Incremental Quantitative Verification for Markov Decision Processes

Marta Kwiatkowska, David Parker, Hongyang Qu

Incremental Quantitative Verification for Markov Decision Processes

Marta Kwiatkowska, David Parker and Hongyang Qu
Department of Computer Science, University of Oxford, Parks Road, Oxford, OX1 3QD, UK
Email: {marta.kwiatkowska, david.parker, hongyang.qu}@cs.ox.ac.uk

***Abstract*—Quantitative verification techniques provide an effective means of computing performance and reliability properties for a wide range of systems. However, the computation required can be expensive, particularly if it has to be performed multiple times, for example to determine optimal system parameters. We present efficient *incremental* techniques for quantitative verification of Markov decision processes, which are able to re-use results from previous verification runs, based on a decomposition of the model into its strongly connected components (SCCs). We also show how this SCC-based approach can be further optimised to improve verification speed and how it can be combined with symbolic data structures to offer better scalability. We illustrate the effectiveness of the approach on a selection of large case studies.**

***Keywords*-Quantitative verification; incremental verification; Markov decision processes; performance analysis; probabilistic model checking**

I. Introduction

In almost all aspects of everyday life, we are reliant on computerised systems: from the controllers found in cars and planes, to the computer networks underlying our communication, transport and finance systems. The prevalence of such systems, combined with their increasing complexity, means that effective methods to assure their reliability and performance are essential. Model-based analysis techniques provide an effective way of achieving this. The systems to be analysed are often inherently stochastic: device components may be failure-prone; messages sent across communication networks may get lost or delayed; and wireless technologies such as Bluetooth and ZigBee use randomisation. Thus, models are typically probabilistic in nature; they are also often extended with time and/or quantities for resources.

Approaches to the analysis of such models range from discrete-event simulation, to analytical methods, to numerical solution, and each have their own strengths and weaknesses. In this paper, we focus on techniques that exhaustively construct probabilistic models and then perform an exact analysis, based on numerical computation. In particular, we consider *quantitative verification*, which is a formal approach for specifying and checking quantitative properties of a system model. The model itself, typically a Markov chain or Markov decision process, is systematically constructed from a formal description in some high-level modelling language. Whereas traditional formal verification techniques are usually applied to check the correctness of systems, quantitative verification can be used to analyse properties such as performance or reliability. A key strength of quantitative verification is that it yields exact results, as opposed to, for example, the approximations produced from simulation-based analysis techniques. In fact, for Markov decision processes, which are the focus of this paper, simulation-based techniques are inappropriate due to the presence of nondeterminism.

Probabilistic temporal logics such as PCTL [1], [2] and its variants are used to formally specify a wide range of system properties, for example "the probability that a data packet has been successfully transmitted within 0.5 seconds", "the probability that both sensors are simultaneously non-operational" or "the expected time taken to execute the protocol". Probabilistic model checkers, e.g. PRISM [3], have been used to apply quantitative verification to a wide range of systems, including communication protocols, security protocols and dynamic power management schemes.

Tools such as PRISM also facilitate investigation of *trends* or *variations* in quantitative properties, for example, to study how changes in the failure probability of an individual component affect overall system reliability, or to select the optimal value for a system parameter to maximise performance. Building on these ideas, quantitative *runtime* verification techniques have been proposed [4], [5], which dynamically monitor a system's behaviour and, through exposed control interfaces, enforce the satisfaction of formally specified constraints on performance or reliability at runtime. As observed in [4], however, the need to repeat the verification process for a range of parameter values (in order to configure system parameters appropriately) can incur significant time and memory overheads.

In this paper, we present *incremental* quantitative verification techniques, which offer improvements in efficiency by re-using existing results when analysing a model to which minor changes have been made. We focus on Markov decision processes (MDPs), which generalise (discrete-time) Markov chains and are well suited to modelling systems such as communication protocols. We target scenarios in which a model needs to be analysed repeatedly and where the probability of certain events occurring is subject to change.

The key idea behind our approach to incremental analysis is to use a decomposition of the model into its *strongly*

connected components (SCCs). Exploiting model structure in this way has already been shown to be effective for an isolated instance of MDP verification [6] but the benefits for reducing work across multiple verifications has not been considered. Furthermore, we present additional optimisations that can be applied when using an SCC-based analysis of an MDP, incrementally or otherwise. First, we show how to reduce the amount of *precomputation* performed: this is an analysis of the underlying graph structure of the MDP that needs to be executed before numerical solution is applied. Secondly, we demonstrate how analysis of the decomposed MDP is amenable to *parallelisation*. We have implemented our techniques, using explicit-state data structures, in an extension of PRISM. Using a selection of large benchmark case studies, we demonstrate that our incremental verification techniques yield impressive speed-ups, and that these are further enhanced by our optimisations.

When implementing verification techniques in practice, there is a need not just for improvements in terms of speed, but also memory consumption. In fact, scalability is arguably the bigger challenge of the two. For this reason, state-of-the-art verification tools such as PRISM often rely on *symbolic* techniques, employing data structures such as binary decision diagrams (BDDs) or multi-terminal BDDs (MTBDDs). These exploit the regularity that is often present in models to provide compact storage and efficient manipulation. In the latter part of this paper, we present a symbolic implementation of SCC-based MDP verification. The main difficulty is the crucial step of identifying SCCs in a model. The classic algorithm to do this, due to Tarjan [7], is not well suited to a symbolic (BDD-based) implementation. Symbolic versions have been proposed [8], [9] but are difficult to adapt to this setting: unlike Tarjan, they do not preserve information about the topological order, and this information is expensive to obtain afterwards with BDDs. We present a customised version of Tarjan's algorithm which resolves this problem. Further experimental results show that, for large models, this new approach is faster than the existing solution engines in PRISM, with only a limited increase in memory usage (linear in the size of the state space).

The remainder of this paper is structured as follows. Below, we briefly review some related work. Section II covers background material on quantitative verification for MDPs, including the SCC-decomposition techniques of [6]. In Section III, we present our SCC-based optimisations for graph-based computation and parallelisation. In Section IV, we describe techniques for SCC-based incremental verification. Section V summarises our experimental results and Section VI discusses our symbolic implementation of SCC-based verification. Section VII concludes the paper.

Related work. In addition to [6], SCC decomposition was proposed for quantitative verification in [10], but for discrete-time Markov chains and with an emphasis on counterexample generation. We are not aware of any work on incremental verification for probabilistic models. For non-probabilistic systems, incremental techniques have been proposed, e.g., [11]–[13]; these focus on speeding up state space generation or checking of functional properties; quantitative properties or numerical computation are not considered.

II. Background

We let $Dist(S)$ be the set of all discrete probability distributions over S, i.e., the set of functions $\mu : S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$.

A. Markov Decision Processes

Markov decision processes (MDPs) are widely used to model systems that exhibit both probabilistic and nondeterministic behaviour. Real-life systems are often inherently stochastic, for example due to the presence of failures, unpredictable delays or randomisation. In addition, nondeterminism may be essential, for example to capture *concurrency*, i.e. the possible interleavings of multiple components operating in parallel, or *underspecification*, where a probability or other parameter is not known or is not relevant.

Formally, an MDP is a tuple $\mathcal{M} = (S, \overline{s}, \mathcal{T}, r)$ where:

- S is a finite set of *states*,
- $\overline{s} \in S$ is the *initial state*,
- $\mathcal{T} : S \to 2^{Dist(S)}$ is a *probabilistic transition function*,
- $r : S \times Dist(S) \to \mathbb{R}_{\geqslant 0}$ is a *reward function*.

The transition probability function $\mathcal{T}$ maps each state $s \in S$ to a finite, non-empty set $\mathcal{T}(s)$ of probability distributions. There are two steps to determine the successor of a state s in the MDP: first, a distribution μ is chosen nondeterministically from the set $\mathcal{T}(s)$; second, the next state s' is chosen randomly according to μ, i.e. the probability of moving to each state s' is given by $\mu(s')$. For simplicity, we do not include action labels in MDPs. Distributions are, however, augmented with *rewards* (sometimes called impulse rewards).

A *path* in an MDP, representing a possible execution of the system being modelled, is a non-empty (finite or infinite) sequence of the form: $s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} s_2 \ldots$ where $s_i \in S$, $\mu_i \in \mathcal{T}(s_i)$ and $\mu_i(s_{i+1}) > 0$ for all $i \geq 0$. We use $\omega(i)$ to denote the $(i+1)$th state in the path ω, i.e. $\omega(i) = s_i$, and $step(\omega, i)$ is the distribution taken in state $\omega(i)$, i.e. $step(\omega, i) = \mu_i$. We let $Path_s$ denote the set of all (infinite) paths starting in state s.

In order to reason formally about the probabilistic behaviour of an MDP $\mathcal{M}$, we require the notion of *adversary* (sometimes called strategy, policy or scheduler), which is one possible resolution of the nondeterministic choices in $\mathcal{M}$. Formally, an *adversary* selects an available distribution in each state based on the history of choices made so far. An adversary A restricts the behaviour of the MDP to a set of paths $Path_s^A \subseteq Path_s$. It also induces a probability

space [14] $Prob_s^A$ over the paths $Path_s^A$. We use $Adv_{\mathcal{M}}$ to denote the set of all possible adversaries for $\mathcal{M}$.

B. Quantitative Verification of MDPs

Usually, properties to be verified against MDPs are expressed in temporal logics, such as PCTL [1], [2] and LTL [15]. Performing verification reduces to the computation of a few key properties of MDPs [2], [16]. The first are the *minimum* or *maximum reachability probabilities*, i.e. the minimum or maximum probability that a path through the MDP eventually reaches a state in some target set $F \subseteq S$, quantified over all possible adversaries:

$$p_s^{\min}(F) = \inf_{A \in Adv_{\mathcal{M}}} p_s^A(F), \quad p_s^{\max}(F) = \sup_{A \in Adv_{\mathcal{M}}} p_s^A(F)$$
$$\text{where: } p_s^A(F) = Prob_s^A(\{\omega \in Path_s^A \mid \exists i \,.\, \omega(i) \in F\})$$

Secondly, we may require the *minimum* or *maximum expected reward* accumulated until target $F \subseteq S$ is reached:

$$e_s^{\min}(F) = \inf_{A \in Adv_{\mathcal{M}}} e_s^A(F), \quad e_s^{\max}(F) = \sup_{A \in Adv_{\mathcal{M}}} e_s^A(F)$$
$$\text{where: } e_s^A(F) = \int_{\omega \in Path_s^A} r_F(\omega)\, dProb_s^A$$

where $r_F(\omega)$ gives, for any path $\omega \in Path_s^A$, the total reward accumulated along ω until a state in F is reached:

$$r_F(\omega) = \begin{cases} \sum_{i=1}^{n_F} r(\omega(i-1), step(\omega, i-1)) & \text{if } \exists j \,.\, \omega(j) \in F \\ \infty & \text{otherwise.} \end{cases}$$

and $n_F = \min\{j \,|\, \omega(j) \in F\}$.

For simplicity, in the remainder of this paper, we will focus on the case of maximum reachability probabilities, i.e. computing $p_s^{\max}(F)$, but our techniques adapt easily to the case of minimum probabilities or expected rewards (with the exception of Section III-A, which applies only to reachability probabilities). Throughout the remainder of the paper, we will assume a fixed MDP $\mathcal{M} = (S, \overline{s}, \mathcal{T}, r)$ and target set F. For clarity, we will abbreviate $p_s^{\max}(F)$ to just $p_s^{\max}$. We use $\underline{p}^{\max}$ to denote the vector of probabilities $p_s^{\max}$ for all states $s \in S$.

Calculation of reachability probabilities (or expected reward values) proceeds in two steps. The first step, referred to as *precomputation*, executes an analysis of the underlying graph of the MDP to identify states that have reachability probabilities of 0 or 1. Second, *numerical computation* is performed to determine values for the remaining states; this can be done with a variety of standard techniques, including value iteration and linear programming. We describe this process (for $p_s^{\max}$) in more detail below.

Precomputation. A graph-based analysis is used to partition the state space S into sets S^n, S^y and $S^?$, containing states s for which the probability $p_s^{\max}$ is 0, 1 or in $(0, 1)$, respectively. In fact, the analysis is performed using two separate algorithms, $Prob0A$ and $Prob1E$:

$$S^n = Prob0A(F), S^y = Prob1E(F), S^? = S \setminus (S^y \cup S^n).$$

Algorithm $Prob0A$ [2] first computes the set of states with maximum probability greater than zero of reaching a state in F. It then returns the complement of this set as S^n. For each state $s \in S^n$, the probability of reaching F is zero under any adversary A.

Algorithm 1 $Prob0A(F)$

```
R := F; done := false
while done = false do
    R' := R ∪ {s ∈ S | ∃μ∈T(s) . ∃s'∈R . μ(s')>0}
    if R' = R then done := true end if
    R := R'
end while
return S\R
```

Algorithm $Prob1E$ [17] uses two nested loops to determine the set S^y of states s for which $p_s^A(F) = 1$ for some adversary A. The outer loop identifies states from which no adversary can make $p_s^A(F) = 1$, and removes those states from S. The inner loop collects states from which one cannot reach a state in F without passing through a state already removed from S.

Algorithm 2 $Prob1E(F)$

```
R := S; done := false
while done = false do
    R' := F; done' := false
    while done' = false do
        R'' := R' ∪ {s ∈ S | ∃μ ∈ T(s) . (∀s' ∈
        S . μ(s') > 0 → s' ∈ R) ∧ (∃s'∈R' . μ(s') > 0)}
        if R'' = R' then done' := true end if
        R' := R''
    end while
    if R' = R then done := true end if
    R := R'
end while
return R
```

Finally, we remark that, although $Prob1E(F)$ computes *all* states for which $p_s^{\max} = 1$, this is not strictly necessary. It suffices to use any set of states S^y that satisfies the condition $F \subseteq S^y \subseteq \{s \in S \mid p_s^{max} = 1\}$. In contrast, for S^n, it may be essential (e.g. when using linear programming) that the set contains *all* states with $p_s^{\max} = 0$.

Value iteration. One way to compute the probabilities $p_s^{\max}$ for the remaining states $s \in S^?$ is to use *value iteration*, an iterative numerical method which can approximate the values up to some desired accuracy. In practice, this method is widely used since it scales well to large MDPs.

Value iteration works by computing a sequence of vectors $\underline{p}^{\max,k}$ for increasing k. Initially, i.e. for the case $k = 0$, we set $p_s^{\max,0}$ to 1 if $s \in S^y$ and 0 otherwise. Then, the kth

iteration of computation is defined, for each $s \in S$, as:

$$p_s^{\max,k} := \begin{cases} 1 & s \in S^y \\ 0 & s \in S^n \\ \max\limits_{\mu \in \mathcal{T}(s)} \sum\limits_{s' \in S} \mu(s') \cdot p_{s'}^{\max,k-1} & s \in S^?. \end{cases}$$

The sequence of vectors $\underline{p}^{\max,k}$ is guaranteed to converge eventually to $\underline{p}^{\max}$. In practice, though, the computation is terminated when a pre-specified convergence criterion is met. One common approach is to check that the maximum (absolute) difference between the corresponding elements of successive vectors is below some fixed threshold δ, i.e.:

$$\max_{s \in S} |p_s^{\max,k} - p_s^{\max,k-1}| < \delta.$$

Another is to check the maximum *relative* difference:

$$\max_{s \in S} |(p_s^{\max,k} - p_s^{\max,k-1})/p_s^{\max,k}| < \delta.$$

Linear programming. An alternative to value iteration is to use linear programming (LP) techniques [16]–[18]. This has both advantages and disadvantages. One positive is that LP yields an exact, rather than approximate, solution (ignoring the possibility of numerical errors due to floating-point arithmetic). Secondly, whereas the running time for value iteration is sensitive to the convergence criteria used, the time required for LP is independent of the desired accuracy. For some models, value iteration can converge slowly, making LP a faster alternative. On the other hand, LP typically does not scale as well as value iteration so its usage is restricted to relatively small models.

As for value iteration, we know that $p_s^{\max}$ is 1 for states in S^y and 0 for those in S^n. For the remaining states $s \in S^?$, we can compute p_s^{max} by solving a linear optimisation problem over the set of variables $\{x_s \mid s \in S^?\}$, letting x_s be p_s^{max} and minimising $\sum_{s \in S^?} x_s$ under the constraints:

$$x_s \geq \sum_{s' \in S^?} \mu(s') \cdot x_{s'} + \sum_{s' \in S^y} \mu(s')$$

for all $s \in S^?$ and all $\mu \in \mathcal{T}(s)$.

C. SCC-based Value Iteration

We now describe an optimisation for value iteration, first presented in [6], based on a decomposition of the MDP $\mathcal{M}$ to be analysed. The first step of this process is to remove *maximal end components* (MECs). An *end component* [17] of $\mathcal{M}$ is a pair $(S', \mathcal{T}')$, where $S' \subseteq S$ and $\mathcal{T}'(s) \subseteq \mathcal{T}(s)$, which is closed and strongly connected, i.e.:

1) $\forall \mu \in \mathcal{T}'(s)$, $\forall s' \in S$. $(\mu(s') > 0 \rightarrow s' \in S')$
2) $\forall s, s' \in S'$, there is a path in $(S', \mathcal{T}')$ from s to s'.

A *maximal* end component is one for which there is no larger end component that contains it. It is known [17] that all states s within an MEC have the same probability value $p_s^{\max}$. Furthermore, we can safely compress each MEC into a single state [6]. In the rest of the paper, we assume that all MECs have already been compressed in this way.

Next, we identify *strongly connected components* (SCCs) in the MDP. An SCC C is a set of states that is strongly connected (there is a path between any two states in C) and maximal (no superset of C is also strongly connected).

SCCs are particularly important in value iteration. Let C be an SCC, and $Pre^*(C) \subseteq S \backslash C$ be the set of states that can reach C, but are not contained within it. Any change of a state's probability value in C affects probability values of all other states in C, as well as those of states in $Pre^*(C)$. Furthermore, until the probability values of the states in C converge, the probability values of states in $Pre^*(C)$ cannot converge. In fact, the computation of probability values for states in $Pre^*(C)$ can be postponed until the probability values in C converge [6].

The set of SCCs in $\mathcal{M}$ forms a partition of its states S. Let $\Pi = \{C_1, \ldots, C_m\}$ be this partition. The *successor set* $Succ(C_i)$ of C_i is the set of states outside C_i that are immediate successors of states in C_i. We say that C_i *depends* on C_j if $Succ(C_i) \cap C_j \neq \emptyset$. As there is no cyclic dependence among SCCs, we generate a *reversed topological order* $\mathcal{C}$ among SCCs such that C_j will appear before C_i in $\mathcal{C}$ if C_i depends on C_j.

SCC-based value iteration processes each SCC separately, according to the ordering $\mathcal{C}$, and then terminates. For each SCC, a sequence of approximations is computed, like for value iteration. For each state s in an SCC, $p_s^{\max,k}$ denotes the value computed for s in the kth iteration and $p_s^{\max,0}$ the initial value for s. For any SCC, we set $p_s^{\max,0}$ to 1 if $s \in S^y$ and 0 otherwise. We also let $p_s^{\max}$ denote the final value for s. Consider now a particular SCC C_i. The first iteration is performed as follows. For each $s \in C_i$:

$$p_s^{\max,1} := \begin{cases} \max\limits_{\mu \in \mathcal{T}'(s)} \sum\limits_{s' \in S} \mu(s') \cdot p'_{s'} & p_s^{\max,0} < 1 \wedge \mathcal{T}'(s) \neq \emptyset \\ p_s^{\max,0} & \text{otherwise.} \end{cases}$$

where $\mathcal{T}'(s) = \{\mu \in \mathcal{T}(s) \mid \exists s' \in Succ(C_i) \; . \; \big(\mu(s') > 0 \wedge p_{s'}^{\max} > 0\big)\}$ and $p'_{s'}$ is $p_{s'}^{\max}$ if $s' \in Succ(C_i)$, or $p_{s'}^{\max,0}$ otherwise.

In the remaining iterations, we only update probabilities for states that are affected by the previous iteration. Other states simply keep their probability from the previous iteration. Algorithm 3 describes the k-th iteration (for $k > 1$), where $p'_{s'} = p_{s'}^{\max}$ if $s' \in Succ(C_i)$; otherwise $p'_{s'} = p_{s'}^{\max,k-1}$. The iteration on C_i terminates at the k-th iteration when X in Algorithm 3 is empty. Note that Algorithm 3 also works when we use δ as a maximum relative difference, e.g., the condition $p_x^{\max,k-1} - p_x^{\max,k-2} \geq \delta$ in Algorithm 3 can be replaced by $\frac{p_x^{\max,k-1} - p_x^{\max,k-2}}{p_x^{\max,k-2}} \geq \delta$.

D. The Tarjan Algorithm for SCC Identification

We conclude this section by describing the process of identifying SCCs. A well-known and efficient method for this is the Tarjan algorithm [7]. Its time and space complexity is linear in the size of the model. The basic idea is to

Algorithm 3 The k-th iteration (for states in C_i)

$X := \{x \in C_i \mid p_x^{\max,k-1} - p_x^{\max,k-2} \geq \delta\}$
for all $x \in X$ **do**
  $Y := \{y \in C_i \mid p_y^{\max,k-1} < 1$ and $\exists \mu \in \mathcal{T}(y) \;.\; \mu(x) > 0\}$
  for all $y \in Y$ **do**
    $\mathcal{T}'(y) := \{\mu \in \mathcal{T}(y) \mid \mu(x) > 0\}$
    $p_y^{\max,k} := \max_{\mu \in \mathcal{T}'(y)} \sum_{s' \in S} \mu(s') \cdot p'_{s'}$
  end for
end for

execute a depth-first search (DFS) on the model, using a stack to store states, which we denote $stack$. During the search, each state s is assigned two values: $s.index$ for the order in which states are visited, and $s.lowlink$ for the smallest index in the SCC containing the state. The second value is changed as more states in the SCC are discovered. The root state is the one in which $s.index = s.lowlink$.

Algorithm 4 is an improved version of the algorithm, based on [19]. In this algorithm, the root node of each SCC is never pushed into the DFS stack in order to save time and space. Note that we convert the probabilistic transition function $\mathcal{T}$ in an MDP into a non-probabilistic transition relation $\mathcal{E}$, i.e. $(s, s') \in \mathcal{E}$ if and only if $\exists \mu \in \mathcal{T}(s) \;.\; \mu(s') > 0$. The Tarjan algorithm starts with a call to the recursive function $tarjan$ with the initial state $\bar{s}$ and initial value 1 for the global variable $index$.

Algorithm 4 $tarjan(s)$

$s.index := index$; $s.lowlink := index$
$index := index + 1$
for all $(s, s') \in \mathcal{E}$ **do**
  if $s'.index$ is undefined **then**
    $tarjan(s')$
    $s.lowlink := \min\{s.lowlink, s'.lowlink\}$
  else if $s' \in stack$ **then**
    $s.lowlink := \min\{s.lowlink, s'.lowlink\}$
  end if
end for
if $s.lowlink = s.index$ **then**
  while $stack \neq \emptyset \wedge TOP(stack).index \geq s.index$ **do**
    $POP(stack)$ and report
  end while
else
  $PUSH(stack, s)$
end if

III. Accelerating SCC-based Value Iteration

The SCC-based version of value iteration, described in Section II-C above, has already been shown to provide a speed-up in the time required for quantitative verification of MDPs [6]. We begin by proposing two further improvements to the technique. Later, in Section V, we will illustrate that these yield further gains in terms of speed.

A. Eliminating Precomputation

The precomputation step presented in Section II-B, which identifies the sets S^y and S^n, often speeds up value iteration, can reduce numerical error and, in the case of S^n, may be required for correctness. It can, however, be time-consuming. Here, we show that precomputation can be eliminated for SCC-based value iteration, whilst retaining most of its advantages. First, we consider S^n, i.e. the identification of states s with $p_s^{\max} = 0$.

Lemma 3.1: A state in an SCC C has $p_s^{\max} = 0$ if and only if $p_{s'}^{\max} = 0$ for all states s' in $Succ(C)$.

Proof: $\Leftarrow$. This is trivial. $\Rightarrow$. Suppose that $p_s^{\max} = 0$ for some $s \in C$ but $p_{s'}^{\max} \neq 0$ for some $s' \in Succ(C)$. There exists a state $s'' \in C$ such that $\exists \mu \in \mathcal{T}(s'') \;.\; \mu(s') > 0$. Apparently, s'' has non-zero maximum probability. By the definition of an SCC, there exists a path $s_1 \ldots s_n$ in C such that $s_1 = s$ and $s_n = s''$. Working back along the path, we deduce that $p_{s_{n-1}}^{\max} > 0$, $p_{s_{n-2}}^{\max} > 0$, …, $p_{s_1}^{\max} = p_s^{\max} > 0$. ∎

According to Lemma 3.1, and the algorithm of Section II-C, $Prob0A$ can in fact be omitted. We simply take S^n to be the empty set; states that have maximum probability 0 will not be considered beyond the first iteration.

Second, we consider S^y, i.e. the identification of states s with $p_s^{\max} = 1$. The following lemma gives us a sufficient check to identify *some* states with $p_s^{\max} = 1$.

Lemma 3.2: Given an SCC C, let suc^0 be the set $\{x \in Succ(C) \mid p_x^{\max} < 1.0\}$. If either:

1) suc^0 is empty and $Succ(C)$ is not, or
2) suc^0 is non-empty and there does not exist a state $s \in C$ such that $\forall \mu \in \mathcal{T}(s) \;.\; \exists s' \in suc^0 \;.\; \mu(s') > 0$

then $p_s^{\max} = 1$ for all states s in C.

Proof: First, recall that there are no MECs in the MDP. By removing every distribution from states in C such that it has a transition reaching suc^0 with probability greater than zero, we obtain a partition of C where each block forms a connected graph and there are no connections between blocks. In each block $B \subseteq C$, each state only has transitions either leading to states in the same block or in $suc^1 = Succ(C) \backslash suc^0$. For all states $s \in B$, if the maximum probability of reaching suc^1 is less than one, there exists an infinite path ω starting at s and only passing states in B. Let $inft(\omega)$ be the set of state-distribution pairs that occur infinitely often in ω. Then according to [17, Theorem 3.2, page 46], $inft(\omega)$ is an end component, which contradicts the premise. ∎

By Lemma 3.2, we can also omit $Prob1E$, replacing it with a simpler check before the first iteration for each SCC (having first initialised S^y to F). This check is considerably

simpler than the computation required for $Prob1E$. Since Lemma 3.2 only gives a sufficient condition, we will not identify all states with $p_s^{\max} = 1$ but, as mentioned above, this does not affect correctness. Furthermore, in our experimental results, this approach always yields the same result as $Prob1E$ and, more importantly, runs much faster.

Finally, although we do not cover the case of minimum reachability probabilities in any detail, we briefly state the following two lemmas which, like Lemma 3.1 and 3.2 above, permit removal of the precomputation step when calculating minimum probabilities.

Lemma 3.3: Given an SCC C and its successor set $Succ(C)$, let $suc^1 = \{x \in Succ(C) \mid p_x^{\max} > 0\}$. If either:

1) suc^1 is empty, or
2) suc^1 is non-empty and there does not exist a state $s \in C$ such that $\forall \mu \in \mathcal{T}(s)\ .\ \exists s' \in suc^1\ .\ \mu(s') > 0$,

then all states in C have minimum probability zero.

Lemma 3.4: A state in an C has minimum probability one iff all states in the successor set $Succ(C)$ of C, have minimum probability one.

B. Parallel Computation

SCC-based value iteration also presents opportunities for *parallelisation*, which is particularly desirable to exploit, given the increasing prevalence of multi-core architectures in mainstream CPU design. The topological order among SCCs provides a natural structure for parallel computation. At any step, an SCC can be processed independently (and thus in parallel), as long as all of its successor set have been processed. To achieve this, we need a queue to store SCCs that are ready to be processed. Initially, all SCCs that have an empty successor set are put in the queue. Each computation thread takes one SCC from the queue to process, and when it is done, it puts SCCs that newly become ready into the queue. The whole process terminates when the queue is empty. Let $\overline{Succ}(C)$ be a copy of the successor set $Succ(C)$ of an SCC C in Π. Algorithm 5 shows the procedure for parallel computation. Note that in the **while** loop, only line 5 can be executed in parallel.

As an additional optimisation, MECs identification can also be parallelised. This is done by by first partitioning into SCCs, then searching each one for MECs in parallel.

IV. Incremental Value Iteration

Our main aim in this paper is to develop *incremental verification* techniques, which accelerate the process of analysing a model that has undergone minor changes, by exploiting the presence of existing verification results. This is a common scenario in practice, for example, when varying a parameter of a model to investigate the effect that this has on overall model performance. Another situation when incremental verification is particularly useful is in the context of using quantitative verification for online monitoring in a self-adaptive framework [4].

Algorithm 5 Parallel processing of SCCs

1: $Queue := \{C_i \in \Pi \mid \overline{Succ}(C_i) = \emptyset\}$
2: $\Pi := \Pi \backslash Queue$
3: **while** $Queue \neq \emptyset$ **do**
4: $\quad scc :=$ the head of $Queue$
5: $\quad$ compute maximum probabilities for states in scc
6: $\quad$ **for all** $C \in \Pi\ .\ \overline{Succ}(C) \cap scc \neq \emptyset$ **do**
7: $\quad\quad \overline{Succ}(C) := \overline{Succ}(C) \backslash scc$
8: $\quad\quad$ **if** $\overline{Succ}(C) = \emptyset$ **then**
9: $\quad\quad\quad \Pi := \Pi \backslash \{C\}$
10: $\quad\quad\quad Queue := Queue \cup \{C\}$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end while**

In this paper, we target cases where the probabilities of some transitions in an MDP undergo changes. We assume, though, that the transition structure of the model remains untouched. This means that transitions with probability one or zero cannot be changed; otherwise, some transitions with non-zero probability would be added or deleted from the model. We use $\mathcal{M} = (S, \overline{s}, \mathcal{T}, r)$ to denote the original MDP and $\overline{\mathcal{M}} = (S, \overline{s}, \overline{\mathcal{T}}, r)$ for the modified one. Notice that only $\mathcal{T}$ is modified.

When some probabilities in $\mathcal{T}$ are changed, it may be unnecessary to recompute probability values for all states. We first identify the set $\overline{\Pi}$ of SCCs that have been affected by the changes. It can be generated using Algorithm 6.

First, $\overline{\Pi}$ is initialised to an empty set. Then, we scan the SCC partition according to the reverse topological order and add C_i to $\overline{\Pi}$ if C_i satisfies one of two conditions:

1) There exists a state $s \in C_i$ such that one distribution from s is involved in the changes;
2) There exists an SCC $C \in \overline{\Pi}$ that C_i depends on.

Algorithm 6 Generate $\overline{\Pi}$

1: $\overline{\Pi} := \emptyset$
2: **for all** $i \in 1, \ldots, m$ **do**
3: $\quad$ **if** $\exists s \in C_i\ .\ \mathcal{T}(s) \neq \overline{\mathcal{T}}(s)$ **or** $\exists C \in \overline{\Pi}\ .\ Succ(C_i) \cap C \neq \emptyset$ **then**
4: $\quad\quad \overline{\Pi} := \overline{\Pi} \cup \{C_i\}$
5: $\quad$ **end if**
6: **end for**

Let $p_s^{\max}$ be the maximum probability for state s computed previously on $\mathcal{M}$ and $\overline{p}_s^{\max}$ the one we need to compute after the changes occur. The SCC-based value iteration algorithm of Section II-C can be adapted to handle changes in probabilities by replacing Π by $\overline{\Pi}$ and initialising

$\overline{p}_s^{\max}$ as follows:

$$\overline{p}_s^{\max,0} := \begin{cases} 1 & s \in S^y \\ 0 & s \in S^n \\ p_s^{\max} & s \in S^? \text{ and } s \in C \text{ for some } C \in \Pi \backslash \overline{\Pi} \\ 0 & \text{otherwise} \end{cases}$$

In addition, before we recompute the probability for an SCC C in $\overline{\Pi}$, we perform a test on its successor set $Succ(C)$. This test checks the following conditions:

1) for every state $s \in Succ(C)$, its probability is not affected by the changes, i.e.:

$$\forall s \in Succ(C) \;.\; \overline{p}_s^{\max} = p_s^{\max}, \tag{1}$$

2) all distributions from a state in C are not affected by the changes, i.e.:

$$\forall s \in C \;.\; \overline{\mathcal{T}}(s) = \mathcal{T}(s).$$

If both conditions hold, there is no need to perform recomputation in this SCC, i.e.:

$$\forall s \in C \;.\; \overline{p}_s^{\max} = p_s^{\max}.$$

Although the above test can eliminate unnecessary recomputation for SCCs that might be affected by the changes, condition (1) is quite restrictive since it requires all states in the successor set to have the same probability as before the changes occurred. Recomputation is executed even if, for all states in $Succ(C)$, there are only tiny changes, e.g., $|\overline{p}_s^{\max} - p_s^{\max}| \in (0, \epsilon)$ for some small $\epsilon > 0$.

In this case, the change in the probability for a state in C is bounded by ϵ with respect to its original value. If ϵ is less than the required accuracy, we can use $p_s^{\max}$ as $\overline{p}_s^{\max}$ for state s in C, which speeds up the recomputation by introducing a small approximation error. Lemma 4.1 formalises this idea.

Lemma 4.1: 1) If the condition $\overline{p}_s^{\max} = p_s^{\max}$ in condition (1) is replaced by $|\overline{p}_s^{\max} - p_s^{\max}| < \epsilon$ and the test succeeds, then:

$$\forall x \in C \;.\; |\overline{p}_x^{\max} - p_x^{\max}| < \epsilon. \tag{2}$$

2) If condition $\overline{p}_s^{\max} = p_s^{\max}$ is replaced by $|\frac{\overline{p}_s^{\max} - p_s^{\max}}{p_s^{\max}}| < \epsilon$ and the above test succeeds, then:

$$\forall x \in C \;.\; |\frac{\overline{p}_x^{\max} - p_x^{\max}}{p_x^{\max}}| < \epsilon. \tag{3}$$

Proof: Consider the base case where $s_0 \xrightarrow{p} s_1$ and $s_0 \xrightarrow{1-p} s_2$. Let v_0, v_1 and v_2 be the probability values in state s_0, s_1 and s_2 respectively. Note that $v = p \cdot v_1 + (1-p) \cdot v_2$. Let v_0', v_1' and v_2' be the new probability value for s_0, s_1 and s_2 respectively after some probabilities in the model are changed (but p is not changed). Here, we discard the absolute value in formulae (2) and (3) and consider $v_1' \geq v_1$ and $v_2' \geq v_2$ only.

Maximum absolute difference. If $v_1' - v_1 < \epsilon$ and $v_2' - v_2 < \epsilon$, we have:

$$\begin{aligned} v_0' - v_0 &= \big(p \cdot v_1' + (1-p) \cdot v_2'\big) - \big(p \cdot v_1 + (1-p) \cdot v_2\big) \\ &= p \cdot (v_1' - v_1) + (1-p) \cdot (v_2' - v_2) \\ &< p \cdot \epsilon + (1-p) \cdot \epsilon = \epsilon. \end{aligned}$$

Maximum relative difference. Assume $\frac{v_1' - v_1}{v_1} < \epsilon$ and $\frac{v_2' - v_2}{v_2} < \epsilon$. We rearrange the inequalities to obtain:

$$v_1' < (1+\epsilon) \cdot v_1 \text{ and } v_2' < (1+\epsilon) \cdot v_2.$$

Then we have:

$$\begin{aligned} v_0' - v_0 &= \frac{\big(p \cdot v_1' + (1-p) \cdot v_2'\big) - \big(p \cdot v_1 + (1-p) \cdot v_2\big)}{p \cdot v_1 + (1-p) \cdot v_2} \\ &= \frac{p \cdot (v_1' - v_1) + (1-p) \cdot (v_2' - v_2)}{p \cdot v_1 + (1-p) \cdot v_2} \\ &< \frac{p \cdot \big((1+\epsilon) \cdot v_1 - v_1\big) + (1-p) \cdot \big((1+\epsilon) \cdot v_2 - v_2\big)}{p \cdot v_1 + (1-p) \cdot v_2} \\ &= \frac{p \cdot \epsilon \cdot v_1 + (1-p) \cdot \epsilon \cdot v_2}{p \cdot v_1 + (1-p) \cdot v_2} \\ &= \frac{\epsilon \cdot \big(p \cdot v_1 + (1-p) \cdot v_2\big)}{p \cdot v_1 + (1-p) \cdot v_2} = \epsilon. \end{aligned}$$

Note that if $s_1 = s$ or $s_2 = s$, the above reasoning still holds. For example, let $s_1 = s$. We have $v = p \cdot v + (1-p) \cdot v_2$, which is simplified to $v = v_2$. If $v_2' - v_2 < \epsilon$, then $v' - v < \epsilon$; if $\frac{v_2' - v_2}{v_2} < \epsilon$, then $\frac{v' - v}{v} < \epsilon$.

Lemma 4.1 is proved by generalisation of the base case. ■

In practice, we can use δ, the maximum absolute difference or maximum relative difference, as ϵ, or a smaller value than δ to increase the accuracy but possibly also increase computation time.

V. Experiments

We have implemented the techniques described in the previous sections, using explicit-state data structures, in an extension of PRISM [3] and investigated performance on several case studies. The first is a model of the Zeroconf network configuration protocol for allocating IP addresses in a local network [20]. We compute "the maximum probability of the protocol correctly configuring an IP address within time T". The second is a model [21] of the shared coin protocol used in the randomised consensus algorithm of Aspnes & Herlihy [22]. The algorithm allows N processes in a distributed network to reach a consensus. We compute "the maximum probability of terminating without consensus being reached". The third case study is a model of the IEEE 802.11 Wireless LAN [23], featuring two stations sending data over a shared channel, each with a backoff counter of size N. We compute "the maximum probability of the backoff counters for both stations reaching their maximum value". All models can be found in the PRISM case study

repository [24], along with details of any model parameters not explained here (e.g. K for Zeroconf, K for Consensus)

The experiments were performed on an AMD Phenom(tm) 9600B Quad-Core Processor with 8GB memory running Fedora 12 x86_64 Linux. Our experimental results are presented in two parts. The first covers the optimisations to accelerate SCC-based value iteration from Section III; the second focuses on the incremental quantitative verification techniques of Section IV.

Accelerating SCC-based value iteration. First, we compare running times, on the various case studies, of our implementation of three approaches:

1) the original version of value iteration,
2) SCC-based value iteration with precomputation [6],
3) SCC-based value iteration without precomputation, including both sequential and parallel versions (parallel uses 4 threads, one per core).

The times are shown in the columns 'Original', 'SCC pre', and 'SCC no-pre' in Table I. For the first, we report the time spent on the precompution and value iteration phases in addition to the total time required. For 'SCC pre', we only give the total running time, as the precomputation time is in general the same as for 'Original'. For the two 'SCC no-pre' approaches, i.e. sequential and parallel, we give the time for computing SCCs (including identification and compression of maximal end components), which is required for both approaches, and for SCC-based value iteration.

In all cases, our approach (SCC-based value iteration without precomputation) is much faster than the original one. The time to generate the SCC partition pays off, even though it could take longer than SCC-based value iteration itself. In many examples, though, the entire SCC-based approach is faster than even just the value iteration phase of the original. The precomputation phase accounts for a large part of the running time in both the original version and the SCC-based approach of [6]; thus eliminating it proves to be a good strategy. In general, the gain from our approach is more significant for larger state spaces.

The parallel version also shows improvement with respect to the sequential one: in the best case, the speed up is about 2.5. Although this is lower than the number of threads, numerical computations such as value iteration are known to be hard to parallelise so this remains a very encouraging result. There are several factors preventing further speed-ups for the parallel version. The major ones are: (1) at some point in the process, there are fewer independent SCCs than threads; (2) the synchronisation overhead is comparably heavy for SCCs that contain only one state (as, for example, in the Consensus models); In addition, the implementation could be further tuned to alleviate memory contention among threads.

Incremental value iteration. To demonstrate the incremental verification algorithm without bias, we randomly choose three states that are not in any MEC, and have a distribution with probabilistic choices. For each state s, we pick such a distribution $\mu \in \mathcal{T}(s)$ and modify the probability distribution as follows. Assume there exist m ($m > 1$) states $s_1, \ldots, s_m \in S$ such that $\mu(s_i) > 0$ for $1 \leq i \leq m$. The new distribution μ' in a is such that, for $1 \leq i \leq m-1$, we keep half of the value, i.e., $\mu'(s_i) = \mu(s)/2$; for $i = m$, we increase the value such that $\mu'(s_m) = \mu(s_m) + \sum_{i=1}^{m-1} \mu(s_i)/2$.

Times for the incremental value iteration algorithm, described in Section IV, are reported in the final two columns of Table I. This includes both the sequential and parallel versions. We do not consider the time for SCC computation, since this does not need to repeated. Even when ignoring this, we see that the times for incremental value iteration represent significant speed-ups compared to the non-incremental (SCC-based) version: they are always faster, up to 50 times faster in some cases. The sequential version works particularly well; for models where a small number of SCCs need to be updated, the gains for the parallel version are less impressive.

VI. Symbolic SCC-based Verification

In the previous sections, we have demonstrated that the SCC-based incremental verification can be very fast. A problem with the implementation, however, is that the explicit-state data structures used to store the state space and transition relation can limit the size of models that can be handled. A successful approach for alleviating this in the context of verification is to use *symbolic* implementations, based on binary decision diagrams (BDDs) [25] and extensions such as multi-terminal BDDs (MTBDDs).

A problem here, though, is that the Tarjan algorithm for identifying SCCs is known to be poorly suited to symbolic implementation. Various SCC decomposition algorithms have been proposed, specifically for implementation with BDDs [8], [9]. Unfortunately, they do not explore SCCs in reverse topological order, and it is very slow to generate this order once the SCCs are stored as BDDs.

In this section, we adapt the Tarjan algorithm to the case where model information is stored using BDDs. We omit here low-level details of how BDDs can be used to represent and manipulate sets of states and transition relations (see e.g. [26]). Here, it suffices to know that some operations are efficient in this form and others are not. For example, some operations in the original Tarjan algorithm cannot be performed efficiently with BDDs, notably association and update of an integer index to a state. We propose a novel *hybrid* adaption of the algorithm that combines symbolic and explicit-state data structures. Keeping overhead to a minimum for efficiency is non-trivial. We maintain:

- the non-probabilistic transition relation $\mathcal{E}$ and the union $allsccs$ of all visited SCCs, stored as BDDs;

Table I
PERFORMANCE COMPARISON FOR SCC-BASED TECHNIQUES.

Model	Parameters	States	Original			SCC pre	SCC no-pre			Incremental	
			Total time (s)	Precomp. time (s)	Val. iter. time (s)	Total time (s)	SCC comp. time (s)	Sequential time (s)	Parallel time (s)	Sequential time (s)	Parallel time (s)
Zeroconf (K, T)	$1, 10$	292,733	129	121	8.0	126	2.4	10.4	6.0	2.1	1.2
	$1, 14$	422,636	184	173	11	180	3.4	14.8	6.4	2.4	2.0
	$2, 10$	665,567	368	334	34	360	5.7	27.2	11.5	5.3	4.4
	$2, 14$	1,061,771	605	548	57	577	13.5	63.3	24.4	9.1	8.6
	$3, 10$	949,912	614	555	59	576	9.2	52.3	21.4	11.1	8.9
	$3, 14$	1,735,014	1,143	1,035	108	1,088	16.0	126	52.4	12.4	18.6
	$4, 10$	976,247	733	663	70	706	9.5	53.9	22.1	20.5	12.2
	$4, 14$	2,288,771	1,768	1,606	162	1,659	27.4	191	78.4	24.2	28.4
Consensus (N, K)	$2, 4$	9,062	4.1	2.4	1.7	2.6	0.1	0.6	0.5	0.09	0.1
	$2, 8$	16,870	33.2	13.5	19.7	15.1	0.2	1.8	1.7	0.8	0.7
	$2, 12$	24,678	112	43.1	68.1	47.2	0.3	5.3	3.7	0.03	0.03
	$2, 16$	32,486	237	91.2	146	104	0.3	11.1	8.4	11.2	5.7
	$2, 20$	40,294	453	177	276	195	0.3	19.4	12.8	14.2	7.0
	$3, 1$	72,9337	74.1	62.4	12.3	69.1	6.7	10.5	22.1	0.2	14.7
	$3, 2$	1,418,545	358	269	89.2	297	13.6	16.8	25.0	0.4	13.3
	$3, 3$	2,259,817	1,085	798	297	844	19.9	30.9	25.0	14.0	13.3
WLAN (N)	2	28,480	0.9	0.9	0.1	1.1	0.2	0.2	0.2	0.1	0.1
	3	96,302	4.2	4.0	0.2	4.8	0.7	0.8	0.7	0.2	0.3
	4	345,000	23.7	22.8	0.8	25.0	1.4	3.5	2.2	0.8	1.3
	5	1,295,218	144	140	3.7	152	10.2	16.4	12.8	0.9	6.8

- a stack $stack$ and hash table M, used during depth-first search, whose size is linear in the number of states.

Generation of $\overline{\Pi}$ for incremental verification with BDDs is not a simple task either. Algorithm 6 is not efficiently implementable with BDDs due to the condition $\exists C \in \overline{\Pi} \ . \ Succ(C_i) \cap C \neq \emptyset$. It requires that, in each iteration of the **for** loop in Algorithm 6, we scan the intermediate $\overline{\Pi}$ to decide if C_i needs to be included in $\overline{\Pi}$. For explicit-state data structures, this is preferable because it saves memory with very little time cost. However, it is better to generate a (sparse) matrix T to store the relation between SCCs. An entry $T[i, j] = 1$ means that C_i depends on C_j. Thus, if C_j is included in $\overline{\Pi}$, all C_i such that $T[i, j] = 1$ are included in $\overline{\Pi}$ too. Note that T can also be encoded symbolically in order to save space. This needs the following extra variables:

- A hash table M_2 to store the root index of each SCC. In the hash table, the root index is the key, as it is unique among all SCCs, and the pointer to BDD for the SCC is the value.
- An MTBDD M_3 to store the root index ($vlowlink$) of each state. A hash table can be used for the same purpose, but would use more space than an MTBDD.

The adapted Tarjan algorithm, which we call the *hybrid Tarjan algorithm*, begins with a call to the recursive function $hybrid\_tarjan$, shown in Algorithm 7, from the initial state with $index = 1$. The lines shaded grey are used to compute T. In Algorithm 7, x and y are integers, v, v' w are BDDs, each of which represents a single state, and scc is a BDD storing the set of states in the current SCC. $M[v]$ represents the corresponding value for the hash key v. Here we utilise a feature of most BDD implementations (including CUDD, which we use): equivalent BDDs are guaranteed to have the same pointer in memory. Thus, the pointer is used as the hash key for the BDD v. $M[v] = NULL$ means that the key v cannot be found in the table and $M[v] := NULL$ denotes that the key and its value are deleted from the hash table.

In the original Tarjan algorithm, each state v is associated with two values for the index of v and the minimum index $lowlink$ among the states in the SCC containing v. To reduce memory consumption, states that have already been identified in some SCCs are stored in $allsccs$, and the hash table M_2 only stores the attribute for the current state and states in the stack. As indicated in [19], only one attribute is actually needed in an elegant implementation. Indeed, only the value $lowlink$ is stored in the hash table. For the current state v, its attributes are stored in the local variables $vindex$ and $vlowlink$; the value $vlowlink$ from its successor states is obtained from the return value of function $hybrid\_tarjan$.

Theorem 6.1: The hybrid Tarjan algorithm partitions a graph $\mathcal{E}$ into SCCs correctly.

Proof: The key idea is to prove that $vlowlink$ is computed correctly with respect to Algorithm 4 when the **for** loop terminates. First of all, notice that $M[v]$ and $vlowlink$ for the current state v in Algorithm 7 cannot be increased once they are initialised. For each successor state v', there are three possibilities:

- v and v' belong to different SCCs and v' is not explored before v. In this case, v' is the root of the SCC it belongs to. In Algorithm 4, this is characterised as $v.lowlink < v'.lowlink$, and therefore, $v.lowlink :=$

Algorithm 7 *hybrid_tarjan*(v)

```
1: vlowlink := index; M[v] := index
2: vindex := index; index := index + 1
3: succ := ∅
4: for all (v, v') ∈ E do
5:    x := 0
6:    if M[v'] = NULL then
7:       if v' ∉ allsccs then
8:          x := hybrid_tarjan(v')
9:       end if
10:   else
11:      x := M[v']
12:   end if
13:   if x > 0 ∧ vlowlink > x then
14:      M[v] := x; vlowlink := x
15:   else
16:      if x = 0 then succ := succ ∪ {M3[v']} end if
17:   end if
18: end for
19: if vlowlink = vindex then
20:   vlowlink := 0; M[v] := NULL; scc := {v}
21:   M3[v] := vindex
22:   while stack ≠ ∅ ∧ M[TOP(stack)] ≥ vindex do
23:      w := TOP(stack); POP(stack)
24:      for all k such that T[M[w], k]=1 do
25:         T[M[w], k] := 0; T[vindex, k] := 1
26:      end for
27:      M[w] := NULL; scc := scc ∪ {w}
28:      M3[w] := vindex
29:   end while
30:   allsccs := allsccs ∪ scc
31:   M2[vindex] := scc
32: else
33:   PUSH(stack, v)
34: end if
35: for all k ∈ succ do T[vindex, k] := 1 end for
36: return vlowlink
```

$v.lowlink$ in line 6. In Algorithm 7, x gets value zero in line 8, as $vlowlink$ for v' is set to zero in line 20. Hence, $vlowlink$ for v does not change its value by the **if** statement in line 13-15.

- v and v' belong to different SCCs and v' is explored before v. In Algorithm 4, v' is not in S because the SCC it belongs to was deleted from S by the **if** statement in line 11-17. Thus, the value of $v.lowlink$ is not changed, as $v'.lowlink$ is defined and $v' \notin S$. In Algorithm 7, this is characterised as $M[v'] = NULL \wedge v' \in allsccs$, and therefore, $vlowlink$ for v keeps its value.
- v and v' belong to the same SCC, which indicates that $vlowlink := 0$ in line 20 of Algorithm 7 is not triggered (when processing v'). Therefore, Algorithms 7 and 4 behave in the same way: the condition "$v'.index$ is undefined" in Algorithm 4 is equivalent to $M[v'] = NULL \wedge v' \notin allsccs$ in Algorithm 7, and $v' \in S$ is equivalent to $M[v'] \neq NULL$. Moreover, we have $x > 0$ under both conditions, as $vlowlink$ is not reset to zero, and therefore the **if** statement in lines 13-15 mimics the function $\min$ in Algorithm 4.

■

The SCCs computed by the hybrid Tarjan algorithm are stored symbolically, which make it impossible to adapt Algorithm 3 to compute each iteration efficiently. This is because it requires access to individual elements of the SCCs, which is inefficient for BDD-based data structures. Our approach is to generate a corresponding explicit-state data structure: a sparse matrix. This can be done relatively efficiently and is then amenable either to value iteration, or in fact solution via linear programming. Because, these matrices are typically small, here we choose to solve an LP problem (in our implementation, we use the ECLiPSe Constraint Logic Programming system with the COIN-OR CBC/CLP solver for this). We also employ an additional optimisation: we treat trivial SCCs, containing a single state without self-loops, as a special case. Probabilities for these can be computed quickly and easily using value iteration on the symbolic data structure.

It is also interesting to note that, to speed up SCC decomposition using BDDs, it is preferable to perform precomputation *before* applying the hybrid Tarjan algorithm; this is the opposite situation to the explicit-state data structure case. This is because precomputation is more efficient when using BDDs and reduces the number of states that need to be explored by the hybrid Tarjan algorithm.

A. Experimental Results

Table II shows experimental results for two of the three previous case studies: WLAN and Consensus[1]. We compare the time for computing maximum reachability probabilities using the standard verification engines in PRISM ("Sparse", "Hybrid", and "MTBDD"), as well as for SCC-based verification using the hybrid Tarjan algorithm ("Hybrid Tarjan" in the table). For the latter, we also report the time spent generating SCCs by the hybrid Tarjan algorithm, which is included in the total time. Model construction time is the same for all cases and therefore not reported in the tables. Since the performance of SCC decomposition by the hybrid Tarjan algorithm is of independent interest, we compare it with two BDD-based algorithms for SCC decomposition: SCC-Find [9] and Lockstep [8]. The decomposition times (without considering reverse topological order) using these algorithms are given in column 'BDD SCC comp.'.

[1]We omit the Zeroconf model from these experiments because some SCCs are too large to solve using LP.

Table II
PERFORMANCE COMPARISON FOR THE HYBRID TARJAN ALGORITHM.

Model	Parameter	States	Sparse Total time (s)	Hybrid Total time (s)	MTBDD Total time (s)	Hybrid Tarjan		BDD SCC comp.	
						SCC comp. time (s)	Total time (s)	SCC-Find time (s)	Lockstep time (s)
WLAN (N)	2	28,480	1.51	1.62	1.63	0.03	0.39	0.11	0.22
	3	96,302	3.63	4.09	4.27	0.15	1.22	0.51	0.57
	4	345,000	10.1	14.5	14.6	0.33	3.00	1.63	2.16
	5	1,295,218	32.7	60.8	48.6	1.06	7.14	3.68	5.81
	6	5,007,548	118.4	313.1	172.1	2.31	16.0	9.55	18.8
Consensus (N, K)	$2, 4$	9,062	4.33	5.63	72.6	0.07	11.1	0.18	0.11
	$2, 8$	16,870	17.3	25.3	554.2	0.14	12.5	0.35	0.16
	$2, 12$	24,678	38.5	62.7	1,773.9	0.20	14.3	0.44	0.21
	$2, 16$	32,486	71.7	127.7	4,163.7	0.28	16.8	0.68	0.35
	$2, 20$	40,294	114.5	215.8	7,793.4	0.33	19.3	0.79	0.35
	$3, 1$	729,337	119.9	148.3	209.0	2.24	356.6	5.47	8.71
	$3, 2$	1,418,545	367.8	552.3	1,068.0	4.65	381.9	8.89	10.1
	$3, 3$	2,259,817	570.4	1,163.1	2,887.6	5.47	405.5	11.5	11.2
	$3, 4$	3,253,153	934.8	2,361.9	6,547.4	7.50	429.1	13.6	12.6

Table III
PERFORMANCE COMPARISON FOR HYBRID TARJAN WITH REWARDS.

N, K	States	Sparse Total time (s)	MTBDD Total time (s)	Hybrid Tarjan	
				SCC comp. time (s)	Total time (s)
$3, 2$	1,806	0.225	46.4	0.11	12.5
$3, 4$	2,894	0.505	286.1	0.17	12.5
$3, 8$	5,070	2.15	2,018.6	0.26	13.2
$3, 12$	7,246	6.14	6,561.9	0.45	13.8
$3, 16$	9,422	14.5	16,158.1	0.48	14.7
$3, 20$	11,598	26.4	-	0.59	15.4
$4, 2$	7,478	0.81	-	0.65	55.3
$4, 4$	12,406	2.21	-	1.25	58.7
$4, 8$	22,262	10.9	-	1.85	63.8
$4, 12$	32,118	32.9	-	2.68	69.5
$4, 16$	41,974	76.0	-	3.99	75.6
$4, 20$	51,830	150.1	-	5.39	81.4
$5, 2$	30,166	2.30	-	3.62	235.9
$5, 4$	50,454	8.22	-	6.72	250.6
$5, 8$	91,030	44.0	-	11.8	286.5
$5, 12$	131,606	134.7	-	15.9	314.2
$5, 16$	172,182	311.4	-	25.6	356.4
$5, 20$	212,758	605.8	-	29.1	389.9

In addition to reachability probabilities, we extend the hybrid Tarjan algorithm and SCC-based LP computation to compute maximum expected reward properties. As mentioned earlier, this is a straightforward adaption. Experimental results for the Consensus study and the property "the maximum expected steps for the first K rounds" are shown in Table III. Note that: (1) the model is slightly different from the one used for reachability probabilities so the number of states differs; (2) PRISM's "Hybrid" engine does not support this class of properties so is not compared to.

We also consider an additional case study to illustrate computation of expected reward properties: a model of the IEEE 1394 FireWire Root Contention Protocol [27]. This is a leader election algorithm for a multimedia bus; the property we check is "the maximum expected time to elect a leader". The model has two parameters, *delay* and *fast*. Figure 1 shows the total running time for the hybrid Tarjan algorithm and PRISM's sparse engine, with three different values of *fast* and varying values of *delay*.

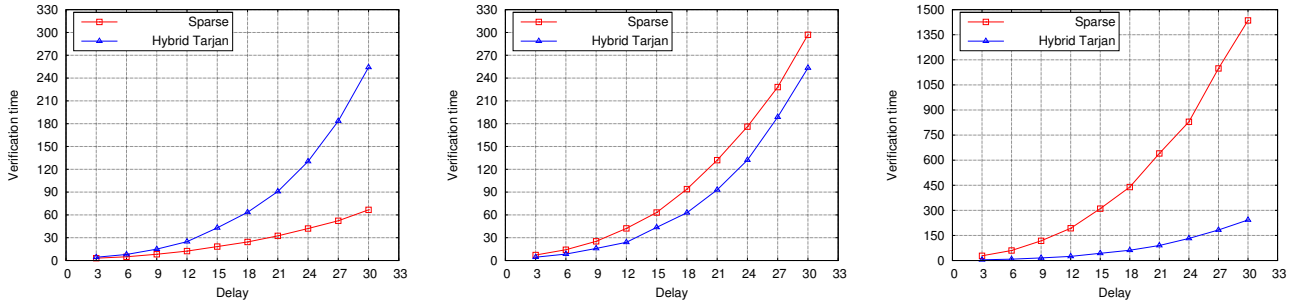
The experimental results for computation of both maximum reachability probabilities and maximum expected rewards clearly demonstrate the advantage of the new approach, which outperforms the other PRISM engines in most cases. In Figure 1, we observe trade-offs between the contrasting techniques for different model parameter values. More precisely, the SCC-based approach becomes more beneficial when value iteration is slow to converge in the full model.

Crucially, we are also able to handle larger models than for the explicit-state implementation presented earlier in the paper. Lastly, we note that the hybrid Tarjan algorithm performs well irrespective of whether we are applying incremental verification: this means it is of independent benefit, for general quantitative verification purposes.

Finally, we note that the hybrid Tarjan SCC decomposition outperforms the existing BDD-based algorithms implemented in PRISM.

VII. CONCLUSION

We have presented techniques for optimising quantitative verification of MDPs, based on a decomposition into strongly connected components. This is shown to reduce the amount of graph-based computation required and to provide opportunities for parallelisation. In particular, we also focused on the applicability of this to incremental verification: re-analysing an MDP after small changes in its probability values, by re-using existing verification results. In the future, we plan to develop these techniques further, for example, considering also the case where the structure of the model changes.



(a) fast=0.1 (b) fast=0.01 (c) fast=0.001

Figure 1. Performance comparison for hybrid Tarjan computing expected reward (FireWire case study)

ACKNOWLEDGEMENTS

The authors are part supported by European Commission FP 7 project CONNECT (IST 231167), ERC Advanced Grant VERIWARE, DARPA/Air Force Research Laboratory contract FA8650-10-C-7077 (PRISMATIC) and EPSRC project PSS (EP/F001096/1).

REFERENCES

[1] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[2] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *Proc. FSTTCS'95*, ser. LNCS, vol. 1026. Springer, 1995.

[3] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. CAV'11*, ser. LNCS. Springer, 2011.

[4] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *Proc. ICSE'09*, 2009, pp. 100–110.

[5] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proc. ICSE'11*. ACM, 2011.

[6] F. Ciesinski, C. Baier, M. Größer, and J. Klein, "Reduction techniques for model checking Markov decision processes," in *Proc. QEST'08*. IEEE CS Press, 2008, pp. 45–54.

[7] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, pp. 146–160, 1972.

[8] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in log symbolic steps," in *Proc. FMCAD'00*, ser. LNCS, vol. 1954. Springer, 2000, pp. 37–54.

[9] R. Gentilini, C. Piazza, and A. Policriti, "Computing strongly connected components in a linear number of symbolic steps," in *Proc. SODA'03*, 2003, pp. 573–582.

[10] E. Ábrahám, N. Jansen, R. Wimmer, J. Katoen, and B. Becker, "DTMC model checking by SCC reduction," in *Proc. QEST'10*. IEEE CS Press, 2010, pp. 37–46.

[11] O. V. Sokolsky and S. A. Smolka, "Incremental model checking in the modal mu-calculus," in *CAV 94*, ser. LNCS, vol. 818. Springer, 1994, pp. 351–363.

[12] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *Proc. CAV'05*, ser. LNCS, vol. 3576. Springer, 2005, pp. 449–461.

[13] K. Heljanko, T. Junttila, and T. Latvala, "Incremental and complete bounded model checking for full PLTL," in *CAV 05*, ser. LNCS, vol. 3576. Springer, 2005, pp. 98–111.

[14] J. Kemeny, J. Snell, and A. Knapp, *Denumerable Markov Chains*, 2nd ed. Springer-Verlag, 1976.

[15] A. Pnueli, "The temporal logic of programs," in *Proc. FOCS'77*. IEEE Computer Society Press, 1977, pp. 46–57.

[16] C. Courcoubetis and M. Yannakakis, "The complexity of probabilistic verification," *J. of the ACM*, vol. 42, no. 4, 1995.

[17] L. de Alfaro, "Formal verification of probabilistic systems," Ph.D. dissertation, Stanford University, 1997.

[18] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[19] E. Nuutila and E. Soisalon-soininen, "On finding the strongly connected components in a directed graph," *Information Processing Letters*, vol. 49, pp. 9–14, 1994.

[20] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, "Performance analysis of probabilistic timed automata using digital clocks," *FMSD*, vol. 29, 2006.

[21] M. Kwiatkowska, G. Norman, and R. Segala, "Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM," in *Proc. CAV'01*, ser. LNCS, vol. 2102. Springer, 2001.

[22] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of Algorithms*, vol. 15, no. 1, 1990.

[23] M. Kwiatkowska, G. Norman, and J. Sproston, "Probabilistic model checking of the IEEE 802.11 wireless local area network protocol," in *Proc. PAPM/PROBMIV'02*, ser. LNCS, vol. 2399. Springer, 2002, pp. 169–187.

[24] http://www.prismmodelchecker.org/casestudies/.

[25] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[26] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proc. LICS'90*. IEEE Computer Society Press, 1990, pp. 428–439.

[27] M. Kwiatkowska, G. Norman, and J. Sproston, "Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol," *Formal Aspects of Computing*, vol. 14, no. 3, pp. 295–318, 2003.