Distributed Verification of Rare Properties using Importance Splitting Observers

Cyrille Jegourel, Axel Legay, Sean Sedwards, Louis-Marie Traonouez

Distributed Verification of Rare Properties using Importance Splitting Observers

Cyrille Jegourel, Axel Legay, Sean Sedwards and Louis-Marie Traonouez

INRIA Rennes – Bretagne Atlantique

Abstract. Rare properties remain a challenge for statistical model checking (SMC) due to the quadratic scaling of variance with rarity. We address this with a variance reduction framework based on lightweight importance splitting observers. These expose the model-property automaton to allow the construction of score functions for high performance algorithms. The confidence intervals defined for importance splitting make it appealing for SMC, but optimising its performance in the standard way makes distribution inefficient. We show how it is possible to achieve equivalently good results in less time by distributing simpler algorithms. We first explore the challenges posed by importance splitting and present an algorithm optimised for distribution. We then define a specific bounded time logic that is compiled into memory-efficient observers to monitor executions. Finally, we demonstrate our framework on a number of challenging case studies.

1 Introduction

Failure in critical systems is required to be very infrequent. Numerical model checking can quantify the probability of such failure with certainty, but is limited in its application to real systems because of the 'state explosion problem' [7]. This is addressed by statistical model checking (SMC) [27], which includes a number of approximative techniques based on Monte Carlo sampling [14]. Using SMC, only a subset of system states are generated on the fly during stochastic simulation, while results converge in a predictable way. Performance is typically independent of the size of the state space [24] and simulations may be efficiently divided on parallel computation architectures. SMC has therefore been successfully applied to real systems in a critical context, such as to the cabin communication system of an aeroplane [2]. Rare properties (those with probability close to zero) nevertheless pose a problem because the standard and relative estimation errors scale quadratically with rarity [14, 25]. For example, 4000 simulations would be sufficient to estimate a probability of $0.1 \pm 10\%$ with 95% confidence, whereas 4×10^{13} simulations would be necessary to estimate a probability of $10^{-6} \pm 10\%$ with the same confidence. Desirable failure rates in critical systems may be orders of magnitude lower, so we seek to enhance SMC with variance reduction techniques, such as importance sampling and importance splitting [22, 14, 25], without sacrificing the easy distribution that SMC affords.

Importance sampling weights the executable model of a system so that the rare property occurs more frequently in simulations. The proportion of simulations that satisfy the property using the weighted model overestimates the true probability, but the estimate may be exactly compensated by the weights. It is generally not feasible to implement a perfectly weighted executable model for importance sampling because (*i*) the perfect model may not actually exist as a re-parametrisation of the original model and (*ii*) a perfect re-parametrisation typically requires an iteration over all the transitions, defeating the benefits of sampling. Practical approaches tend to use a low dimensional vector of parameters to weight the model [18, 17]. Given such a parametrisation, importance sampling can be implemented with minimal memory and may be distributed efficiently on parallel computational architectures. The principal limitation of importance sampling is that without a guarantee that the simulation model is perfect, it is difficult to formally bound the error of estimates. In contrast, useful confidence intervals have been defined for importance *splitting* [6, 5].

Importance splitting divides a rare property into a set of less rare sub-properties that correspond to an increasing sequence of disjoint levels: the initial state corresponds to the lowest level, while states that satisfy the rare property corresponds to the final level. Importance splitting algorithms use a series of easy simulation experiments to estimate the conditional probabilities of going from one level to the next. Since relatively few simulations fail to satisfy the sub-properties, the overall simulation budget may be reduced. Each experiment comprises simulations initialised with the terminal states of previous simulations that reached the current level. The overall probability is the product of the estimates, with the best performance (lowest variance) achieved with many levels of equal conditional probability.

Importance splitting poses several challenges for optimisation and distribution. In the context of SMC, importance splitting algorithms repeatedly initialise simulations with states of the model-property product automaton. For arbitrary properties this may have size proportional to the length of a simulation trace. At the same time, increasing the number of levels to maximise performance reduces the number of simulation steps in each simulation experiment. The cost of sending the model-property state across slow communication channels may be significantly greater than the cost of short simulations. In addition, to specify levels with equal conditional probabilities it is necessary to define a 'score function' that maps the states of the product automaton to a value. This cannot easily be automated, so a syntactic description of the property automaton must be accessible for the user to construct a score function manually.

To address the above challenges we present an importance splitting framework for SMC, specifically considering the problems of distribution. We first discuss the problems of distributing importance splitting algorithms and present a fixed level algorithm optimised for distribution. We then define an expressive bounded time temporal logic and describe the system of efficient lightweight observers that implement it. These make the product automaton (*i*) accessible to the user, (*ii*) efficient to construct, (*iii*) efficient to distribute and (*iv*)

efficient to execute. Finally, we demonstrate the performance and flexibility of our framework on a number of case studies that are intractable to numerical methods.

We believe the present work is the first to describe a practical importance splitting framework for SMC and is therefore the first to consider the problems of distributing importance splitting for SMC.

Related Work

There have been many ad hoc implementations of importance splitting based on the original ideas of [21, 22]. The algorithm of [26] is a relatively recent example that is often cited. The work of [6, 5] is novel because the authors define efficient adaptive importance splitting algorithms that also include confidence intervals. To our knowledge, [19] is the first work to explicitly link importance splitting to arbitrary logical properties.

SMC tools construct an automaton (a monitor) to accept traces that satisfy a temporal logic formula, typically based on a time bounded variant of temporal logic. The proportion of independent simulations of a stochastic model that satisfy the property is then used to estimate the probability of the property or to test hypotheses about the probability. There have been several works that construct runtime verification monitors from temporal logic (e.g., [11, 13, 15, 10, 3]). Such monitors typically comprise tableau-based automata [12] whose states represent the combinations of subformulas of the overall property. While some have considered timed properties (e.g., [3]), the focus is predominantly unbounded LTL properties interpreted on finite paths [9]. In contrast, SMC typically checks formulas with explicit time bounds (see, e.g., (1)), which are inherently defined on finite traces. To avoid the combinatorial explosion of subformulas caused by including time in this way, the monitors used by [18, 4] and other high performance tools are compact "programs" that generate the states of an automaton on the fly and do not store them. Such programs incorporate notions of optimality that may be subtly different from those that apply in other contexts. Since states of the automaton are generated on the fly, it is not necessary for the automaton to have the minimum number of states. The actual requirements are that the automaton reaches a conclusion with the minimum number of input states and that its programmatic representation is as compact as possible. We adapt this "lightweight" approach to allow importance splitting for SMC to be efficiently distributed on high performance parallel computational architectures.

2 Technical Background

Our SMC tools (Plasma [18], Plasma-lab [4]) implement a bounded linear temporal logic having the following syntactic form:

$$\phi = \mathbf{X}^k\phi \mid \mathbf{F}^k\phi \mid \mathbf{G}^k\phi \mid \phi\mathbf{U}^k\phi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \alpha \qquad (1)$$

This syntax allows arbitrary combinations and nesting of temporal and atomic properties (i.e., those which may be evaluated in a single state and denoted by α). The time bound k may denote discrete steps or continuous time, but in this work we consider only discrete time semantics.

Given a finite trace ω, comprising sequence of states $\omega_0\omega_1\omega_2\cdots$, $\omega^{(i)}$ denotes the suffix $\omega_i\omega_{i+1}\omega_{i+2}\cdots$. The semantics of the satisfaction relation $\models$ is constructed inductively as follows:

$$\begin{aligned}
&\omega^{(i)} \models true \\
&\omega^{(i)} \models \alpha \iff \alpha \text{ is } true \text{ in state } \omega_i \\
&\omega^{(i)} \models \neg\varphi \iff \omega^{(i)} \models \varphi \not\models \\
&\omega^{(i)} \models \varphi_1 \vee \varphi_2 \iff \omega^{(i)} \models \varphi_1 \text{ or } \omega^{(i)} \models \varphi_2 \\
&\omega^{(i)} \models \mathbf{X}^k\varphi \iff \omega^{(k+i)} \models \varphi \\
&\omega^{(i)} \models \varphi_1 \mathbf{U}^k \varphi_2 \iff \exists j \in \{i,\ldots,i+k\} : \omega^{(j)} \models \varphi_2 \\
&\qquad \wedge (j = i \vee \forall l \in \{i,\ldots,j-1\} : \omega^{(l)} \models \varphi_1)
\end{aligned} \tag{2}$$

Other elements of the relation are constructed using the equivalences $false \equiv \neg true$, $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\mathbf{F}^k\varphi \equiv true\, \mathbf{U}^k\varphi$, $\mathbf{G}^k\varphi \equiv \neg(true\, \mathbf{U}^k\neg\varphi)$. Hence, given a property φ with syntax according to (1), $\omega \models \varphi$ is evaluated by $\omega^{(0)} \models \varphi$.

Importance Splitting and Score Functions

The neutron shield model of [21, 22] is illustrative of how importance splitting works. The distance travelled by a neutron in a shield defines a monotonic sequence of levels $0 = s_0 < s_1 < s_2 < \cdots < s_m = \textit{shield thickness}$, such that reaching a given level implies having reached all the lower levels. While the overall probability γ of passing through the shield is small, the probability of passing from one level to another can be made arbitrarily close to 1 by reducing the distance between levels. Denoting the abstract level of a neutron as s, the probability of a neutron reaching level s_i can be expressed as $\mathrm{P}(s \geq s_i) = \mathrm{P}(s \geq s_i \mid s \geq s_{i-1})\mathrm{P}(s \geq s_{i-1})$. Defining $\gamma = \mathrm{P}(s \geq s_m)$ and $\mathrm{P}(s \geq s_0) = 1$,

$$\gamma = \prod_{i=1}^{m} \mathrm{P}(s \geq s_i \mid s \geq s_{i-1}). \tag{3}$$

Each term of (3) is necessarily greater than or equal to γ, making their estimation easier. By writing $\gamma_i = \mathrm{P}(s \geq s_i \mid s \geq s_{i-1})$ and denoting the estimates of γ and γ_i as respectively $\hat{\gamma}$ and $\hat{\gamma}_i$, [19] defines the unbiased confidence interval

$$CI = \left[\hat{\gamma}/\left(1 + \frac{z_\alpha\sigma}{\sqrt{n}}\right), \hat{\gamma}/\left(1 - \frac{z_\alpha\sigma}{\sqrt{n}}\right)\right] \qquad \text{with} \qquad \sigma^2 \geq \sum_{i=1}^{m} \frac{1-\gamma_i}{\gamma_i}. \tag{4}$$

Confidence is specified via z_α, the $1 - \alpha/2$ quantile of the standard normal distribution, while n is the per-level simulation budget. We infer from (4) that

for a given γ the confidence is maximised by making both the number of levels m and the simulation budget large, with all γ_i equal.

The concept of levels can be generalised to arbitrary systems and properties in the context of SMC, treating s and s_i in (3) as values of a score function over the model-property product automaton. Intuitively, a score function discriminates good paths from bad, assigning higher scores to paths that more nearly satisfy the overall property. Since the choice of levels is crucial to the effectiveness of importance splitting, various ways to construct score functions from a temporal logic property are proposed in [19]. Formally, given a set of finite trace prefixes $\omega \in \Omega$, an ideal score function $S : \Omega \to \mathbb{R}$ has the characteristics $S(\omega) > S(\omega') \iff \mathrm{P}(\models \varphi \mid \omega) > \mathrm{P}(\models \varphi \mid \omega')$, where $\mathrm{P}(\models \varphi \mid \omega)$ is the probability of eventually satisfying φ given prefix ω. Intuitively, ω has a higher score than ω' iff there is more chance of satisfying φ by continuing ω than by continuing ω'. The minimum requirement of a score function is $S(\omega) \geq s_\varphi \iff \omega \models \varphi$, where s_φ is an arbitrary value denoting that φ is satisfied. Any trace that satisfies φ must have a score of at least s_φ and any trace that does not satisfy φ must have a score less than s_φ. In what follows we assume that (3) refers to scores.

3 Distributing Importance Splitting

Simple Monte Carlo SMC may be efficiently distributed because once initialised, simulations are executed independently and the result is communicated at the end with just a single bit of information (i.e., whether the property was satisfied or not). By contrast, the simulations of importance splitting are dependent because scores generated during the course of each simulation must be processed centrally. The amount of central processing can be minimised by reducing the number of levels, but this generally reduces the variance reduction performance.

Alternatively, entire instances of the importance splitting algorithm may be distributed and their estimates averaged, with each instance using a proportionally reduced simulation budget. We use this approach to generate some of the results in Section 6, but note that if the budget is reduced too far, the algorithm will fail to pass from one level to the next (because no trace achieves a high enough score) and no valid estimate will be produced.

Distribution of importance splitting is thus possible, but its efficiency is dependent on the particular problem. In this work we therefore provide the framework to explore different approaches. In Section 3.1 we first describe the concept of an adaptive importance splitting algorithm and then explain why this otherwise optimised technique is unsuitable for distribution. In Section 3.2 we motivate the use of a fixed level algorithm for "lightweight" distribution and provide a suitable algorithm. The results we present in Section 6 demonstrate that this simpler approach can be highly effective.

3.1 The Adaptive Algorithm

The basic notion of importance splitting described in Section 2 can be directly implemented in a so-called fixed level algorithm, i.e., an algorithm in which the

levels are pre-defined by the user. With no a priori information, such levels will typically be chosen to subdivide the maximum score equally. In general, however, this will not equally divide the conditional probabilities of the levels, as required by (4) to minimise variance. In the worst case, one or more of the conditional probabilities will be too low for the algorithm to pass between levels. Finding good or even reasonable levels by trial and error may be computationally expensive and has prompted the development of adaptive algorithms that discover optimal levels on the fly [6, 19, 20]. Instead of pre-defining levels, the user specifies the proportion of simulations to retain after each iteration. This proportion generally defines all but the final conditional probability in (3).

The adaptive importance splitting algorithm first performs a number of simulations until the overall property is decided, storing the resulting traces of the model-property automaton. Each trace induces a sequence of scores and a corresponding maximum score. The algorithm finds a level that is less than or equal to the maximum score of the desired proportion of simulations to retain. The simulations whose maximum score is below this current level are discarded. New simulations to replace the discarded ones are initialised with states corresponding to the current level, chosen at random from the retained simulations. The new simulations are continued until the overall property is decided and the procedure is repeated until a sufficient proportion of simulations satisfy the overall property.

The principal advantage of the adaptive algorithm is that by simply rejecting the minimum number of simulations at each level it is possible to maximise confidence for a given score function. The principal disadvantage is that it stores simulation traces, severely limiting the size of model and simulation budget. The use of lightweight computational threads is effectively prohibited. Moreover, minimising the number of rejected simulations reduces the number of simulations performed between levels, thus reducing the possibility to perform computations in parallel. Minimising the rejected simulations also maximises the number of levels, which in turn minimises the number of simulation steps between each level. This further limits the feasibility of dividing the algorithm, since sending a model-property state over a slow communication channel may be orders of magnitude more costly than performing a short simulation locally.

3.2 A Fixed Level Algorithm for Distribution

In contrast to the adaptive algorithm, the fixed level importance splitting algorithm does not need to store traces, making it lightweight and suitable for distribution. Scores are calculated on the fly and only the states that achieve the desired level are retained for further consideration. While the choice of levels remains a problem, an effective strategy is to first use the adaptive algorithm with a relatively high rejection rate to find good fixed levels. An estimate with high confidence can then be generated efficiently by distributing the fixed level algorithm.

Algorithm 1 is our fixed level importance splitting algorithm optimised for distribution. We use the terms server and client to refer to the root and leaf nodes

of a network of computational devices or to mean independent computational threads on the same machine. In essence, the server manages the job and the clients perform the simulations.

The server initially sends compact representations of the model and property to each client. Thereafter, only the state of the product automaton is communicated. In general, each client returns terminal states of simulations that reached the current level and the server distributes these as initial states for the next round of simulations. Algorithm 1 optimises this. The server requests and distributes only the number of states necessary to restart the simulations that failed to reach the current level, while maintaining the randomness of the selection. Despite this optimisation, however, the performance of this and other importance splitting algorithms will be confounded by the combination of large state size and properties having short time bounds. Under such circumstances it may be preferable to distribute entire instances of the algorithm, as described above.

The memory requirements of Algorithm 1 are minimal. Each client need only store the state of n simulations. As such, it is conceivable to distribute simulations on lightweight computational threads, such as those provided by GPGPU (general purpose computing on graphics processing units).

Algorithm 1: Distributed Fixed Level Importance Splitting

input: $s_1 < s_2 < \cdots < s_m$ is a sequence of scores, with $s_m = s_\varphi$ the score necessary to satisfy property φ

$\hat{\gamma} \leftarrow 1$ is the initial estimate of $\gamma = \mathrm{P}(\omega \models \varphi)$
server sends compact description of model and observer to k clients
each client initialises n simulations
for $s \leftarrow s_1, \ldots, s_m$ **do**
    each client continues its n simulations from their current state simulations halt as soon as their scores reach s
    $\forall$ clients, client i sends server the number of traces n_i that reached s
    server calculates $\hat{\gamma} \leftarrow \hat{\gamma} n'/kn$, where $n' = \sum n_i$
    for $j \leftarrow 1, \ldots, kn - n'$ **do**
        server chooses client i at random, with probability n_i/n'
        client i sends server a state chosen uniformly at random from those that reached s
        server sends state to client corresponding to failed simulation j, as initial state of new simulation to replace simulation j

output: $\hat{\gamma}$

4 Linear Temporal Logic for Importance Splitting

High performance SMC tools, such as [18, 4], avoid the complexity of standard model checking by compiling the property to a program of size proportional to the formula and memory proportional to the maximum sum of nested time

bounds. This program implicitly encodes the model checking automaton, but is exponentially smaller. For example, the property $\mathbf{X}^k\varphi$ can be implemented as a loop that generates k simulation steps before returning the truth of φ in the last state; the property $\vartheta\,\mathbf{U}^k\varphi$ can be implemented as a loop that generates up to k simulation steps while ϑ is true and φ is not true, returning the value of φ in the last state otherwise. If ϑ and φ are atomic, the programs require just $\mathcal{O}(\log k)$ bits of memory to hold a loop counter.

In contrast, the nested property $\mathbf{F}^{k1}(\vartheta \vee \mathbf{G}^{k2}\varphi)$ has an $\mathcal{O}(k2)$ memory requirement. If ϑ is not true on step $i < k1$ it may be necessary to simulate up to step $i + k2$ to decide subformula $\mathbf{G}^{k2}\varphi$. If $\vartheta \vee \mathbf{G}^{k2}\varphi$ turns out to be false on step i, it will then be necessary to consider the truth of ϑ on step $i + 1$, noting that the last simulated step could be $i + k2$. To evaluate this formula it is effectively necessary to remember the truth of ϑ on $\mathcal{O}(k2)$ simulation steps. Similar requirements can arise when the until operator ($\mathbf{U}$) is a subformula of a temporal operator. In all such cases the sequence of stored truth values become part of the state of the property automaton.

SMC using importance splitting requires that simulations are repeatedly and frequently initialised with the state of the model-property product automaton. If the size of this state is proportional to the time bounds of temporal operators, initialisation may have comparable complexity to simulation. This becomes especially problematic if the state is to be transmitted across relatively slow communication channels for the purposes of distribution. We therefore define a subset of (1), the size of whose automata is not dependent on the bounds of temporal operators:

$$\begin{aligned} \phi =& \mathbf{X}^k\phi \mid \psi\mathbf{U}^k\psi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \psi \\ \psi =& \mathbf{X}^k\psi \mid \mathbf{F}^k\psi \mid \mathbf{G}^k\psi \mid \alpha \end{aligned} \tag{5}$$

The semantics of (5) is the same as (1), but (5) restricts how temporal operators may be combined. In particular, $\mathbf{U}$ may not be the subformula of a temporal operator other than $\mathbf{X}$ and temporal operators that are subformulas of other temporal operators may not be combined with Boolean connectives. Temporal operators containing other temporal operators as subformulas may, however, be combined. This logic expresses many useful properties, including nested bounded temporal properties that are not implemented in the numerical model checker PRISM[1].

5 Lightweight Observers for Importance Splitting

To facilitate the construction of score functions we implement the logic given by (5) as a set of nested observers. Each observer corresponds to either a temporal operator, a Boolean operator acting on temporal operators, or as a predicate describing an atomic property. In our implementation observers are written in a syntax based on the commonly used reactive modules language [1], using the

[1] www.prismmodelchecker.org

notion of 'guarded commands' [8] with sequential semantics. The observers are easily implemented in other modelling languages.

An observer comprises a set of guarded commands, any number of which may be enabled and executed on a given simulation step. Updates are performed in syntactic order after all guards have been evaluated, hence the update of one command does not affect the guards of commands in the same observer. In general, the output of one observer is the input to another and observers are therefore executed in reverse order of their nesting.

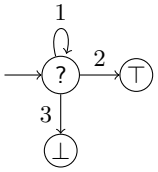
Observers evaluate states as they are generated by the simulation. Since it may not be possible to decide a property before seeing a certain number of states, observers implement a three valued logic. In Figs. 1, 2 and 3 we use the symbols ?, $\top$ and $\bot$ to denote the three values *undecided*, *true* and *false*, respectively. The state of an observer changes only when at least one of its inputs is decided. An observer may reach a deadlock state (no commands enabled) once its output is decided and cannot be changed by further input. A simulation terminates when the output of the root observer is decided, i.e., the property is decided. Simulations may also be paused by the importance splitting algorithm if the score reaches a desired level.

Observers implementing the same temporal operator behave differently according to their level of nesting within a formula. We therefore distinguish *outer* and *inner* temporal observers. The temporal operators closest to the root of any branch of the syntax tree induced by a formula are implemented by outer observers. Their output proceeds from *undecided* to either *true* or *false* and then does not change. Inner observers encode temporal operators that are the subformulas of other temporal operators. Their output proceeds from *undecided* to a possibly alternating sequence of *true*, *false* and *undecided* values because their enclosing operator(s) cause them to evaluate a moving window of states in the execution trace. The inner and outer variants of **X**, **F** and **G** are closely related—outer observers are essentially simplified inner observers. When **U** is a subformula of **X**, however, the **X** is implemented as a delay within the **U** observer.

In what follows we describe the important aspects of the various observers that implement (5). The accompanying figures include diagrammatic representations of how the observers work and sets of commands written in the form *predicate* : *update*. Each observer has Boolean output variables o and d to indicate respectively the result and whether the property has been decided (observers for atomic formulas omit d). Observers for temporal operators take discrete time bound k as a parameter and use a counter variable w (**U** uses counter variables w' and w''). Inner temporal operators make use of an additional counter, t (**U** uses t' and t''). The inputs of observers are Boolean variables o' and o'', with corresponding decidedness d' and d''.

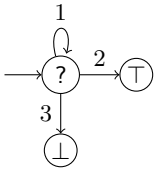
Connective Observers These observers implement Boolean connectives at syntactic level ϕ in (5) and take advantage of the equivalences $\mathit{false} \wedge ? = \mathit{false}$, $\mathit{true} \vee ? = \mathit{true}$, $\mathit{false} \Rightarrow ? = \mathit{false}$ and $? \Rightarrow \mathit{true} = \mathit{true}$, for any truth value of ?.

Figure 1a describes the observer for conjunction and Fig. 1b describes the observer for implication. The observer for disjunction may be derived from that of conjunction by negating all instances of o' and o'', and by exchanging $o \leftarrow \mathit{true}$ and $o \leftarrow \mathit{false}$. Negation is implemented by inverting the truth assignment of the observer to which it applies, i.e., by exchanging $o \leftarrow \mathit{true}$ and $o \leftarrow \mathit{false}$. The connectives may be combined with themselves and with outer temporal operators. Boolean connectives that apply only to atomic properties (i.e., syntactic level α) are implemented directly in formulas within observers for atomic properties.



1. $\neg d \wedge (\neg d' \vee \neg d'') \wedge \neg(\neg o' \wedge d' \vee \neg o'' \wedge d'')$
2. $\neg d \wedge d' \wedge o' \wedge d'' \wedge o'' : d \leftarrow \mathit{true}, o \leftarrow \mathit{true}$
3. $\neg d \wedge (\neg o' \wedge d' \vee \neg o'' \wedge d'') : d \leftarrow \mathit{true}, o \leftarrow \mathit{false}$

(a) $o \leftarrow o' \wedge o''$



1. $\neg d \wedge (\neg d' \wedge \neg(d'' \wedge o'') \vee d' \wedge o' \wedge \neg d'')$
2. $\neg d \wedge (\neg o' \wedge d' \vee o'' \wedge d'') : d \leftarrow \mathit{true}, o \leftarrow \mathit{true}$
3. $\neg d \wedge d' \wedge o' \wedge d'' \wedge \neg o'' : d \leftarrow \mathit{true}, o \leftarrow \mathit{false}$

(b) $o \leftarrow o' \Rightarrow o''$

Fig. 1: Connective observers. Initially $d = \mathit{false}$.

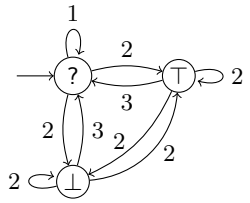
Inner Temporal Observers These observers act on a moving window of states created by an enclosing temporal operator. The output may pass from one decided value to the other and also become undecided.

Figure 2a describes the observer for $\mathbf{X}^k$. Command 1 counts decided input states until bound k is reached. Thereafter command 2 sets the output decided and equal to the value of the input.

Figure 2b describes the observer for $\mathbf{F}^k$. While decided inputs are not *true*, command 1 increments w from 0 to k. If at any time the input is *true*, command 2 sets the output to *true* and the "true-counter" t is set to w. Command 5 decrements t on subsequent false inputs. The output remains true while $t > 0$. If w reaches k while $t = 0$, command 3 sets the output to *false*.

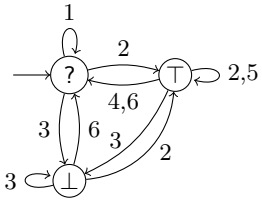
The observer for $\mathbf{G}^k$ may be derived from that of $\mathbf{F}^k$ by negating all instances of o' and $\neg o'$, and by exchanging $o \leftarrow \mathit{true}$ and $o \leftarrow \mathit{false}$.

Outer Temporal observers The outer observers for $\mathbf{X}^k$ and $\mathbf{F}^k$ are not illustrated but may be derived from their respective inner observers given in Fig. 2.



1. $\neg d \wedge d' \wedge w < k : w \leftarrow w + 1$
2. $d' \wedge w = k : d \leftarrow \mathit{true}, o \leftarrow o'$
3. $d \wedge \neg d' : d \leftarrow \mathit{false}$

(a) $o \leftarrow \mathbf{X}^k o'$



1. $\neg d \wedge d' \wedge \neg o' \wedge w < k : w \leftarrow w + 1$
2. $d' \wedge o' : o \leftarrow \mathit{true}, d \leftarrow \mathit{true}, t \leftarrow w$
3. $d' \wedge \neg o' \wedge t = 0 \wedge w = k : d \leftarrow \mathit{true}, o \leftarrow \mathit{false}$
4. $d \wedge d' \wedge \neg o' \wedge t = 0 \wedge w < k : d \leftarrow \mathit{false}$
5. $d \wedge d' \wedge \neg o' \wedge t > 0 : t \leftarrow t - 1$
6. $d \wedge \neg d' : d \leftarrow \mathit{false}$

(b) $o \leftarrow \mathbf{F}^k o'$

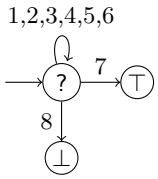
Fig. 2: Observers for inner temporal operators. Initially $w = t = 0, d = \mathit{false}$.

For $\mathbf{X}^k$, command 3 is removed and the guard of command 2 is strengthened with $\neg d$. For $\mathbf{F}^k$, commands 4, 5 and 6, together with all references to counter t, are removed, while the guards of commands 2 and 3 are strengthened by $\neg d$. The outer observer for $\mathbf{G}^k$ can be derived from that of $\mathbf{F}^k$ in the same way as described for inner temporal observers.

Figure 3 describes the observer for properties of the form $\mathbf{X}^{k_\mathbf{X}}(\vartheta\, \mathbf{U}^k \varphi)$, which can be simplified to implement properties of the form $\vartheta\, \mathbf{U}^k \varphi$. Since ϑ and φ may be temporal formulas that are satisfied on different simulation steps in arbitrary order, the observer employs variables w' and w'' to respectively count the sequences of $\neg\varphi$ and ϑ (commands 3 and 5). Variable t' then records the position of the first φ (command 4), while t'' records the position of the last ϑ (command 5). Using t' and t'', commands 7 and 8 are able to determine if the property is satisfied or falsified, respectively. The $\mathbf{X}^{k_\mathbf{X}}$ part of the formula is implemented by initialising variables w' and w'' to $-k_\mathbf{X}$, forcing the observer to ignore the first $k_\mathbf{X}$ decided values of ϑ and φ. In the case of properties of the form $\vartheta\, \mathbf{U}^k \varphi$, w' and w'' are initialised to 0 and the automaton may be simplified by removing commands 1 and 2 and all instances of expressions $w' \geq 0$ and $w'' \geq 0$.

6 Case Studies

We have implemented our importance splitting framework in Plasma-lab [4] and demonstrate its use on three case studies whose state space is intractable to numerical model checking. The following results do not seek to promote a particular methodology (adaptive or fixed level algorithm, distributed or single



1. $d' \wedge w' < 0 : w' \leftarrow w' + 1$
2. $d'' \wedge w'' < 0 : w'' \leftarrow w'' + 1$
3. $\neg d \wedge d' \wedge \neg o' \wedge w' \geq 0 \wedge w' \leq k : w' \leftarrow w' + 1$
4. $\neg d \wedge d' \wedge o' \wedge w' \geq 0 \wedge w' \leq k : t' \leftarrow w', w' \leftarrow k + 1$
5. $\neg d \wedge d'' \wedge o'' \wedge w'' \geq 0 \wedge w'' < k : w'' \leftarrow w'' + 1, t'' \leftarrow w''$
6. $\neg d \wedge d'' \wedge \neg o'' \wedge w'' \geq 0 \wedge w'' < k : w'' \leftarrow k$
7. $\neg d \wedge t' \geq 0 \wedge t'' \geq t' - 1 : d \leftarrow \mathit{true}, o \leftarrow \mathit{true}$
8. $\neg d \wedge (t' < 0 \wedge w' = k + 1 \vee w'' = k \wedge (t'' < t' - 1 \vee t' < 0 \wedge t'' \leq w' - 1)) : d \leftarrow \mathit{true}, o \leftarrow \mathit{false}$

Fig. 3: Observer for $o \leftarrow \mathbf{X}^{k_\mathbf{X}}(o''\mathbf{U}^k o')$. Initially $t'' = 0, t' = -1, d = d_\mathbf{X} = \mathit{false}$ and $w' = w'' = -k_\mathbf{X}$ (see text).

machine), but serve to illustrate the flexibility of our platform. The software, models and observers can be downloaded from our website[2]. The leader election and dining philosophers models are also illustrated on the PRISM case studies website[3].

For each model we performed a number of experiments to compare the performance of the fixed and adaptive importance splitting algorithms with and without distribution, using different simulation budgets and levels. Our results are illustrated in the form of empirical cumulative probability distributions of 100 estimates, noting that a perfect (zero variance) estimator distribution would be represented by a single step. The results are also summarised in Table 1. The probabilities we estimate are all close to 10^{-6} and are marked on the figures with a vertical line. Since we are not able to use numerical techniques to calculate the true probabilities, we use the average of 200 low variance estimates as our best overall estimate.

As a reference, we applied the adaptive algorithm to each model using a single computational thread. We chose parameters to maximise the number of levels and thus minimise the variance for a given score function and budget. The resulting distributions, sampled at every tenth percentile, are plotted with circular markers in the figures. Over these points we superimpose the results of applying a single instance of the fixed level algorithm with just a few levels. We also superimpose the average estimates of five parallel threads running the fixed level algorithm, using the same levels.

The figures confirm our expectation that the fixed level algorithm with few levels is outperformed by the adaptive algorithm. The figures also demonstrate that the average of parallel instances of the fixed level algorithm are very close to the performance of the adaptive algorithm. The timings given in Table 1 show that the distributed approach achieves these results in less time. For comparison we also include the estimated time of using a simple Monte Carlo (MC) estimator to achieve the same standard deviation. Importance splitting gives more than

[2] projects.inria.fr/plasma-lab/importance-splitting
[3] www.prismmodelchecker.org/casestudies

three orders of magnitude improvement in all cases. All results were generated using an Intel Core i7-3740 CPU with 4 cores running at 2.7 GHz.

In the remainder of this section we briefly describe our models and their associated properties and score functions.

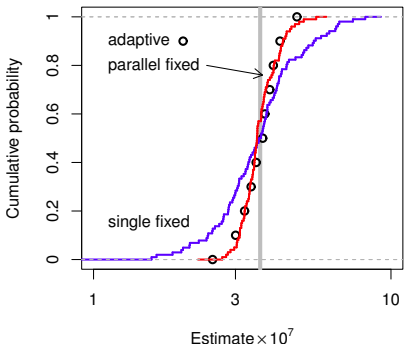


Fig. 4: Leader election.

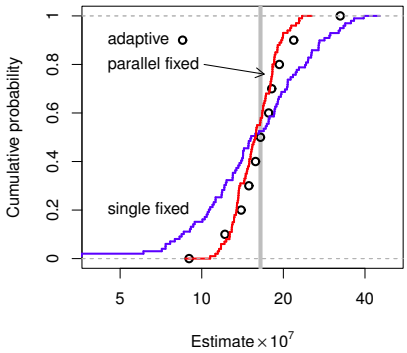


Fig. 5: Dining philosophers.

Leader Election Our leader election case study is based on the PRISM model of the synchronous leader election protocol of [16]. With $N = 20$ processes and $K = 6$ probabilistic choices the model has approximately 1.2×10^{18} states. We consider the probability of the property $\mathbf{G}^{420}\neg \mathit{elected}$, where *elected* denotes the state where a leader has been elected. Our chosen score function uses the time bound of the $\mathbf{G}$ operator to give nominal scores between 0 and 420. The model constrains these to only 20 actual levels (some scores are equivalent with respect to the model and property), but with evenly distributed probability. For the fixed level algorithm we use scores of $70, 140, 210, 280, 350$ and 420.

Dining Philosophers Our dining philosophers case study extends the PRISM model of the fair probabilistic protocol of [23]. With 150 philosophers our model contains approximately 2.3×10^{144} states. We consider the probability of the property $\mathbf{F}^{30}\mathit{Phil\ eats}$, where *Phil* is the name of an arbitrary philosopher. The adaptive algorithm uses the heuristic score function described in [20], which includes the five logical levels used by the fixed level algorithm. Between these levels the heuristic favours short paths, based on the assumption that as time runs out the property is less likely to be satisfied.

Dependent Counters Our dependent counters case study comprises ten counters, initially set to zero, that with some probability dependent on the values of the other counters are either incremented or reset to zero. This can be viewed as

modelling an abstract computational process, a set of reservoirs of finite capacity, or as the failure and repair of ten different types of components in a system, etc. With a maximum count of 10, the model has approximately 2.6×10^{10} states.

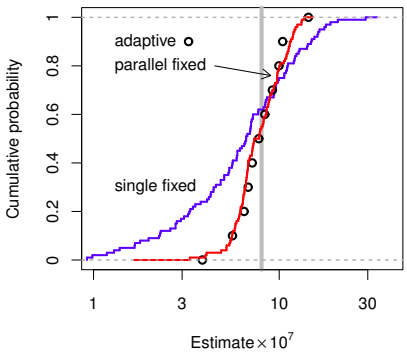


Fig. 6: Dependent counters.

		Adaptive	Single	Parallel
Leader	Std. dev.	4.8×10^{-8}	1.3×10^{-7}	5.2×10^{-8}
	Levels	20	6	6
	Budget	1000	1000	5×1000
	Time (MC)	7.3s (30h)	2.5s (4.4h)	5.8s (5.0h)
Philosophers	Std. dev.	4.2×10^{-7}	7.7×10^{-7}	2.8×10^{-7}
	Levels	109	5	5
	Budget	1000	1000	5×1000
	Time (MC)	5.4s (2.3h)	1.7s (41m)	3.7s (1.4h)
Counters	Std. dev.	2.1×10^{-7}	5.0×10^{-7}	2.3×10^{-7}
	Levels	3942	4	4
	Budget	500	500	5×500
	Time (MC)	15s (7.5h)	2.8s (1.2h)	4.8s (1.9h)

Table 1: Summary of results.

We consider the probability of the property $\mathbf{X}^1(\neg \mathit{init}\, \mathbf{U}^{1000}\, \mathit{complete})$, where *init* and *complete* denote the initial state and the state where all counters have reached their maximum value. Our score function ranges over values between 0 and 99, but the probabilities are not evenly distributed. With a budget of 500, uniformly distributed fixed scores fail to produce traces that satisfy the property until the difference between the last two levels is about 5. Note that our budget is limited to only 500 simulations due to the length of the traces that must be stored by the adaptive algorithm. We maintain this budget for the fixed level algorithm to simplify comparison. After a small amount of trial and error, we adopted fixed scores of $80, 90, 95$ and 99.

7 Challenges and Prospects

Our results demonstrate the effectiveness and flexibility of our framework with discrete time properties applied to standard case studies. Future challenges include industrial scale examples and the implementation of continuous time properties. We also intend to provide proofs of the correctness of our observers and of our logic's memory requirements.

Although the manual construction of score functions adds to the overall cost of using importance splitting, we believe that distribution relaxes the need for these to be highly optimised. We nevertheless expect that it will be possible to construct good score functions automatically using statistical learning techniques.

Acknowledgement

This work was partially supported by the European Union Seventh Framework Programme under grant agreement number 318490 (SENSATION).

References

1. Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
2. Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model checking of large heterogeneous systems. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, volume 6117 of *LNCS*, pages 32–46. Springer, 2010.
3. Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.
4. Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. PLASMA-lab: A flexible, distributable statistical model checking library. In Kaustubh Joshi, Markus Siegle, Marille Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems*, volume 8054 of *LNCS*, pages 160–164. Springer, 2013.
5. F. Cérou, P. Del Moral, T. Furon, and A. Guyader. Sequential Monte Carlo for rare event estimation. *Statistics and Computing*, 22:795–808, 2012.
6. Frédéric Cérou and Arnaud Guyader. Adaptive multilevel splitting for rare event analysis. *Stochastic Analysis and Applications*, 25:417–443, 2007.
7. E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009.
8. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.
9. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Computer Aided Verification*, pages 27–39. Springer, 2003.
10. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.
11. M. C. W. Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes in Theoretical Computer Science*, 55(2):181–199, 2001.
12. Rob Gerth, Doron Peled, Moshe Y. Vardi Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.
13. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of 16th Annual International Conference on Automated Software Engineering*, pages 412–416. IEEE, Nov 2001.
14. J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Methuen & Co., 1964.
15. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
16. Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

17. Cyrille Jegourel, Axel Legay, and Sean Sedwards. Cross-entropy optimisation of importance sampling parameters for statistical model checking. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *LNCS*, pages 327–342. Springer, 2012.
18. Cyrille Jegourel, Axel Legay, and Sean Sedwards. A platform for high performance statistical model checking – PLASMA. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 498–503. Springer, 2012.
19. Cyrille Jegourel, Axel Legay, and Sean Sedwards. Importance splitting for statistical model checking rare properties. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 576–591. Springer, 2013.
20. Cyrille Jegourel, Axel Legay, and Sean Sedwards. An effective heuristic for adaptive importance splitting in statistical model checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *LNCS*, pages 143–159. Springer, 2014.
21. Herman Kahn. Random sampling (Monte Carlo) techniques in neutron attenuation problems. *Nucleonics*, 6(5):27, 1950.
22. Herman Kahn and T. E. Harris. Estimation of particle transmission by random sampling. In *Applied Mathematics*, volume 5 of *series 12*. National Bureau of Standards, 1951.
23. Daniel Lehmann and Michael O. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proc. 8^{th} Ann. Symposium on Principles of Programming Languages*, pages 133–138, 1981.
24. H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.
25. Gerardo Rubino and Bruno Tufin (eds.). *Rare Event Simulation using Monte Carlo Methods*. John Wiley & Sons, Ltd, 2009.
26. Manuel Villén-Altamirano and José Villén-Altamirano. RESTART: A method for accelerating rare event simulations. In J. W. Cohen and C. D. Pack, editors, *Queueing, Performance and Control in ATM*, pages 71–76. Elsevier, 1991.
27. HåkanL. S. Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, 2006.