wedge$. The probability that this shape occurs can be determined by multiplying together the weights of the corresponding transitions, yielding a value of $\frac{1}{3}$.

It is likewise straightforward to produce a program implementing three insertions followed by a deletion and an insertion, all of them random—the code can be found in [18]. The corresponding probabilistic automaton is the following:



The reader will note that the balanced three-element tree shape occurs with slightly greater probability: $\frac{25}{72}$. Thus the two programs are indeed not equivalent.

Note that none of this, of course, contradicts Hibbard's theorem, to the effect that the distribution on tree shapes upon performing two random insertions is the same as that obtained from three random insertions followed by a random deletion. The reason is that, although the distribution on tree *shapes* is the same, that on *trees* is not. This is then witnessed by performing an additional random insertion, which in the second case very slightly biases the resulting tree shape towards balance, as compared to the first case.

## 6 The Dining Cryptographers

Anonymity is a key concept in computer security. It arises in a wide range of contexts, such as voting, blogging, making donations, passing on sensitive information, etc. A celebrated toy example illustrating anonymity is that of the 'Dining Cryptographers protocol' [3]. Imagine that a certain number of cryptographers are sharing a meal at a restaurant around a circular table. As the end of the meal, the waiter announces that the bill has already been paid. The cryptographers conclude that it is either one of them who has paid, or the organisation that employs them. They resolve to determine which of the two alternatives is the case, with the proviso that for the former the identity of the payer should remain secret.

A possible solution goes as follows. A coin is placed between each pair of adjacent cryptographers. The cryptographers flip the coins and record the outcomes for the two coins that they can see, i.e., the ones that are to their immediate left and right. Each cryptographer then announces whether the two outcomes *agree* or *disagree*, except that the payer (if there is one) says the opposite. When all cryptographers have spoken, they count the number of *disagree*s. If that number is odd, then one of them has paid, and otherwise, their organisation has paid. Moreover, if the payer is one of the cryptographers, then no other cryptographer is able to deduce who it is.

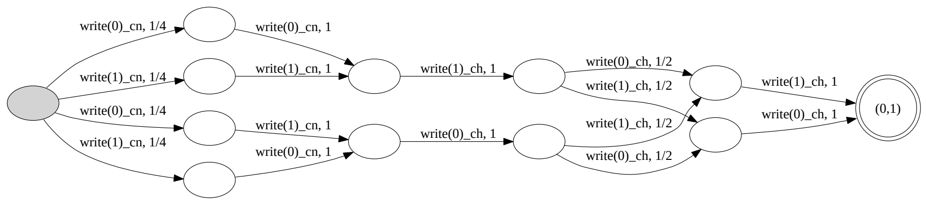
There are many formalisations of the concept of anonymity in the literature. The earliest approaches ignored probabilities and relied instead on nondeterminism; anonymity was then equated with 'confusion', or more precisely with notions of equivalence between certain processes [24]. For example, in the case of the dining cryptographers, every possible behaviour visible to one of the cryptographers (i.e., outcomes of the two adjacent coin flips and subsequent round of announcements) should be consistent with any of the other cryptographers having paid, provided the number of *disagree*s is odd.

A more sophisticated treatment takes probabilities into account. In our example, assuming the coins are fair, it can be shown that the *a posteriori* probability of having paid, given a particular protocol run, is the same for all cryptographers. Note that this does not hold if the coins are biased, which highlights one of the advantages of using probability over nondeterminism. A survey of the literature, as well as an in-depth treatment using process algebra, can be found in [2].

We show how to model the Dining Cryptographers protocol in our probabilistic programming language, and verify anonymity using APEX. Let us consider the case of three cryptographers, numbered 1, 2, and 3, from the point of view of the first cryptographer; the open program below enacts the protocol. This program has a local variable `whopaid` that can be set to 2 or 3, to model the appropriate situation. All events meant to be visible to the first cryptographer, i.e., the outcomes of his two adjacent coins, as well as the announcements of all cryptographers, are written to the free identifiers `cn` and `ch` respectively. (Probabilistic) anonymity with respect to the first cryptographer corresponds to the assertion that the program in which `whopaid` has been set to 2 is equivalent to the program in which `whopaid` has been set to 3.

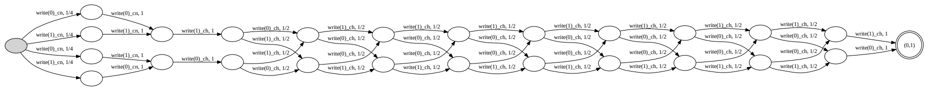
```
void main(var%2 ch, var%2 cn) {
  int%4 whopaid; int%2 first; int%2 right; int%2 left; int%4 i;
  whopaid:=2; first:=coin; right:=first; i:=1;
  while (i) do {
    left := if (i=3) then first else coin;
    if (i=1) then { cn:=right; cn:=left };
    if ((left=right)+(i=whopaid)) then ch:=1 else ch:=0;
    right:=left;
    i:=i+1
  }
}
```

From this code, APEX produces the following probabilistic automaton:



It turns out that setting `whopaid` to 3 in the above program yields precisely the same automaton. The two programs are therefore equivalent, which establishes anonymity of the protocol with three cryptographers.

One can easily investigate larger instances of the protocol, through very minor modifications of the code. For example, the probabilistic automaton below corresponds to an instance of the protocol comprising 10 cryptographers. It is interesting to note that the size of the state space of the automata grows only linearly with the number of cryptographers, despite the fact that the raw cryptographers state space is ostensibly exponential (due to the set of possible outcomes of the coin flips). Note however that this complexity is in our case reflected in the number of *paths* of the automata rather than in the number of their states. In fact, in our experiments (see Figure 1), the state spaces of the intermediate automata as well as the total running times grew linearly as well. This unexpected outcome arose partly from APEX's use of bisimulation reduction throughout the construction, in which most symmetries were factored out.



We can also show that probabilistic anonymity fails when the coins are biased, as described in [18]. Note that thanks to full abstraction, whenever two probabilistic programs are not equivalent, their corresponding probabilistic automata will disagree on the probability of accepting some particular word. This word, whose length need at most be linear in the sizes of the automata, can be thought of as a *counterexample* to the assertion of equivalence of the original programs, and can potentially be used to 'debug' them. In the case at hand, such a word would illustrate why anonymity fails when the coins are biased, albeit only in a probabilistic sense. A would-be spymaster could then return to the programs and attempt to fix the problem.

Although APEX does not at present generate counterexamples in instances of inequivalence, we remark that it would be straightforward and computationally inexpensive to instrument it to do so.

**Related Work.** Although the Dining Cryptographers protocol was proposed almost twenty years ago and has been extensively studied since[7], it had until

[7] Google Scholar lists over 500 papers dealing with the Dining Cryptographers!

recently never been verified[8] in a fully automated way. In the last few weeks, we have become aware of two automated verification instances (other than that proposed in the present paper): [21] and [20].

As explained earlier, assertions of anonymity are most commonly and naturally expressed as *equivalences*; in [24], for example, trace equivalence in (non-probabilistic) CSP is used, whereas [2] is based on bisimulation equivalence for a probabilistic extension of the $\pi$-calculus. Most probabilistic verification engines, however, focus on *model checking*, i.e., whether a particular probabilistic system satisfies a given specification, where the latter is usually given in some (probabilistic) temporal logic such as PCTL.

As regards anonymity, model checking is considerably less convenient than equivalence checking. In [21], for example, the authors establish anonymity of the Dining Cryptographers protocol by considering all possible visible behaviours, and proving for each that the likelihood of its occurrence is the same regardless of the payer. This leads to exponentially large specifications, and correspondingly intractable model-checking tasks.[9] In practice, a proper verification of the protocol can only be carried out for a handful of cryptographers—see the experimental results reported in Figure 1, running on a Fujitsu-Siemens Lifebook P7120 at 1.2 GHz, with 1 GB RAM, under Windows XP. In particular, PRISM takes over an hour to handle 10 cryptographers, and runs out of memory for larger instances. By contrast, we can handle 100 cryptographers in approximately 125 seconds.

The same difficulties beset the framework of [20], which gives an algorithm for PCTL model checking of the probabilistic $\pi$-calculus, along with a PRISM-based implementation. Again, the combinatorial explosion (of both the model and the specification) severely limits the sizes of the protocol instances that can be verified. We believe that the work presented in this section makes a forceful case for the development of probabilistic equivalence checkers alongside model-checking tools.

## 7 Future Work

There are many avenues for further research. We are currently implementing support for pointers, which would enable more flexible modelling of algorithms that use dynamic data structures. We would also like to extend APEX to handle programs that feature *parameterised* random constants, representing undetermined sub-distributions, which can be viewed as a form of nondeterminism. For

[8] By *verification* of the protocol we refer here to the automated handling of instances in which the number of cryptographers is fixed; the *parameterised* verification problem, which deals at once with all possible numbers of cryptographers, is substantially more difficult to achieve in fully automated fashion.

[9] The state space of the underlying Markov chain generated by PRISM also grows exponentially, but this is mitigated by PRISM's use of symbolic representations in the form of MTBDDs.

| # crypt. | PRISM | APEX |
|---|---|---|
| 3 | 4 | 7 |
| 4 | 4 | 8 |
| 5 | 7 | 8 |
| 6 | 39 | 9 |
| 7 | 95 | 9 |
| 8 | 282 | 10 |
| 9 | 964 | 10 |
| 10 | > 1h | 11 |
| 15 | OOM | 13 |
| 50 | OOM | 56 |
| 100 | OOM | 125 |

**Fig. 1.** Dining Cryptographers protocol verification times. Timeout was set at one hour, all other times reported in seconds. OOM indicates 'out of memory'.

instance, in a version of Herman's protocol with biased coins, one could verify termination for all possible biases at once.

To support such an extension, we believe that Tzeng's algorithm for language equivalence of probabilistic automata [25] can be generalised to a randomised polynomial-time algorithm for determining *universal equivalence* of parameterised probabilistic automata: 'Are two automata equivalent for all possible instantiations of their parameters, subject to a set of linear constraints?'

We also aim to extend the capabilities of APEX beyond equivalence checking, by exploiting the probabilistic automata it generates in different ways. Model checking and counterexample generation are the most obvious examples, but refinement checking and performance analysis, among others, would also be very useful.

A more ambitious line of research would consist in extending the current approach to handle concurrency, which would facilitate the modelling of distributed systems and protocols.

**Acknowledgements.** We would like to thank Marta Kwiatkowska, Gethin Norman, and Dave Parker for useful discussions and their help with PRISM, and the London Mathematical Society for financial support.

## References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163:409–470, 2000.
[2] M. Bhargava and C. Palamidessi. Probabilistic anonymity. In *Proceedings of CONCUR*, volume 3653 of *LNCS*, 2005.
[3] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.
[4] F. Ciesinski and C. Baier. LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Proceedings of QEST*. IEEE Computer Society, 2006.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[6] V. Danos and R. Harmer. Probabilistic game semantics. *ACM Trans. Comput. Log.*, 3(3):359–382, 2002.

[7] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[8] J. Esparza and K. Etessami. Verifying probabilistic procedural programs. In *Proceedings of FSTTCS*, volume 3328 of *LNCS*, 2004.

[9] T. Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[10] T. N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM*, 9(1):13–28, 1962.

[11] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proceedings of TACAS*, volume 3920 of *LNCS*, 2006.

[12] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. *Inf. Comput.*, 163(2):285–408, 2000.

[13] A. T. Jonassen and D. E. Knuth. A trivial algorithm whose analysis isn't. *J. Comput. Syst. Sci.*, 16(3):301–322, 1978.

[14] G. D. Knott. *Deletion in Binary Storage Trees*. PhD thesis, Stanford University, 1975. Computer Science Technical Report STAN-CS-75-491.

[15] D. E. Knuth. Sorting and searching. In *Volume 3 of The Art of Computer Programming (first printing)*. Addison-Wesley, 1973.

[16] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[17] K. Larsen and A. Skou. Compositional verification of probabilistic processes. In *Proceedings of CONCUR*, volume 630 of *LNCS*, 1992.

[18] A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell. Verification of probabilistic programs via equivalence checking. *In preparation*.

[19] A. S. Murawski and J. Ouaknine. On probabilistic program equivalence and refinement. In *Proceedings of CONCUR*, volume 3653 of *LNCS*, 2005.

[20] G. Norman, C. Palamidessi, D. Parker, and P. Wu. Model checking the probabilistic $\pi$-calculus. In *Proceedings of QEST*. IEEE Computer Society, 2007.

[21] PRISM case study: Dining Cryptographers. `www.prismmodelchecker.org/casestudies/dining_crypt.php`.

[22] M. O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[23] M. O. Rabin. Probabilistic algorithms. In *Proceedings of the Symposium on Algorithms and Complexity*. Academic Press, 1976.

[24] S. Schneider and A. Sidiropoulos. CSP and anonymity. In *Proceedings of ESORICS*, volume 1146 of *LNCS*, 1996.

[25] W.-G. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM J. Comput.*, 21(2):216–227, 1992.