APEX: An analyzer for open probabilistic programs[*]

Stefan Kiefer[1], Andrzej S. Murawski[2], Joël Ouaknine[1],
Björn Wachter[1], and James Worrell[1]

[1] Department of Computer Science, University of Oxford, UK
[2] Department of Computer Science, University of Leicester, UK

Abstract. We present APEX, a tool for analysing probabilistic programs that are open, i.e. where variables or even functions can be left unspecified. APEX transforms a program into an automaton that captures the program's probabilistic behaviour under all instantiations of the unspecified components. The translation is compositional and effectively leverages state reduction techniques. APEX can then further analyse the produced automata; in particular, it can check two automata for equivalence which translates to equivalence of the corresponding programs under all environments. In this way, APEX can verify a broad range of anonymity and termination properties of randomised protocols and other open programs, sometimes with an exponential speed-up over competing state-of-the-art approaches.

1 Introduction

APEX is an analysis tool for *open* probabilistic programs. Such programs are very well suited to analyse randomised algorithms and, in particular, anonymity in security protocols: they (i) represent the behaviour of an algorithm succinctly for a range of inputs, and (ii) allow to differentiate internal behaviour from externally observable behaviour, so that anonymity can be established by proving that secret information is not externally observable.

APEX's key technology is the use of *game semantics* [2], which provides a compositional translation of open probabilistic programs to *probabilistic automata*. Probabilistic automata [10] are essentially nondeterministic automata whose transitions are decorated with probabilities. A theorem [8] guarantees that two open probabilistic programs are equivalent if and only if the probabilistic automata are *language equivalent*, i.e., accept every word with the same probability. Language equivalence between probabilistic automata reduces to a linear-algebra problem for which efficient algorithms have been developed, see [5] and the references therein. APEX performs both the translation from programs to automata and the language-equivalence check. Thus, given two open probabilistic programs, APEX either proves them equivalent, or provides a word that separates the programs.

APEX has been applied to a range of case studies [6]: it provides the most efficient automatic verification of dining cryptographers to date, an analysis of

[*] Research supported by EPSRC. The first author is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

Hibbard's algorithm for random tree insertion, and of Herman's self-stabilisation algorithm.

Example: The Grade Protocol in APEX. We illustrate the use of APEX by analysing a protocol for a group of students, who have been graded and would like to find out the sum of their grades (e.g., to compute the average) without revealing the individual grades. This is accomplished with the following randomised algorithm. Let $S \in \mathbb{N}$ be the number of students, and let $\{0, \ldots, G-1\}$ ($G \in \mathbb{N}$) be the set of grades. Define $N = (G-1) \cdot S + 1$. We assume that the students are arranged in a ring, as depicted in Figure 1, and that each pair of adjacent students shares a random integer between 0 and $N-1$. Thus a student shares a number l with the student on its left and a number r with the student on its right, respectively. Denoting the student's grade by g, the student announces the number $(g + l - r) \bmod N$. The sum of the announced numbers (mod N) is telescoping, so it equals the sum of all grades. We require that no participant glean anything from the announcements other than the sum of all grades. This correctness condition can be formalised by a specification in which the students make random announcements subject to the condition that the sum of the announcements equals the sum of their grades.

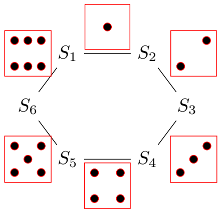


Fig. 1.

```
\\ Implementation

const N := S*(G-1)+1;

grade:int%G, out:var%N |-
   var%(S+1) i; i:=0;
   var%N first; first:=rand[N];
   var%N r; r:=first;
   while(i<S) do {
     var%N l;
     i:=succ(i);
     if(i=S) then
    left:=first
     else
    left:=rand[N];
     out:= (grade + l) - r;
     r:= l;
   } : com
```

```
\\ Specification

const N := S * (G-1) + 1;

grade:int%G, out:var%N |-
    var%S i;
    var%N total;
    i:=1;

    while(i) do {
      total := grade + total;
      var%N r;
      r := rand[N];
      out:=r;
      total := total - r;
      i:=succ(i)
    };
    out:= grade + total: com
```

Fig. 2. Grade protocol: APEX programs.

Figure 2 shows two APEX programs: on the left, the implementation of the grade protocol to be verified against the specification program, given on the right. The input language of APEX is an imperative sequential programming language with a C-like syntax and support for procedures and arrays. There are several constructs to define probability distributions, e.g., the expression `rand[N]` gives a uniform distribution over the numbers $\{0, \ldots, N-1\}$, `coin` is a shorthand for `rand[2]`, and `coin[0:1/4, 1:3/4]` a biased coin. Variables are defined over finite ranges, e.g., `var%N total;` declares a variable over range $\{0, \ldots \mathtt{N}-1\}$. There are internal variables such as the counter `i` which are defined locally, and externally observable program variables through which the program communicates with the outside world. Externally observable variables are declared before the turnstile symbol `|-`; in our example, there are two such variables: the individual grades of students are read from variable `grade`, while `out` is an output variable that announces the random numbers generated by the protocol.

APEX checks whether the programs are equivalent. In this case it finds that they are, so one can conclude that the grade protocol guarantees anonymity.

2 How APEX Works

APEX is implemented in C and OCaml and consists of approximately 18K lines of code. Figure 3 shows APEX's architecture. It has two main components: an automaton construction routine and an equivalence checker.

2.1 Automaton Construction

The automaton constructor builds a probabilistic automaton using game semantics. The construction works at the level of the program's abstract syntax tree (AST). The leaves of the tree correspond to variables and constants, while internal nodes correspond to semantic operations of the language like arithmetic expressions, probabilistic choice, conditionals, sequential composition of commands, and loops. APEX labels each AST node with the automaton that captures its semantics, by proceeding bottom-up and composing the automata of the children. Ultimately, the automaton computed for the root of the AST gives the semantics of the entire program.

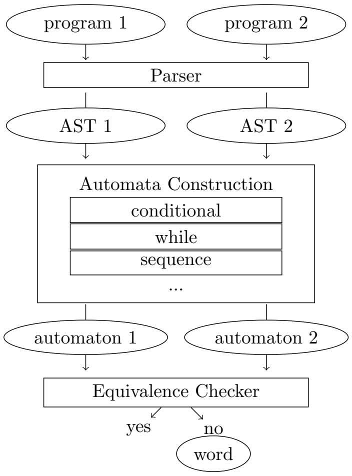
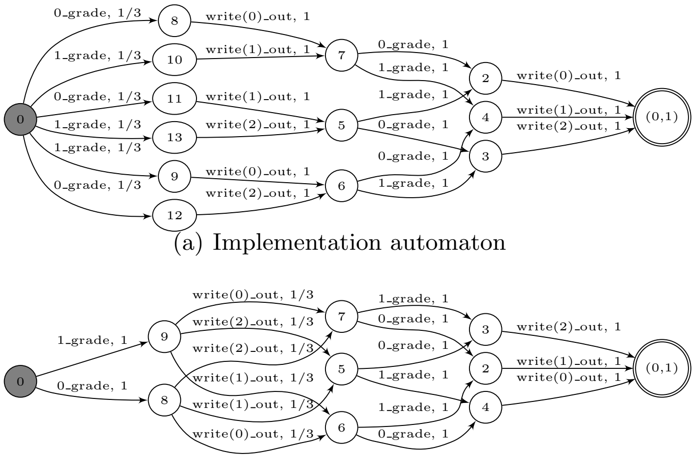


Fig. 3. APEX architecture

Figure 4 shows the probabilistic automata obtained from the programs in Figure 2 (with $S = 2$ and $G = 2$). Transitions in the automaton contain only reads and writes to *observable* variables; e.g., label 1_grade means that value 1 has been read from variable `grade` and $write(2)\_out$ means that value 2 has been written to variable `out`. Actions on *internal* variables are hidden. Each transition is also labeled (comma-separated) with a probability.

2.2 Equivalence Checking

To check equivalence of the input programs, it suffices for APEX to check the corresponding probabilistic automata for language equivalence. If they are not equiv-



(a) Implementation automaton

(b) Specification automaton

Fig. 4. Probabilistic automata for the grade protocol.

alent, APEX presents a counterexample word, which corresponds to a run which is executed with different probabilities in the two programs. APEX uses efficient linear-algebra-based algorithms [5] for the equivalence check and the counterexample extraction.

3 Comparison With Other Tools

PRISM [4] is the leading probabilistic model checker with a large collection of case studies. As regards anonymity, model checking is considerably less convenient than equivalence checking. In [9], 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.[3] In practice, a proper verification of the protocol can only be carried out for a handful of cryptographers [6], while APEX scales to around 800 cryptographers on a state-of-the-art workstation.

Mage [1] is a software verification tool based on game semantics which applies to *non-probabilistic* programs. To our knowledge, APEX is the only game-semantics-based tool for probabilistic programs.

4 Novel Features

We discuss several new features and algorithmic improvements that have not been covered in previous publications [5, 6, 8].

[3] 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.

Variable Binding by Reachability Analysis. Game semantics views a state variable as an automaton that answers read requests and stores values on writes. The subprogram in which the variable lives, i.e. its scope, is an open program in terms of this variable. A variable-binding operator closes this subprogram by synchronous composition of the subprogram with the automaton of the variable. At the semantic level, both variable and scope are represented as automata.

The automata semantics of the binding operator is defined in terms of the synchronous product of the two automata, in which transitions involving reads and writes of the variable are turned into silent ε-transitions which are to be removed e.g. using ε-removal algorithms [7] to obtain the ultimate result.

In the previous implementation of APEX, the product automaton was formed by copying the automaton that represents the scope of the variable k times where k is the number of potential values of the variable (which coincides with the number of states of the automaton that represents the variable). In practice, many of the state-value pairs thus created turn out to be unreachable because the variable typically only takes on a subset of the potential values in its domain at a certain program location. In the new version of APEX, the binding operator computes the automaton by enumerating reachable state-value pairs only. In this way, the peak number of states of the constructed automata is reduced.

Live-Variable Detection. The reachability analysis for variable binding can be further optimised by taking into account liveness information. A variable is live if there is a path to a usage of the variable, and is dead otherwise. At a state in which a variable is dead, the product construction can lump all state-value pairs with the same state, as the value of the variable has become irrelevant.

Liveness analysis is collected by a simple procedure that tracks the liveness of the bound variable with a bit and proceeds backwards from the reading occurrences of the variable in an automaton. This information is subsequently used in the product construction.

Early ε-Removal. In the previous version of APEX, product construction and ε-removal were two distinct phases. Now linear chains of ε-transitions are immediately removed in the product construction, while branching and cyclical structures are left to the full ε-removal routine. By eliminating the 'simple' cases of ε-transitions, subsequent steps such as bisimulation and ε-removal run faster.

Lumping Bisimulation. APEX applies lumping bisimulation to reduce automata size. The bisimulation routine runs frequently during the compositional automata construction. Hence its performance is crucial. Recently we have improved the underlying algorithms. APEX now features a signature-based refinement algorithm [3] that computes a strong bisimulation. Key to its efficiency is to leverage very inexpensive algorithms that compute a coarse pre-partition to which the precise signature-based partition refinement is subsequently applied to obtain the final lumping. Pre-partitioning proceeds in two phases: (1) APEX lumps states according to their minimal distance from an accepting state; whereby the distance is computed by a backwards depth-first search from the accepting states. (2) APEX runs an approximate version of signature-based refinement which utilises hash values of signatures to refine the partition, instead of comparing signatures with each other. Both steps have helped to significantly lower the cost of lumping.

Counterexample generation. The previous version of APEX did not provide diagnostic information in case two programs were inequivalent. Now APEX generates a counterexample word, i.e., a word which the programs accept with a different probabilities. The new feature is enabled by the techniques presented in [5].

Online Tool Demo. We have implemented an online version of APEX which offers a convenient user interface and runs on any device with a recent web-browser:

www.cs.ox.ac.uk/apex

The user can either select from existing case studies, load case studies on the server, or drag & drop into the input window. Automata are displayed as scalable vector graphics (SVG). The view can be zoomed with the mouse wheel and the viewing window can be moved by panning. Further the interface has an equivalence checking mode in which counterexamples are shown. Internally, the tool runs on a server and the dynamic web pages through which the user interacts with APEX are generated by PHP scripts.

References

1. Adam Bakewell and Dan R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, pages 78–92, 2008.
2. V. Danos and R. Harmer. Probabilistic game semantics. *ACM Transactions on Computational Logic*, 3(3):359–382, 2002.
3. Salem Derisavi. Signature-based symbolic algorithm for optimal markov chain lumping. In *QEST*, pages 141–150, 2007.
4. Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, pages 441–444, 2006.
5. S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell. Language equivalence for probabilistic automata. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 526–540. Springer, 2011.
6. A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell. On automated verification of probabilistic programs. In *Proceedings of TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2008.
7. M. Mohri. Generic e-removal and input e-normalization algorithms for weighted transducers. *Int. J. Found. Comput. Sci.*, 13(1):129–143, 2002.
8. A. S. Murawski and J. Ouaknine. On probabilistic program equivalence and refinement. In *Proceedings of CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2005.
9. PRISM case study: Dining Cryptographers.
`www.prismmodelchecker.org/casestudies/dining_crypt.php`.
10. M. O. Rabin. Probabilistic automata. *Information and Control*, 6 (3):230–245, 1963.