Simulación de eventos raros con Importance Splitting, extendiendo FIG con Fixed Effort y Fixed Success.

Trabajo final realizado para la Facultad de Matemática, Astronomía, Física y Computación (FaMAF), de la Universidad Nacional de Córdoba, en cumplimiento parcial para la obtención del grado de Licenciado en Ciencias de la Computación.

Matías José Hunicken Berardo

Directores: Pedro R. D'Argenio y Carlos E. Budde

3 de diciembre de 2018

Esta obra está bajo una licencia Creative Commons "Reconocimiento-NoCommercial-SinObraDerivada 4.0 Internacional".

Resumen

Dadas las potenciales implicaciones trágicas de las fallas en sistemas críticos, es necesario utilizar métodos que permitan reducir lo más posible las posibilidades de errores. En muchos casos, los sistemas que nos interesa verificar tienen comportamiento cuantificable probabilísticamente. Existen técnicas analíticas o numéricas para analizar ciertos modelos probabilistas. Sin embargo estas técnicas no son aplicables a la generalidad de los modelos, por ejemplo cuando estos usan distribuciones probabilistas continuas arbitrarias. En estos casos se recurre a la simulación (model checking estadístico). Un problema que surge al hacer simulación es cuando el evento a analizar exhibe una probabilidad muy baja. Esto es un caso bastante común en la verificación de sistemas, porque es posible que no se pueda garantizar que el sistema no falle nunca, pero se busque garantizar que esta probabilidad sea aceptablemente baja. El problema en estos casos es que resulta en un esfuerzo computacional muy grande y por consiguiente requiere de técnicas específicas para lidiar con este problema. En este trabajo nos enfocamos en la técnica de Importance splitting (división por importancia) para la simulación de eventos raros. La contribución del trabajo consiste en la implementación de las técnicas de splitting fixed-effort y fixed-success en la herramienta FIG, las cuales presumiblemente se comportan generalmente mejor que la técnica RESTART (ya implementada en FIG) para la estimación de propiedades transitorias. Además, comparamos el rendimiento de las tres técnicas para varios casos de estudio con el fin de validar la hipótesis.

Abstract

Given the potentially tragic implications of failures in critical systems, it is necessary to use methods that allow to keep the posibilities of errors as low as possible. In many cases, the systems we want to verify exhibit probabilistic behavior. There are analytic and numeric techniques for analysing certain probabilistic models. However, this techniques can not be applied in general to all types of models, for example, when they use arbitrary probabilistic distributions. In these cases, we turn to simulation (statistical model checking). One problem that arises when doing simulation is when the event that we want to analyze has very low probability of occurrence. This is a common case in the verification of systems, since it is possible that we cannot guarantee that the system will not fail, but we want to guarantee that this probability is acceptably low. The problem in these cases is that it results in a huge computational effort, and therefore, it requires specific techniques to deal with this problem. In this work, we focus on the technique of Importance Splitting for rare event simulation. The contribution of this work consists in the implementation of the splitting techniques fixed-effort and fixed-success in the tool FIG. These techiniques behave presumably better than the technique RESTART (already implemented in FIG) for estimation of transient properties. In addition, we compare the performance of the three techniques on various case studies with the goal of validating this hypothesis.

Índice general

Capítulo 1

Introducción

Los sistemas automáticos, en particular los sistemas informáticos, tienen un papel cada vez más relevante en la vida de las personas. El rápido desarrollo tecnológico durante el pasado siglo ha causado un gran impacto en la forma en que nos comunicamos, trabajamos, nos transportamos, etc.

Dada la creciente complejidad de estos sistemas, es prácticamente inevitable que fallen por algún motivo, ya sea por errores humanos durante su diseño o su implementación, o por deterioro con el paso del tiempo. En la mayoría de las ocasiones, tales errores no tienen consecuencias negativas muy relevantes, sino que resultan en una simple molestia. Pero hay errores que pueden tener consecuencias realmente catastróficas, cuando ocurren en sistemas críticos, es decir, aquellos en los que una falla puede resultar en grandes pérdidas económicas, o amenazas a la salud y la vida de las personas.

Tenemos por ejemplo el caso de Therac-25 [20], una máquina de radioterapia que por errores en la concurrencia podía exponer a los pacientes a cientos de veces más radiación de la esperada. Ocurrieron al menos 6 accidentes entre 1985 y 1987, que resultaron en muerte o lesión grave para los pacientes en cuestión. Otro caso notable es el del Mars Climate Orbiter [23], una nave espacial no tripulada enviada por la NASA en 1998 para estudiar el clima marciano, que se desintegró al ingresar a la atmósfera de Marte por un error en el cálculo del recorrido, a causa de que la nave y el sistema de control usaban distintas unidades de medida. Un ejemplo más reciente es Heartbleed [10], un error en la librería OpenSSL descubierto en 2014, que permitía acceder a información privada a causa de un *buffer over-read*.

Dadas las potenciales implicaciones trágicas en tal tipo de sistemas, es necesario utilizar métodos que permitan reducir lo más posible las posibilidades de fallas. En varios de los ejemplos mencionados, la falla ocurrió a pesar de que el proceso de desarrollo del sistema en cuestión estuvo sujeto a rigurosos métodos de revisión y control, ya que los errores humanos ocurren también en tales instancias. Se puede concluir que se necesita algo más que la revisión humana, y en este contexto surge la necesidad de técnicas formales para la verificación de sistemas críticos.

Para aplicar verificación formal, el sistema se suele transformar a una descripción formal que representa su funcionamiento, llamada modelo. Se conoce como *Model checking* [4] al conjunto de técnicas que permiten verificar automáticamente propiedades sobre este tipo de modelos. Los modelos suelen ser expre-

sados como sistemas de transiciones, es decir, un conjunto de estados en los que se encuentra el sistema y el conjunto de transiciones que pueden ocurrir. Una forma típica de *Model checking* es verificar que cierta propiedad se satisface para cualquier sucesión de transiciones posible desde un estado inicial.

En muchos casos, los sistemas que nos interesa verificar tienen comportamiento probabilístico. Por lo general, esto ocurre en casos donde parte de su funcionamiento depende de agentes externos al sistema que no controlamos, pero podemos describir su acción mediante un proceso estocástico. Por ejemplo, podría tratarse de un servidor que recibe consultas de clientes externos al sistema, y cada pedido tiene un tiempo de respuesta que depende del tipo de consulta recibida, asumiendo que tenemos conocimiento de las ditribuciones del tiempo entre consultas y del tipo de consulta. En este tipo de modelos, las propiedades que nos interesa conocer son probabilísticas, por ejemplo, "la probabilidad de que el sistema falle", "tiempo promedio de respuesta", "proporción de tiempo en que el sistema se encuentra en un estado inactivo", y se busca que dichos valores estén en un rango aceptable. La verificación de este tipo de modelos se conoce como *Model checking* probabilístico o estocástico [16].

Usualmente, las propiedades se verifican mediante la exploración del espacio del estados. Esto implica un problema, dado que la mayoría de modelos que se quieren verificar constan de muchas componentes que se ejecutan concurrentemente, y por lo tanto la cantidad de estados del sistema completo puede ser tanto como el producto de las cantidades de estados de cada una de las componentes. Esto puede resultar en una cantidad inmanejable de estados en modelos que constan de muchas componentes. Esta situación se conoce como "explosión de estados", y aunque se han desarrollado técnicas que alivian este problema [28, 24], sigue siendo una limitación importante a la hora de verificar modelos.

En el caso de *Model checking* probabilístico, se da también que los modelos que se pueden verificar de forma analítica tienen ciertas restricciones [4], que no siempre se cumplen en los sistemas que queremos modelar. De más está decir que para que tenga sentido hacer *Model checking*, nuestro modelo debe representar de la forma más exacta posible los aspectos de nuestro sistema que nos interesa verificar.

En situaciones donde se da alguno de los dos problemas mencionados, una alternativa es hacer un análisis estadístico del modelo. Es decir, muestrear ejecuciones del modelo aleatorias, para estimar la propiedad de interés. A este método se lo suele llamar *Model checking* estadístico [29]. Este método tiene la ventaja de que no es necesario generar el espacio de estados completo, sino que nos alcanza con mantener solo el estado actual de la simulación, pudiendo funcionar incluso para espacios de estados infinitos. Además, siempre y cuando no haya no-determinismo, cualquier modelo estocástico puede ser simulado. Otra diferencia con *Model checking* probabilístico es que en este último tenemos certeza de que el valor obtenido para la propiedad es correcto. Al hacer un análisis estadístico, el valor obtenido es una estimación, y se suele calcular, además del valor, una estimación del error que este puede tener, usualmente en la forma de intervalo de confianza.

Un problema que surge cuando se hace simulación de modelos estocásticos es cuando se quiere estimar la probabilidad de que cierto evento ocurra, cuando esta probabilidad es muy baja. En este contexto, decimos que el evento es *raro*. Esto es un caso bastante común en la verificación de sistemas, porque es posible que

no se pueda garantizar que el sistema no falle nunca, pero se busque garantizar que esta probabilidad sea aceptablemente baja. El problema en estos casos es que puede resultar en un esfuerzo computacional muy grande. Realizando la versión más simple de simulación, se debe simular en promedio una cantidad muy grande de veces para que ocurra el evento por primera vez, y aún habiendo encontrado una ocurrencia del evento, necesitamos muchas más simulaciones para obtener un intervalo de confianza que tenga un error relativo aceptable [22].

Se han desarrollado varias técnicas para lidiar con este tipo de situaciones. La gran mayoría se puede encuadrar en dos categorías: *Importance sampling* (muestreo por importancia) e *Importance splitting* (división por importancia). *Importance sampling* [13] consiste en modificar el modelo para lograr una mayor probabilidad de ocurrencia del evento raro y ajustar el estimador obtenido para que sea válido en el modelo original. *Importance splitting* [15, 22], en cambio, no modifica el modelo, sino la forma en que simula, replicando las ejecuciones que se consideran más aptas para que ocurra el evento raro, y terminando aquellas que se consideran menos aptas, por medio de alguna heurística. Esta última técnica es sobre la cuál está basado este trabajo.

Motivación

La técnica de *Importance splitting* consiste en dividir el conjunto de estados del modelo en grupos de estados con (idealmente) chances semejantes de llegar al evento raro. Usualmente esto se hace definiendo un valor de "importancia" para cada estado, que representa qué tan factible es llegar al evento raro desde el estado. El esquema general consiste en clonar las simulaciones que aumentan de nivel de importancia, o sea, dividirlas en varias que se ejecuten de manera independiente desde el mismo estado. La idea intuitiva es que se dedique más esfuerzo computacional en ejecuciones más propensas a que ocurra el evento raro, dado que éstas aportan información más relevante a la hora de estimar propiedades sobre los mismos.

Los dos tipos de propiedades que se pueden estimar con *Importance splitting* son las propiedades transitorias y las propiedades del estado estacionario.

Las propiedades transitorias se refieren a la probabilidad de que lleguemos a cierto conjunto de estados antes de que ocurra cierta condición. Por ejemplo: probabilidad de que un servidor falle antes de terminar la consulta.

Las propiedades del estado estacionario, en cambio, estudian cómo se comporta el sistema a la larga, y se refieren a la proporción de tiempo en que el sistema se encuentra en cierto subconjunto de estados. Por ejemplo: proporción de tiempo en que el servidor se encuentra inactivo (otra forma de verlo sería la probabilidad de que el servidor se encuentra inactivo si lo observamos en un instante aleatorio).

Existen a su vez varias técnicas de *splitting*. Las mismas difieren en cómo y cuánto se replican las ejecuciones, incluso algunas permitiendo terminar ejecuciones que se alejan del evento raro. Algunas técnicas se comportan mejor que otras dependiendo del tipo de propiedad y el modelo en cuestión.

Trabajamos sobre FIG [8], que es una herramienta de *Importance splitting* que implementa varias heurísticas para obtener buenos valores de importancia, y permite estimar propiedades transitorias y del estado estacionario en modelos IOSA [11]. Esta herramienta implementa únicamente la técnica de *splitting*

RESTART [25], la cuál está pensada sobre todo para hacer análisis del estado estacionario [27], pero no siempre es la mejor para estimar propiedades transitorias, siendo muy sensible a ciertos parámetros, y a tener buenos valores de importancia.

Nuestro primer objetivo es extender FIG con otras dos técnicas de *splitting*: *fixed-effort* [18] y *fixed-success* [1], las cuáles suelen ser má eficientes a la hora de obtener estimadores para el cálculo de propiedades transitorias. Nuestro segundo objetivo es comparar las tres técnicas en varias situaciones experimentales.

Contribución

La contribución del trabajo consiste en la implementación de las técnicas de *splitting fixed-effort* y *fixed-success* en la herramienta FIG, las cuáles presumiblemente se comportan generalmente mejor que la técnica RESTART para la estimación de propiedades transitorias. Además, comparamos el rendimiento de las tres técnicas para varios casos de estudio.

Capítulo 2

El problema de la simulación de eventos raros

En este capítulo presentamos situaciones donde es necesaria la simulación, los problemas que surgen cuando se quiere estimar un evento de baja probabilidad mediante simulación por Montecarlo, y describimos de forma general algunas técnicas para lidiar con este problema de manera más eficiente, en particular las téncicas de *importance sampling* e *importance splitting*.

Eventos raros

Dado un proceso estocástico, denominamos "evento" a cualquier subconjunto de los estados. Decimos que un evento "ocurre" si llegamos a un estado perteneciente al evento durante la ejecución del proceso. Hay varios tipos de propiedades sobre los eventos que nos pueden interesar. Una de las más usuales es la probabilidad de que ocurra el evento en algún momento de la ejecución. Sin embargo, esta probabilidad no siempre es interesante. Por ejemplo, si el grafo de transiciones es fuertemente conexo (lo cual es bastante usual), tendríamos que cualquier evento ocurre con probabilidad 1 en una ejecución infinita.

Por lo tanto, en general nos interesa la probabilidad de que ocurra un evento A antes de que ocurra una determinada condición de parada. La condición de parada puede ser por ejemplo: que ocurra otro evento B, que se hayan hecho más de una cierta cantidad de transiciones, que haya pasado más de un cierto tiempo, etc. Denominamos a este tipo de propiedades, propiedades transitorias (*transient*). Se corresponden a la probabilidad de que valga el predicado LTL [21] $\neg B \ U \ A$.

También puede interesarnos la probabilidad de que ocurra el evento en el estado estacionario (*steady-state*). En nuestro caso, nos referimos de ese modo a la proporción de tiempo en el que se satisface el evento a la larga.

Si bien trabjaremos sobre ambos tipos de propiedades, nos enfocaremos particularmente en las propiedades transitorias.

Ambos tipos de propiedades se pueden calcular de forma analítica de manera exacta, siempre y cuando estemos trabajando con cierto tipos de modelos simples de analizar, como las cadenas de Markov de tiempo discreto o continuo. El problema es que para eso se necesita generar todo el espacio de estados,

y aún pudiéndolo generar, los cómputos son muy costosos, haciendo imposible resolver el problema de forma analítica para modelos con demasiados estados. Por lo tanto, para modelos de muchos estados, o si necesitamos modelos más expresivos que las cadenas de Markov, la opción más viable para estimar estas propiedades es la simulación.

Simulación por Montecarlo

El método más simple de simulación es el método de Montecarlo. Supongamos que queremos estimar la probabilidad p de que ocurra un evento A bajo cierto modelo. Lo que hacemos es hacer n ejecuciones independientes del modelo a partir del estado inicial y estimamos la probabilidad como $\frac{k}{n}$, donde k es la cantidad de ejecuciones en las que ocurrió el evento A. Más formalmente: queremos estimar el parámetro p de una variable Bernoulli X (que es 1 cuando A ocurre, 0 cuando no). Llamamos $\sigma^2 = p(1-p)$ a su varianza. Si $X_1, ..., X_n \sim X$ son independientes, entonces $\hat{p} = \frac{\sum_{i=1}^{n} X_i}{n}$ es un estimador no sesgado para p. Por el Teorema Central del Límite, para "n suficientemente grande", $\hat{p}$ tiene aproximadamente una distribución normal con media p y varianza $\frac{\sigma^2}{n}$. Luego, $\frac{(\hat{p}-p)\sqrt{n}}{\sigma} \sim N(0,1)$. Supongamos por ejemplo que queremos un intervalo de 95 % de confianza para $\hat{p}$. Llamemos $z = \Phi^{-1}(0{,}975) \approx 1{,}96$, donde Φ es la distribución normal acumulada. Tenemos $Pr(-z \leq \frac{(\hat{p}-p)\sqrt{n}}{\sigma} \leq z) = 0{,}95$. Despejando, $Pr(p \in (\hat{p} - \frac{z\sigma}{\sqrt{n}}, \hat{p} + \frac{z\sigma}{\sqrt{n}})) = 0{,}95$. O sea, el margen de error (absoluto) es $\frac{z\sigma}{\sqrt{n}}$. Sin embargo, por lo general nos interesa más el error relativo, sobre todo cuando se trata de eventos raros (por ejemplo, no es lo mismo un error de 0,001 cuando la probabilidad a estimar es 0,1 que cuando es 10^{-9}). El error relativo es $RE = \frac{z\sigma}{p\sqrt{n}} = \frac{z\sqrt{1-p}}{\sqrt{p}\sqrt{n}}$. O sea, si queremos garantizar un error relativo menor a cierto valor RE, necesitamos $n > \frac{z^2(1-p)}{p\,RE^2}$ (que tiende a ∞ cuando p tiende a 0, creciendo en el orden de $\frac{1}{p}$).

Para probabilidades muy chicas, necesitamos por lo tanto una cantidad de ejemplos muy grande para garantizar un error relativo razonable. Por ejemplo, si $p = 10^{-9}$ y queremos un (no tan pequeño) error del 10 %, necesitamos $n > 3{,}84 \times 10^{11}$.

Queda en evidencia el problema de estimar valores de probabilidad muy chicos por medio de simulación. Se han desarrollado varios métodos para lidiar con esto, la mayoría de las cuáles se pueden categorizar en dos técnicas generales: *Importance Sampling*, la cuál describiremos superficialmente, e *Importance Splitting*, en la cuál está basado este trabajo y por lo tanto se verá con mayor detalle.

Importance Sampling

A pesar de que el trabajo no trata sobre *Importance Sampling*, vale la pena describirlo brevemente en el contexto de simulación de eventos raros. Se usa en general para mejorar la eficiencia de Montecarlo para la estimación numérica de integrales, y en particular para la estimación de propiedades de procesos estocásticos [13].

Importance Sampling es una simulación por Montecarlo, pero generando la muestra respecto de una distribución distinta de la que nos interesa, escalando luego por cierto factor para obtener el valor de la propiedad en cuestión. Intuitivamente, la idea es darle un mayor peso a simulaciones donde ocurra el evento raro, ya que nos conviene generar este tipo de escenarios con mayor frecuencia. Si se elige correctamente la nueva distribución, puede resultar en una menor varianza y por consiguiente, menor error relativo). El problema es que no es trivial encontrar una buena distribución, que lleve a una baja varianza del estimador y que se pueda generar fácilmente. Incluso, malas elecciones de la misma pueden llevar a una mayor (o incluso infinita) varianza del estimador, y por lo tanto una convergencia más lenta que Montecarlo [2].

Más formalmente: Tenemos una variable aleatoria X (con densidad f) y queremos calcular $\gamma = E[\psi(X)]$ para alguna función ψ sobre el espacio muestral de X (en el caso de simulación de eventos raros, X sería el conjunto de caminos posibles a partir del estado inicial y $\psi(\pi)$ es 1 si A ocurre en π, 0 si no). El estimador por Montecarlo sería $\hat{\gamma} = (\Sigma_{i=1}^{n}\psi(X_i))/n$, donde $X_1, ..., X_n$ son independientes y tienen densidad f.

Importance Sampling consiste en muestrear X a partir de una densidad distinta f' (la única restricción que imponemos sobre ésta es que $f'(x) > 0$ cuando $\psi(x)f(x) \neq 0$). Llamamos a este procedimiento "cambio de medida" A partir de ahora, cuando escribimos valor esperado, especificamos la densidad que se está usando como subíndice. De la definición de esperanza tenemos que para cualquier función g:

$$E_f[g(X)] = \int g(x)f(x)dx = \int g(x)\frac{f(x)}{f'(x)}f'(x)dx = E_{f'}[g(X)\frac{f(X)}{f'(X)}]$$

Llamamos $L(X) = \frac{f(X)}{f'(X)}$ (se la llama *likelihood ratio*).

Tenemos $\gamma = E_f[\psi(X)] = E_{f'}[\psi(X)L(X)]$. Entonces, podemos generar $X_1, ..., X_n$, ahora a partir de la densidad f', y tenemos de esta forma un estimador $\gamma' = (\Sigma_{i=1}^{n}\psi(X_i)L(X_i))/n$.

Veamos ahora una intuición de por qué podría resultar en un estimador de menor varianza para el caso de probabilidad de eventos raros: Tenemos que

$$Var[\gamma'] = \frac{1}{n}Var_{f'}(\psi(X)L(X)) = \frac{1}{n}(E_{f'}[\psi^2(X)L^2(X)] - \gamma^2)$$

$$Var[\hat{\gamma}] = \frac{1}{n}Var_f(\psi(X)) = \frac{1}{n}(E_f[\psi^2(X)] - \gamma^2) = \frac{1}{n}(E_{f'}[\psi^2(X)L(X)] - \gamma^2)$$

Por lo que si f' es tal que $f'(x) \gg f(x)$ para los x correspondientes al evento raro ($\psi(x) = 1$), tenemos que $L(x) \ll 1$ para estos valores, y por lo tanto $Var[\gamma'] \ll Var[\hat{\gamma}]$.

Incluso, si tomáramos $f'(x) = \frac{f(x)\psi(x)}{\gamma}$ (siempre y cuando $\psi \geq 0$), tendríamos $L(X)\psi(X) = \gamma$, y por lo tanto $Var[\gamma'] = 0$. Tenemos entonces que existe un cambio de medida óptimo, que lleva a un estimador con varianza 0. Aunque sea obvio, cabe aclarar que este cambio de medida no es relevante desde el punto de vista práctico, ya que para aplicarlo necesitamos el valor de γ, que es lo que queríamos obtener en un principio. Sin embargo, vale la pena tenerlo en cuenta como evidencia de lo conveniente que puede ser *Importance Sampling* si se encuentra un buen cambio de medida, aunque no sea necesariamente el óptimo.

En el contexto de simulaciones, *Importance Sampling* puede aplicarse cambiando las probabilidades de transición entre los estados para favorecer transiciones que conduzcan al evento raro. En este caso, a medida que se simula se debe ir manteniendo la probabilidad del camino, tanto en el modelo original como en el modificado, para conocer el *likelihood ratio* del camino en caso de que se llegue al evento raro. Esta técnica también aplica a análisis del estado estacionario, como se expone en [13].

Importance Splitting

Explicamos en esta sección *Importance Splitting* en términos generales. En secciones posteriores se explicará con mayor detalle.

Así como en *Imporance Sampling*, el objetivo es obtener ocurrencias del evento de interés con mayor frecuencia que con una simulación normal. En el caso de *Importance Splitting*, en vez de cambiar el modelo, la idea es simular sobre el mismo modelo, pero "recompensando" simulaciones que "se acerquen" a la ocurrencia del evento y "desfavoreciendo" simulaciones que "se alejen". En otras palabras, a las simulaciones que consideramos más propensas a llegar a un estado raro las "dividimos" (de ahí el nombre *splitting*) en varias simulaciones independientes a partir del mismo estado, mientras que a las simulaciones que consideramos menos aptas para llegar al evento las finalizamos.

En términos más concretos, la idea es definir niveles de importancia sobre los estados, que representan qué tan cerca estamos del evento raro. Si S es el conjunto de todos los estados, y E es el conjunto de estados que representan el evento raro, definimos los conjuntos de estados $A_0 \supset A_1 \supset ... \supset A_n$ (con $A_0 = S$ y $A_n = E$), donde un mayor índice representa una mayor importancia. Por lo general, estos conjuntos se definen por medio de una función de importancia $\Phi : S \to \mathbb{R}$ y umbrales de importancia (*thresholds*) $t_1 < t_2 < ... < t_n$, de modo que $A_i = \{s \in S : \Phi(s) >= t_i\}, 1 \leq i \leq n$. Si definimos los eventos $B_i =$ "se alcanza un estado $s \in A_i$", entonces lo que buscamos es la probabilidad $P(B_n) = P(B_1 \wedge ... \wedge B_n) = P(B_1)P(B_2|B_1)...P(B_n|B_{n-1})$. *Importance Splitting* consiste en estimar cada una de estas probabilidades condicionales por separado (mediante variaciones del método de Montecarlo), y multiplicarlas para obtener la probabilidad del evento raro.

Así como *Importance Sampling* es altamente sensible a la elección del cambio de medida, el éxito de *Importance Splitting* depende en gran medida de la elección de una buena función de importancia (que refleje realmente "cercanía al evento raro" o "chances de llegar al evento raro luego de pasar por este estado") y buenos umbrales (para los cuales las probabilidades condicionales sean balanceadas). La intuición es que si se dan estas condiciones, entonces las probabilidades de transicionar entre niveles de importancia son mucho más altas que la probabilidad del evento raro, por lo tanto se pueden obtener buenos estimadores para las mismas con relativamente pocas iteraciones de Montecarlo. En muchos de los casos, la elección de la función de importancia y los umbrales se hace a mano, pero se han desarrollado técnicas para derivarlos automáticamente a partir del modelo y el evento cuya probabilidad se quiere estimar.

Existen varias variantes de *Importance Splitting*. Todas tienen en común que los estados de A_i que se alcanzaron al estimar $P(B_i|B_{i-1})$ se usan como puntos de partida de la simulación para estimar $P(B_{i+1}|B_i)$ (por eso decimos

que "se recompensan las simulaciones que se acercan al evento raro"). Difieren en cuántas veces se simula para calcular cada una de las probabilidades de transición, y en qué momento se decide terminar las ejecuciones. Describimos brevemente algunas de ellas:

- **Fixed splitting:** Cada vez que una ejecución cruza el k-ésimo umbral de importancia (es decir, su estado entra en el conjunto A_k) por primera vez, ésta se divide en c_k ejecuciones independientes a partir del mismo estado.

- **RESTART:** Variante de fixed splitting, en la que siempre que una ejecución cruza el k-ésimo threshold, se generan $c_k - 1$ ejecuciones independientes nuevas, las cuales terminan cuando la ejecución baja del nivel de importancia k (es decir, su estado sale del conjunto A_k).

- **Fixed effort:** Para cada nivel de importancia k, se determina una cantidad fija de ejecuciones c_k que se usan para estimar $P(B_{k+1}|B_k)$. Los c_k estados de partida se toman de las ejecuciones que alcanzaron el nivel k en el paso anterior (se pueden muestrear uniformemente, o asignar aproximadamente la misma cantidad a cada una de las posibilidades, lo cual resulta en una menor varianza).

- **Fixed success:** Similar a fixed effort, pero en lugar de fijar la cantidad de ejecuciones, se fija la cantidad de veces que queremos llegar al siguiente nivel. En este caso, tenemos un cierto control sobre la imprecisión del estimador, ya que paramos cuando vemos el evento raro una suficiente cantidad de veces, pero el costo computacional puede tener una gran varianza.

- **Fixed probability of success:** En este enfoque, los umbrales de importancia no se fijan de antemano, sino que se construyen incrementalmente, de modo que la probabilidad de pasar de un nivel de importancia al siguiente sea aproximadamente un valor fijo $\frac{k}{n}(0 < k < n)$. En el primer paso se simulan n ejecuciones independientes a partir del estado inicial. En cada uno de los pasos subsiguientes se toman las n ejecuciones resultantes del paso anterior y se ordenan por el máximo valor de la función de importancia que se alcanzó. Las k mayores se mantienen igual para la siguiente iteración. Se toma el estado donde se alcanzó el mayor valor de importancia en la ejecución $(n-k)-$ésima. Si el estado se corresponde con el evento raro, se para. Si no, a partir de este estado se simulan $n-k$ ejecuciones independientes, que reemplazan a las menores $n-k$ del paso anterior. De este modo mantenemos siempre un conjunto de n ejecuciones para la iteración siguiente.

Nuestro objetivo es extender una herramienta ya existente (la cuál implementa únicamente la técnica RESTART), para que se pueda simular usando Fixed effort o Fixed success, así como comparar la eficiencia de las tres técnicas para algunos casos de estudio.

El resto del trabajo se estructura como sigue:

- En el capítulo 3 describimos el tipo de modelos sobre los que haremos simulaciones (*Input-Output Stochastic Automata*).

- En el capítulo 4 describimos en mayor detalle las diferentes técnicas de splitting y describimos brevemente los tipos de funciones de importancia que utilizaremos.
- En el capítulo 5 presentamos brevemente la herramienta FIG, y damos algunos detalles acerca de la implementación de las nuevas técnicas de splitting.
- En el capítulo 6 presentamos los casos de estudio que analizaremos y mostramos los resultados obtenidos en términos de error relativo de los estimadores.

Capítulo 3

Input-Output Stochastic Automata

En este capítulo presentamos el modelo estocástico que usaremos para las simulaciones (*Input-Output Stochastic Automaton*, de aquí en adelante abreviado IOSA). Este modelo tiene la ventaja de que es composicional, es más expresivo que modelos más comúnmente usados (como las CTMC), y a la vez es determinista bajo ciertas condiciones (y por lo tanto, apto para aplicar simulación).

Definición de IOSA

Definición 1. Un *Input-Output Stochastic Automaton (IOSA)* es una tupla $(S, A^I, A^O, \mathcal{C}, \rightarrow, C_0, s_0)$, donde:

- S es un conjunto enumerable de estados.
- A^I y A^O son conjuntos enumerables y disjuntos de *labels* (o etiquetas). Los labels de A^I se llaman *de entrada*, mientras que los de A^O son *de salida*. Llamamos $A = A^I \cup A^O$.
- $\mathcal{C}$ es un conjunto finito de *clocks* (relojes). Cada uno de los elementos $x \in \mathcal{C}$ tiene asociado una medida de probabilidad continua μ_x sobre $\mathbb{R}$ (es decir, $\mu_x(\{t\}) = 0$ para cualquier $t \in \mathbb{R}$), que satisface además $\mu_x(\mathbb{R}_{\leq 0}) = 0$.
- $\rightarrow \in S \times \mathcal{P}(\mathcal{C}) \times A \times \mathcal{P}(\mathcal{C}) \times S$ es la relación de transición. Se usa la notación $s \xrightarrow{C,a,C'} s'$ para representar $(s, C, a, C', s') \in \rightarrow$. Representa que si se está en el estado s, cuando expiran los relojes de C se puede transicionar a s' reseteando los relojes de C', es decir, asignando a cada reloj $x \in C'$ un valor generado de acuerdo a la distribución correspondiente μ_x.
- $C_0 \subseteq \mathcal{C}$ es el conjunto de relojes que están inicializados al inicio.
- $s_0 \in S$ es el estado inicial.

Además, los IOSA tienen que satisfacer las siguientes restricciones sobre la relación $\rightarrow$:

(a) Si $s \xrightarrow{C,a,C'} s'$ y $a \in A^I$ entonces $C = \varnothing$.

(b) Si $s \xrightarrow{C,a,C'} s'$ y $a \in A^O$ entonces $|C| = 1$.

(c) Si $s \xrightarrow{\{x\},a_1,C_1} s_1$ y $s \xrightarrow{\{x\},a_2,C_2} s_2$, entones $a_1 = a_2, C_1 = C_2, s_1 = s_2$.

(d) Si $s \xrightarrow{\{x\},a,C'} s'$, entonces para cualquier transición $t \xrightarrow{C_1,b,C_2} s$, o bien $x \in C_2$, o $x \notin C_1$ y hay una transición $t \xrightarrow{\{x\},c,C_3} t'$.

(e) Si $s_0 \xrightarrow{\{x\},a,C} s$, entonces $x \in C_0$.

(f) Para cualquier $a \in A^I$ y cualquier estado s, existe una transición de la forma $s \xrightarrow{\varnothing,a,C} s'$.

(g) Si $s \xrightarrow{\varnothing,a,C_1} s_1$ y $s \xrightarrow{\varnothing,a,C_2} s_2$, entonces $C_1 = C_2$ y $s_1 = s_2$.

Las acciones de output se manejan mediante la expiración de clocks. La restricción (b) indica que cada acción de output reacciona a la expiración de un único clock. El paso del tiempo se puede dar sólamente si no hay alguna transición de output activa, es decir, con su clock ya expirado. Las transiciones de input, por el contrario, están siempre activas dado que no reaccionan a la expiración de ningún clock (restricción (a)). Además no impiden el paso del tiempo, por lo que hay no-determinismo cuando hay transiciones de input activas (puede ocurrir la acción o el paso del tiempo). Como veremos, este no-determinismo se resuelve mediante composición paralela, sincronizándose con la acción de output de otro IOSA.

La restricción (c) dice que para cada estado y cada clock, hay a lo sumo una transición que se activa desde ese estado cuando expira ese clock, y la acción de output que se emite es única y determinada. De todos modos, esta restricción por sí sola no alcanza para garantizar que a lo sumo una transición de output se activa al mismo tiempo desde un estado, ya que podría ocurrir que varios clocks estén expirados en el momento en que se entra a un estado. Las restricciones (d) y (e) garantizan que esto no ocurre. La restricción (d) establece que si un clock activa una transición desde cierto estado, entonces para cualquier transición hacia ese estado, o bien se resetea el clock, o bien no se expira al traniscionar y además desde el estado anterior también se activa una transición con el mismo clock. En cualquiera de los dos casos, se tiene que el clock no había expirado estrictamente antes de entrar al estado. La restricción (e) establece que para cualquier transición de output desde el estado inicial, el clock correspondiente fue inicializado al comenzar la ejecución. Entre las restricciones (d) y (e) garantizan que nunca hay un clock expirado al momento de entrar a un estado, a menos que en algún momento de la ejecución hayan dos clocks iguales, lo cuál no ocurre casi nunca (es decir, ocurre con probabilidad 0) ya que los clocks son variables continuas.

Entre las restricciones (c), (d) y (e) garantizan que casi nunca se activan dos transiciones de output distintas a la vez. Es decir, se comportan de manera determinista.

La restricciones (f) y (g) dicen que para cada estado y acción de input hay una y sólo una transición correspondiente. En una composición, la restricción (f)

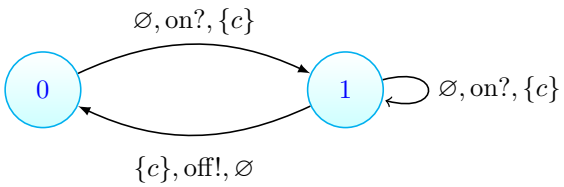

Figura 3.1: Ejemplo de IOSA: interruptor

garantiza que el output no bloquea en sincronización con input, y la restricción (g) que se mantiene el determinismo luego de la composición.

Un ejemplo simple de sistema que podemos modelar con un IOSA puede ser el siguiente: Supongamos que tenemos una habitación con un interruptor que enciende una luz. Una vez pulsado el interruptor, la luz permanece encendida durante un cierto tiempo aleatorio (por ejemplo, uniforme entre 10 y 15 minutos) y luego se apaga automáticamente. Se puede pulsar el interruptor mientras la luz está encendida, y en tal caso se reinicia el tiempo en el que se apagará. Se puede modelar el encendido de la luz como una acción de input, dado que depende del entorno del sistema, mientras que el apagado se puede ver como una acción de output, dado que se produce como reacción a la expiración de cierto tiempo aleatorio. El diagrama del IOSA se muestra en la Figura 3.1. Los estados 0 y 1 representan que la luz está apagada y encendida respectivamente. El clock c es el que marca el tiempo en el que el interruptor se apaga, y tiene asociado una distribución de probabilidad. Por convención, las acciones de input se representan con un "?" al final y las de output con un "!". En el lenguaje de descripción de IOSAs, este modelo se representa como sigue (creemos que la sintaxis del lenguaje es lo suficientemente clara como para no requerir explicación):

```
module Light
        s: [0..1] init 0;
        c: clock;
        [on?] true -> (s' = 1) & (c' = uniform(10,15))
        [off!] s == 1 @ c -> (s' = 0)
endmodule
```

Semántica de IOSA

Si bien ya dimos una descripción informal del funcionamiento de los IOSA, definiremos la semántica más formalmente, para mayor claridad. La semántica de IOSA se define en términos de *Non-deterministic Labeled Markov Processes* (NLMP).

Definición 2. Un *NLMP* es una tupla $(S, \Sigma, \{T_a | a \in L\})$, donde Σ es una σ-álgebra en el conjunto de estados S y para cada etiqueta $a \in L$ tenemos que $T_a : S \to \Delta(\Sigma)$ es medible desde Σ a la *hit*-σ-álgebra $H(\Delta(\Sigma))$.

La definición involucra nociones más complejas de teoría de la medida, pero lo único que nos interesa saber para entender la semántica de IOSA es que $\Delta(\Sigma)$ es una σ-álgebra que se construye sobre las medidas de probabilidad en el espacio (S, Σ). Es decir, las funciones de transición T_a mapean cada estado a un conjunto de distribuciones de probabilidad sobre los estados. Este conjunto representa las transiciones que se pueden hacer, que están etiquetadas con el label a. El autómata se dice no-determinista porque según la definición pueden existir transiciones para varios labels desde el mismo estado, y para cada label pueden ocurrir distintas transiciones que resulten en distintas distribuciones sobre los estados.

La semántica de IOSA se define como un NLMP que incorpora en su estado el valor de cada reloj, además del estado del IOSA. Tiene dos tipos de transición: Uno que representa el paso del tiempo, restándole a los clocks el mismo valor. El otro tipo se condice con las transiciones del IOSA, es decir cambia el estado del IOSA y hace remuestreo de los clocks.

Para esta definición usamos la notación δ_a que representa la medida de probabilidad concentrada en $\{a\}$ (medida de Dirac).

Definición 3. Dado un IOSA $\mathcal{I} = (S, A^I, A^O, \mathcal{C}, \dashrightarrow C_0, s_0)$, donde $\mathcal{C} = \{x_1, ..., x_n\}$ su semántica está dada por el NLMP $\mathcal{P}(\mathcal{I}) = (\boldsymbol{S}, \boldsymbol{\Sigma}, \{T_a | a \in L\})$, donde

- $\boldsymbol{S} = (S \cup \{\text{init}\}) \times \mathbb{R}^n, L = A \cup \mathbb{R}_{>0} \cup \{\text{init}\}$, con init $\notin (A \cup \mathbb{R}_{>0} \cup S)$.
- $\boldsymbol{\Sigma}$ es la σ-algébra de Borel sobre $\boldsymbol{S}$.
- $T_{\text{init}}(\text{init}, \vec{v}) = \{\delta_{s_0} \times \prod_{i=1}^n \mu_{x_i}\}$.
- $T_a(s, \vec{v}) = \{\delta_{s'} \times \prod_{i=1}^n \bar{\mu}_{(\vec{v}, C', x_i)} | s \xrightarrow{C,a,C'} s' \wedge \forall x_i \in C : \vec{v}(i) \leq 0\}$ para cada $a \in A$, $s \in S$, donde $\bar{\mu}_{(\vec{v}, C', x_i)} = \mu_{x_i}$ si $x_i \in C'$, $\bar{\mu}_{(\vec{v}, C', x_i)} = \delta_{\vec{v}(i)}$ caso contrario.
- $T_d(s, \vec{v}) = \{\delta_s \times \prod_{i=1}^n \delta_{\vec{v}(i)-d} | 0 < d \leq min\{\vec{v}(i) | \exists a \in A^O, C' \subseteq \mathcal{C}, s' \in S : s \xrightarrow{\{x_i\},a,C'} s'\}\}$ para cada $d \in \mathbb{R}_{>0}, s \in S$.

El estado es un producto cartesiano de un estado del IOSA (o init) con n valores reales que representan el valor de cada reloj.

El estado adicional init se agrega para representar la inicialización de los clocks. Desde este estado sólo sale una transición hacia s_0, con los clocks inicializados según la medida correspondiente.

Las transiciones discretas del IOSA están representados por las transiciones T_a. Una transición $s \xrightarrow{C,a,C'} s'$ sólo es posible si los relojes de C están expirados (su valor es no positivo). En tal caso se transiciona a s' y los valores de los relojes se mantienen igual para los relojes que no están en C', mientras que se muestrea un nuevo valor para los relojes de C', usando la correspondiente distribución.

El paso del tiempo está representado por las transiciones $T_d, d \in \mathbb{R}_{>0}$. Para que ocurran estas transiciones se requiere que d sea menor o igual que los valores de todos los relojes para los cuáles se habilita una transición de output en el momento de su expiración. Dicho de otra forma, en las próximas d unidades de tiempo (excepto en el instante d unidades más tarde) no se debe activar ninguna transición de output desde el estado actual.

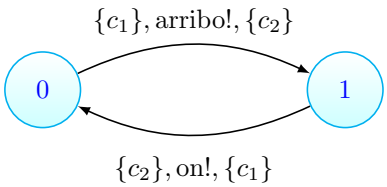

Figura 3.2: Ejemplo de IOSA: persona

Composición paralela

Pasamos a definir composición paralela que, como en otros tipos de modelo, permite la descripción de sistemas más complejos mediante la ejecución simultánea sincronizada de varias componentes. En el caso de IOSAs, la sincronización se realiza mediante las acciones que son de input en uno de los autómatas y de output en otro. Para que se preserven las propiedades de IOSAs en la composición, exigimos que los autómatas que se componen no tengan clocks en común, y como no sincronizamos entre ouputs, requerimos que no tengan acciones de ouput en común.

Definición 4. Dos IOSAs $\mathcal{I}_1, \mathcal{I}_2$ se dicen *compatibles* si no comparten acciones de output ni clocks, es decir $A_1^O \cap A_2^O = \varnothing,\ \mathcal{C}_1 \cap \mathcal{C}_2 = \varnothing$.

Definición 5. Dados dos IOSAs compatibles $\mathcal{I}_1$, $\mathcal{I}_2$, la composición paralela $\mathcal{I}_1 \mid\mid \mathcal{I}_2$ es un IOSA $(S_1 \times S_2, A^O, A^I, \mathcal{C}_1 \cup \mathcal{C}_2, \rightarrow, C_0^1 \cup C_0^2, (s_0^1, s_0^2))$, donde $A^O = A_1^O \cup A_2^O$, $A^I = (A_1^I \cup A_2^I) \setminus A^O$ y $\rightarrow$ es la menor relación dada por las siguientes reglas:

$$\frac{s_1 \xrightarrow{C,a,C'} s_1'}{(s_1, s_2) \xrightarrow{C,a,C'} (s_1', s_2)} \quad (a \in (A_1 \setminus A_2))$$

$$\frac{s_2 \xrightarrow{C,a,C'} s_2'}{(s_1, s_2) \xrightarrow{C,a,C'} (s_1, s_2')} \quad (a \in (A_2 \setminus A_1))$$

$$\frac{s_1 \xrightarrow{C_1,a,C_1'} s_1' \quad s_2 \xrightarrow{C_2,a,C_2'} s_2'}{(s_1, s_2) \xrightarrow{C_1 \cup C_2, a, C_1' \cup C_2'} (s_1', s_2')} \quad (a \in A_1 \cap A_2)$$

La composición paralela entre dos IOSAs compatibles es, efectivamente, un IOSA, como se prueba en [11].

Para ilustrar con un ejemplo de compisición, tomamos el IOSA del interruptor de la Figura 3.1 y lo componemos con un IOSA que representa una persona que entra a la habitación en donde está el interruptor y enciende el interruptor al tiempo que deja la habitación (Figura 3.2). Usamos dos relojes, uno para controlar el arribo de la persona a la habitación (c_1), y el otro para el encendido del interruptor (c_2). Al componer estos IOSAs, se sincroniza la acción "on" de ambos, pasando a a ser una acción de output. El IOSA que resulta de la composición se muestra en la Figura 3.3

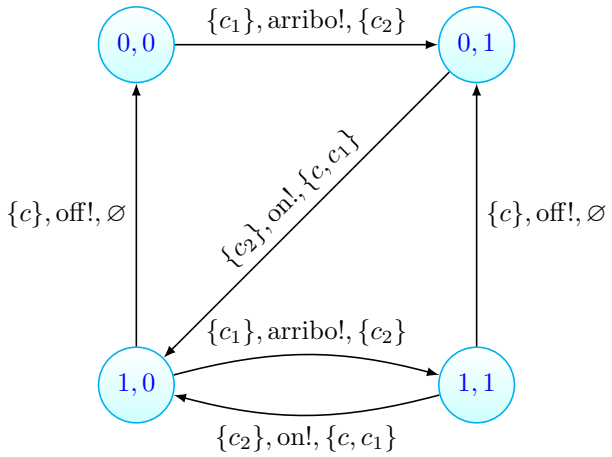

Figura 3.3: Composición de IOSAs interruptor y persona

Un IOSA se dice determinístico cuando casi nunca (es decir, con probabilidad 0) ocurre que se llega a un estado $(s, \vec{v}) \in \boldsymbol{S}$ tal que $\cup_{a \in A} T_a(s, \vec{v})$ tiene al menos dos medidas de probabilidad distintas.

Como se mencionó antes, la existencia de acciones de input introduce no-determinismo en la ejecución de los IOSAs, por lo que un IOSA con acciones de input no puede ser sujeto a simulación. Decimos que un IOSA es *cerrado* si no tiene acciones de input. Entonces, para el propósito de simulación nos interesan en particular los IOSAs cerrados. Una forma de obtener un IOSA cerrado es mediante composición paralela. En tal caso, es posible que algunas componentes tengan acciones de input, siempre y cuando éstas sean sincronizadas con acciones de output de otro IOSA al realizar la composición.

Gracias a las restricciones en la definición de IOSAs, cuya motivación intuitiva ya dimos al explicarlas, se tiene que todo IOSA cerrado es determinístico (la prueba puede encontrarse en [11]). Esto nos da la justificación teórica para que tenga sentido modelar sistemas mediante el uso de IOSAs y luego someterlos a simulación, siempre y cuando la composición paralela sincronice todas las acciones de input.

Capítulo 4

Importance Splitting

En este capítulo trataremos más en detalle lo introducido en el capítulo 2. Formularemos en detalle el problema de la simulación de eventos raros, presentaremos el algoritmo de Importance Splitting con sus variantes y justificaremos su ventaja para resolver el problema.

Marco general de importance splitting

Asumimos que el sistema que queremos analizar se puede describir mediante un proceso de Markov $X = \{X_t, t \geq 0\}$. Tenemos un espacio medible (S, Σ), tal que X_t es una variable aleatoria que toma valores en S, al cuál denominamos el conjunto de estados del sistema. Σ es una σ-álgebra sobre S, que representa el conjunto de eventos (es decir, un evento es un subconjunto medible de S).

El tiempo t puede ser continuo (real) o discreto (entero no negativo). En nuestro caso, dado que trabajamos con modelos IOSA, asumimos que el tiempo es continuo.

Respecto a los IOSAs, notemos que no se cumple la propiedad de Markov si consideramos el conjunto S del IOSA (que en tal contexto se llama *conjunto de estados*) como el conjunto de estados del proceso estocástico. Sin embargo, como se vio, la semántica de IOSAs cerrados se puede expresar mediante un proceso de Markov determinístico (para obtener el estado del proceso de Markov, se extiende el estado del IOSA con los valores de los clocks). Por lo tanto, en el contexto de importance splitting, aclaramos que S no representa el conjunto de estados del modelo IOSA, sino los estados del proceso de Markov correspondiente al IOSA (el NLMP dado por la semántica).

Definimos el tiempo de entrada a un evento $A \in \Sigma$ como $T_A = \text{ínf}\{t : X_t \in A\}$.

El tipo de probabilidades que queremos estimar mediante simulación son las de la forma $P(T_A \leq T)$, donde a A es un evento al cual lo denominamos "evento raro" y T es una variable aleatoria con esperanza finita, la cual denominamos "tiempo de parada". Usualmente, el tiempo de parada lo definimos como la ocurrencia de otro evento B, denominado condición de parada (se requiere que la probabilidad de que B ocurra en algún tiempo sea 1). De este modo, la probabilidad que queremos calcular es $P(T_A \leq T_B)$, que se puede expresar

mediante la fórmula PCTL [14]

$$P(\neg B \ U \ A)$$

En el contexto de simulación de eventos raros, este tipo de análisis se suele denominar *transient analysis* (análisis transitorio).

El otro tipo de análisis que suele hacerse en este contexto, es el *steady-state analysis* (análisis en el estado estacionario), en el cuál se quiere calcular $\lim_{t\to\infty} P(X_t \in A)$ para un evento raro A. Esto se puede expresar con la fórmula CSL [3] $S(A)$. Dejaremos de lado este tipo de análisis para focalizarnos en el análisis transitorio.

Como ya se expuso en la sección 2, si la probabilidad que queremos estimar es muy baja, resulta en un error relativo muy grande del estimador si realizamos la simulación más simple posible (Montecarlo). Sin embargo, en la práctica, tiene sentido que la ocurrencia de un evento raro se de luego de visitar ciertos estados intermedios, que se visitan con mayor probabilidad que el evento raro, y desde los cuáles es más probable llegar al evento raro. De ahí surge la idea de importance splitting, que da un mayor peso a las ejecuciones que más "se acercan" al evento raro y desfavorece a aquellas que se alejan, para de este modo alcanzar el evento raro más veces con la idea de obtener una menor varianza en el estimador.

Dado el proceso de Markov X con conjunto de estados S y un evento raro $A \subset S$, la idea de importance splitting es definir *niveles de importancia*

$$S = A_0 \supset A_1 \supset A_2 ... \supset A_n = A$$

La manera más usual de definir estos niveles es mediante una función de importancia $\Phi : S \to \mathbb{R}$ y umbrales de importancia (*thresholds*) $t_1 < t_2 < ... < t_n$, de modo que $A_i = \{s \in S : \Phi(s) \geq t_i\}, 1 \leq i \leq n$.

Definimos la variable de tiempo de entrada a un nivel de importancia

$$T_k = T_{A_k} = \text{ínf}\{t : \Phi(X_t) \geq t_i\}$$

y el evento

$$E_i = \{T_i \leq T\}$$

donde T es el tiempo de parada. Notemos que los eventos E_i, a diferencia de los A_i, no son eventos sobre el espacio de estados del proceso de Markov, sino sobre el proceso en sí. Luego, la probabilidad que queremos estimar es justamente $P(E_n)$. Llamamos a este valor p.

Dado que $E_i \subseteq E_j$ para $i > j$, tenemos que

$$p = P(E_0 \cap ... \cap E_n) = P(E_1|E_0)P(E_2|E_1)...P(E_n|E_{n-1})$$

(Como $A_0 = S$, $P(E_0) = 1$). Llamamos $p_k = P(E_k|E_{k-1})$.

El objetivo de las técnicas de importance splitting, es calcular separadamente estas probabilidades p_k para obtener así un mejor estimador para p.

Asumimos que el estado inicial se encuentra en el nivel más bajo de importancia $(S - A_1)$[1].Para estimar p_1 podemos simular por Montecarlo desde el

[1]No perdemos generalidad con esta suposición, porque en caso de que no ocurra, podemos crear un nuevo estado inicial con el nivel de importancia más bajo, que transiciona inmediatamente al estado inicial original.

estado inicial N_0 veces hasta el tiempo $min(T, T_1)$. Es decir, hasta llegar a un estado que cumpla la condición de parada o esté en A_1. Si llegan R_1 simulaciones a un estado de A_1 ("suben de nivel de importancia"), entonces tenemos el estimador $\hat{p}_1 = \frac{R_1}{N_0}$. Luego, estas R_1 simulaciones son una muestra de las ejecuciones en las que $T_1 \leq T$, y podemos usarlas como punto de partida para estimar $p_2 = P(T_2 \leq T | T_1 \leq T)$. Es decir, podemos simular N_1 réplicas a partir de los estados que llegaron a A_1. Si $N_1 > R_1$ (que será por lo general el caso que se dé), entonces algunas ejecuciones deben ser elegidas varias veces. De ahí el nombre *splitting*, ya que es como si la ejecución se "dividiera" en varias ejecuciones independientes distintas al cruzar un umbral de importancia.

Si R_2 de las N_1 réplicas llegan a A_2 antes de que ocurra la condición de parada, entonces tenemos un estimador $\hat{p}_2 = \frac{R_2}{N_1}$. Estas R_2 réplicas se pueden usar de manera análoga para estimar $\hat{p}_3$ y así sucesivamente. Finalmente, estimamos $\hat{p}$ como el producto de los $\hat{p}_i$. Si en algún paso $R_i = 0$, entonces estimamos $\hat{p} = 0$.

Dicho más concretamente, el algoritmo general funciona de la siguiente manera:

- Al principio elegimos $N_0 \in \mathbb{N},\ R_0 = 1,\ t_0^1 = s_0$ (Los t_k^i son los estados que llegaron al nivel k desde el anterior).
- En el paso k $(0 \leq k < n)$ tenemos una muestra de estados de entrada al nivel k $(t_k^1, ..., t_k^{R_k})$. Desde estos muestreamos una cierta cantidad N_k de alguna forma y simulamos hasta $min(T_k, T)$. Contamos R_{k+1}: cantidad de ejecuciones en que ocurre E_{k+1} (o sea, $T_{k+1} \leq T$). Si $R_{k+1} = 0$, terminamos el proceso y estimamos $\hat{p}_{k+1} = 0$. Caso contrario, $\hat{p}_{k+1} = \frac{R_{k+1}}{N_k}$ e inicializamos para el paso siguiente los $(t_{k+1}^1, ..., t_{k+1}^{R_{k+1}})$ como las ejecuciones en las que ocurrió E_{k+1}.
- Si al final del proceso tenemos que $R_k > 0$ para todo k, entonces tenemos el estimador

$$\hat{p} = \prod_{k=1}^{n} \frac{R_k}{N_{k-1}}$$

Técnicas de splitting

Las distintas técnicas de splitting difieren en cómo se eligen los N_i y en cómo se muestrean las ejecuciones a partir de las ejecuciones que llegaron al nivel. Las técnicas que nos interesan son RESTART (que es la que estaba implementada originalmente en la herramienta FIG), *fixed-effort* y *fixed-success* (que son las técnicas que se implementaron como parte del trabajo).

RESTART

El método RESTART, inicialmente propuesto en [25], está basado en la técnica de *fixed-splitting*, la cual explicamos a continación: Antes de empezar se determinan valores fijos $c_k > 1 (1 \leq k < n)$, que se denominan "el factor de splitting en el nivel k". La idea es que cada ejecución que entre por primera vez al nivel A_k se divida en c_k ejecuciones independientes. En términos del framework

general de splitting, elegimos $N_k = c_k R_k (1 \leq k < n)$, muestreando cada una de las R_k ejecuciones exactamente c_k veces.

Notemos que esta forma permite simular completamente una ejecución y todas sus ramificaciones antes de pasar a las siguientes ejecuciones en el mismo nivel (siguiendo un orden *depth-first*). Es decir, cuando llegamos a un estado de A_k, podemos simular de inmediato c_k ejecuciones independientes a partir de este estado y una vez que estas (y todas sus ramificaciones) concluyan ya no es necesario guardar el estado de entrada a A_k. De este modo, no tenemos que guardar todos los estados de entrada a cada nivel, sino uno a la vez por nivel, lo cual puede ser más eficiente en memoria.

En el caso de *fixed-splitting*, tenemos que el estimador se simplifica a

$$\hat{p} = \frac{R_n}{N_0 \prod_{k=1}^{n-1} c_k}$$

RESTART modifica *fixed-splitting* para que permita terminar algunas ejecuciones que bajan de nivel de importancia. La intuición detrás de esto es que una ejecución que baja de nivel de importancia puede ser muy poco propensa a volver a subirlo. Por lo tanto, ahorramos el esfuerzo computacional de simular muchas ejecuciones que muy probablemente no lleguen al siguiente nivel de importancia.

Para cada ejecución que sube un nivel de importancia (su estado entra al conjunto A_k), la dividimos en c_k ejecuciones independientes, de las cuales a una la marcamos como "la ejecución original" y continúa normalmente como en el caso de *fixed-splitting*. Las $c_k - 1$ restantes, además de terminar en caso de la ocurrencia de la condición de parada, terminan cuando se baja de nivel de importancia, es decir, cuando se sale del conjunto A_k. Para compensar el hecho de que se terminan estas ejecuciones prematuramente, el splitting se hace siempre que una ejecución cruza un nivel de importancia, no sólo la primera vez como en el caso del resto de algoritmos de splitting. Dicho de otra forma, si "la ejecución original" que resulta luego de cruzar un nivel de importancia baja de nivel de importancia y al continuar su ejecución vuelve a subir el mismo nivel de importancia, entonces se vuelve a hacer la división.

Para expresarlo en términos más concretos, definimos dos nuevos eventos para una ejecución:

- C_k: La ejecución "sube al nivel de importancia k". Es decir, se realizó una transición $s \to s'$ tal que $s \notin A_k$ y $s' \in A_k$. Notar que este evento no es lo mismo que el evento E_k, dado que C_k tiene en cuenta todas las veces que una ejecución sube de nivel de importancia, no solamente la primera.
- D_k: La ejecución "baja del nivel de importancia k". Es decir, se realizó una transición $s \to s'$ tal que $s \in A_k$ y $s' \notin A_k$.

El algoritmo es como sigue:

- Se inician N_0 ejecuciones independientes a partir del estado inicial.
- Cuando ocurre un evento C_n en una ejecución, es porque llegamos al evento raro. Terminamos dicha ejecución e incrementamos la cantidad de ocurrencias del evento raro (R_n).
- Cuando ocurre la condición de parada, terminamos la ejecución.

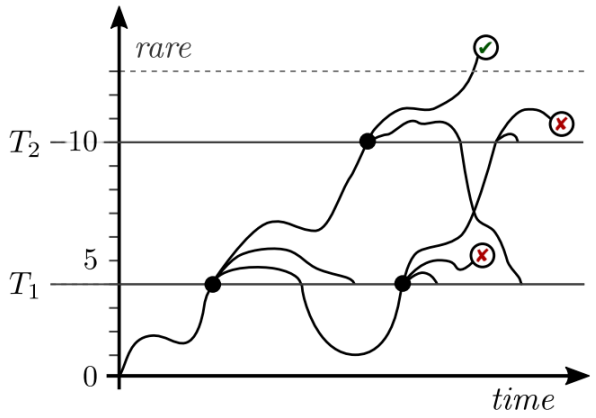

Figura 4.1: Ejemplo de ejecución de RESTART [2]

- Cuando ocurre un evento C_k con $k < n$, se interrumpe la ejecución, guardando su estado. Se generan $c_k - 1$ ejecuciones independientes a partir del mismo estado (las denominamos *copias*). Pero estas ejecuciones difieren de la original en que terminan cuando ocurre un evento D_k (además de ocurrencia de condición de parada o evento raro). Luego de que estas $c_k - 1$ copias terminan, se prosigue normalmente con la ejecución original.

  Vale la pena aclarar que estas copias se comportan igual que la ejecución original respecto a eventos que ocurren en niveles superiores. Es decir, por ejemplo, si ocurre un evento C_{k+1} en una copia, entonces esta se interrumpirá y se ejecutarán $c_{k+1} - 1$ ejecuciones independientes desde su estado, que terminan cuando ocurre D_{k+1}. Luego de esto, la copia continúa normalmente.

- Se debe considerar el caso de que una ejecución sube varios niveles de importancia al mismo tiempo. En tal caso, se considera que ocurren todos los eventos C_i correspondientes (partiendo del primero), tanto para la ejecución original, como para las copias de cierto nivel para el nivel siguiente. Por ejemplo, supongamos que en una de las ejecuciones originales ocurren los eventos C_1 y C_2 simultáneamente. Entonces, se originan $c_1 - 1$ copias que terminan en D_1, y $c_1(c_2 - 1)$ copias que terminan en D_2 (porque es como si el evento C_2 hubiera ocurrido tanto para la ejecución original como para las copias originadas por la ocurrencia de C_1).

En la Figura 4.1 se muestra un ejemplo de ejecución de RESTART, donde la coordenada y representa la importancia de la simulación correspondiente y T_1 y T_2 son umbrales de importancia.

A pesar de la modificación de RESTART respecto a *fixed-splitting*, la implementación que se describió sigue usando un enfoque *depth-first*, por lo que en cada momento sólo es necesario guardar los estados de entrada a cada nivel.

El estimador para el caso de RESTART es el mismo que el estimador simplificado de *fixed-splitting*. Notemos que este estimador no se basa fuertemente

[2] Imagen extraída de [5]

en el esquema general de algoritmos de splitting de calcular los $\hat{p}_k$ por separado para obtener el estimador de $\hat{p}$.

El estimador dado por RESTART es insesgado. La prueba de esto es por inducción en la cantidad de niveles de importancia y se expone en [26].

El algoritmo 1 resume una corrida del método RESTART.

Algorithm 1: Técnica RESTART. `NULL` es cualquier estado en B, $s.next()$ representa un paso de simulación, $s.level()$ es el mayor k tal que $s \in A_k$, asumimos $c_n = 1$.

```
Function restart(s = s_0, l = 0):
    if l = n then
        R_n ← R_n + 1                          // contamos el evento raro
        return NULL                            // terminamos la ejecución
    while s ∉ B ∧ s.level() ≥ l do
        if s.level() > l then
            for i ← 1 to (c_{l+1} − 1) do
                restart(s, l + 1)                              // splitting
            s ← restart(s, l + 1)                  // ejecución principal
        else
            s ← s.next()
    return s
```

Fixed effort

La técnica de *fixed-effort*, descripta en [18], es la variante más simple de splitting, en términos del marco general descripto anteriormente.

En el caso de esta técnica, tenemos que los N_i están fijos de antemano. De ahí el nombre *fixed-effort*, dado que de cierto modo se controla el esfuerzo computacional de la simulación, fijando la cantidad de ejecuciones que se hacen por nivel.

La simulación se debe hacer de a un nivel por vez, porque necesitamos todos los estados de entrada a un nivel antes de pasar al nivel siguiente. Cuando simulamos el k-ésimo paso, elegimos N_{k-1} veces un estado de los que entraron al nivel $k-1$ (o en el primer paso: el estado inicial) y simulamos desde este estado hasta llegar al nivel siguiente, o hasta que se cumpla la condición de parada. En caso de que se llegue al siguiente nivel de importancia, agregamos el estado de entrada a la lista de estados para el siguiente paso. El estimador es tal cual el que se describió en el marco general: $\hat{p} = \prod_{k=1}^{n} \frac{R_k}{N_{k-1}}$.

Existen varias variantes de *fixed-effort*, dependiendo de cómo elijamos los N_{k-1} estados entre los R_{k-1} estados que entraron al nivel $k-1$. Una forma de hacerlo es mediante *asignación aleatoria*: Cada uno de los estados de partida se elige independientemente de manera uniforme entre los R_{k-1} estados posibles. Otra forma es *asignación fija*, es decir elegir cada uno de los estados posibles aproximadamente la misma cantidad de veces: Elegimos cada uno de los

R_{k-1} estados $\left\lfloor \frac{N_{k-1}}{R_{k-1}} \right\rfloor$ veces, y elegimos aleatoriamente N_{k-1} *mod* R_{k-1} estados distintos para realizar una ejecución adicional a partir de los mismos.

Ambas versiones del algoritmo resultan en estimadores insesgados, pero la segunda resulta en un estimador de menor varianza, dado que es una forma de muestreo estratificado de los estados del nivel anterior [19]. Por lo tanto, nos inclinamos por la técnica de *asignación fija*.

Fixed-effort tiene cierta desventaja con respecto a RESTART en términos de memoria, dado que se deben guardar todos los estados de entrada al siguiente nivel para el próximo paso (la simulación se realiza en orden *breadth-first*).

La prueba de que el estimador de *fixed-effort* es insesgado se expone en [12].

El algoritmo 2 resume una corrida del método *fixed-effort*.

Algorithm 2: Técnica *fixed-effort*

```
prev ← [s_0]          // ejecuciones que pasaron el nivel anterior
p̂ ← 1
for l ← 0 to (n − 1) do
    suc ← [ ]          // ejecuciones que pasan al siguiente nivel
    prev.shuffle()
    for j ← 1 to N_l do
        s ← prev[j mod prev.length()]        // selección balanceada
        while s ∉ B ∧ s.level() ≤ l do
            s ← s.next()    // simular hasta terminar o subir nivel
        if s ∉ B then
            suc.add(s)          // submimos de nivel de importancia
    if suc.empty() then
        p̂ ← 0
        break
    p̂ = p̂ · suc.size() / N_l              // actualizamos el estimador
    prev ← suc
```

Fixed success

Esta técnica, presentada en [1], es similar a *fixed-effort*, ya que la simulación se hace nivel por nivel. Pero en lugar de fijar la cantidad de ejecuciones que se realizan en cada nivel, se fija la cantidad de ejecuciones que queremos que crucen al nivel siguiente. Es decir, esta vez fijamos cada uno de los R_i, mientras que los N_i son aleatorios. Notar que es justo al revés que en *fixed-effort*.

Cuando simulamos en el k-ésimo nivel, simulamos hasta que lleguemos R_k veces al siguiente umbral de importancia. Entonces, tenemos que $N_{k-1} - R_k \sim NegBin(R_k, q)$ (variable binomial negativa) para alguna probabilidad q. La variable binomial negativa con parametros (r, q) es la cantidad de fallos que tenemos para un experimento que tiene éxito con probabilidad q, si lo realizamos hasta que hayamos tenido éxito exactamente r veces. En este caso, la probabilidad q depende de los estados de entrada al nivel $k-1$, pero usamos su estimador como estimador de p_k. El estimador que teníamos en el caso de *fixed-effort*: $\hat{p}_k = \frac{R_k}{N_{k-1}}$ no es el mejor que podríamos tener en el caso de *fixed-success*,

ya que no es un estimador insesgado para el parámetro q de la binomial negativa (a pesar de ser el de máxima verosimilitud). En su lugar, requerimos $R_k \geq 2$ y usamos el estimador $\hat{p}_k = \frac{R_k - 1}{N_{k-1} - 1}$. El motivo de la elección de este estimador es que si $r \geq 2$ y $V \sim NegBin(r, q)$, entonces $\frac{r-1}{r+V-1}$ es un estimador insesgado para q (y además es el único). Resumiendo, el estimador para p tiene la siguiente forma:

$$\hat{p} = \prod_{k=1}^{n} \hat{p}_k = \prod_{k=1}^{n} \frac{R_k - 1}{N_{k-1} - 1}$$

Así como en *fixed-effort*, tenemos diversas variantes dependiendo de cómo seleccionamos los estados de entrada al nivel $k-1$ en la k-ésima simulación. En *fixed-success*, tenemos esta técnica análoga a *asignación fija*: Antes de empezar permutamos aleatoriamente los R_{k-1} estados de entrada, elegimos una vez cada uno de ellos y repetimos (usando la misma permutación). Dicho de otra forma: para la i-ésima ejecución elegimos el (i *mod* R_{k-1})-ésimo estado. Análogamente al caso *fixed-effort*, esta versión tiene menor varianza que hacer muestreo uniforme para cada ejecución, por lo que será la que utilizaremos. La prueba de que el estimador de *fixed-success* es insesgado, para ambas versiones, se expone en [1].

A diferencia de *fixed-effort*, que busca controlar el esfuerzo computacional de la simulación, *fixed-success* busca de cierto modo controlar la imprecisión del estimador, ya que se termina de procesar un nivel cuando ocurre que cruzamos al siguiente nivel una cantidad de veces que consideramos suficiente. Por otro lado, el costo computacional de realizar la simulación *fixed-success* puede tener una gran varianza, sobre todo si hay estados de entrada al nivel k que es muy poco probable que lleguen al nivel $k+1$. Incluso, es posible que la simulación no termine nunca, en caso de que existan estados de entrada al nivel k desde los que no se pueda llegar al nivel $k+1$ sin que antes ocurra la condición de parada (notar que este tipo de situación no es problema para las otras técnicas de splitting). Lo ideal sería elegir la función de importancia y los umbrales de modo que este caso sea imposible, pero esto no siempre es sencillo. Por este motivo, es conveniente definir un valor máximo de ejecuciones por nivel, tal que si excedemos esa cantidad de ejecuciones y ninguna de las mismas pudo llegar al siguiente nivel de importancia, entonces paramos la ejecución y reportamos $\hat{p} = 0$. Esta heurística introduce cierto sesgo en la estimación, pero este sesgo será por lo general despreciable para un valor suficientemente grande de la cantidad máxima de ejecuciones.

En resumen, tenemos la ventaja de que controlamos mejor la imprecisión, pero ahora la elección de la función de importancia y los umbrales es crítica no sólo para la varianza del estimador, sino también para la varianza del tiempo de ejecución de cada simulación.

El algoritmo 3 resume una corrida del método *fixed-success*.

Funciones de importancia

Pasamos a hacer una breve descripción de algunas técnicas que se pueden usar para determinar la función de importancia, dado que serán las que utilicemos cuando experimentemos para comparar las diferentes técnicas de splitting.

Algorithm 3: Técnica *fixed-success*

```
prev ← [s_0]                    // ejecuciones que pasaron el nivel anterior
p̂ ← 1
for l ← 0 to (n − 1) do
    suc ← [ ]                   // ejecuciones que pasan al siguiente nivel
    prev.shuffle()
    j ← 0
    while suc.length() < R_{l+1} do
        s ← prev[j mod prev.length()]        // selección balanceada
        while s ∉ B ∧ s.level() ≤ l do
            s ← s.next()        // simular hasta terminar o subir nivel
        if s ∉ B then
            suc.add(s)          // submimos de nivel de importancia
        j ← j + 1
    p̂ = p̂ · (R_{l+1} − 1) / (j − 1)          // actualizamos el estimador
    prev ← suc
```

Ad-hoc. Una forma usual de definir una función de importancia es adecuarla al problema en cuestión. Es decir: definir a mano, mediante conocimiento del dominio, una función sobre los estados que parezca asignar un valor más alto a aquellos que estén "más cerca" de alcanzar el evento raro. Se puede experimentar para varias funciones y quedarse con aquella que tenga mejor rendimiento.

La desventaja de este tipo de técnica es que no es general (se debe elegir la función en base al modelo) y no siempre es fácil encontrar a mano una función que tenga buen rendimiento, sobre todo para modelos más complejos.

En nuestros experimentos no usaremos este tipo de función, dada la dificultad de aplicarlo correctamente, pero vale la pena mencionarlo por ser una de las formas más usuales, en especial cuando se quiere usar importance splitting en un dominio de aplicación real.

Monolítica. Una forma simple de derivar una función de importancia, presentada en [6], es generar el espacio de estados del proceso de Markov, y para cada estado calcular la distancia al estado raro más cercano, en términos de cantidad de transiciones hasta alcanzarlo. Esto se puede hacer en orden lineal en la cantidad total de transiciones del modelo, mediante un BFS desde los estados raros (es decir, inicialmente se los marca a todos como distancia 0 y se agregan todos a la *queue*). El BFS se realiza sobre un grafo que tiene una arista entre a y b si y sólo si hay una transición de b a a. Como queremos que los estados más importantes sean los que están más cerca, se define de esta forma $f(s) = -dist(s)$.

En el caso de IOSA, definimos la función sobre el estado del IOSA, y no sobre el estado del proceso de Markov asociado, ya que este espacio de estados es infinito al incluir información de los relojes.

La desventaja más importante de esta técnica es que requiere generar explícitamente el espacio de estados y las transiciones. Esto puede volver inaplicable esta estrategia en modelos con una gran cantidad de estados/transiciones. Este

caso suele darse cuando se tiene un modelo altamente composicional, ya que se suele dar el problema de la explosión de estados: La cantidad de estados crece exponencialmente en la cantidad de componentes.

Otra desventaja que tiene esta técnica es que sólo considera la cantidad de transiciones, y no tiene de ninguna manera en cuenta la probabilidad de que estas transiciones ocurran. Puede darse el caso de que se requieran pocas transiciones para llegar al evento raro, pero estas transiciones sean muy poco probables. De este modo se asignaría al estado un valor de importancia injustamente grande. Esto podría mejorarse ponderando de algún modo por la probabilidad de que la transición ocurra, pero no se ahondó en esto.

Composicional. La estrategia composicional, presentada en [7], busca resolver el problema de la explosión de estados que ocurre al aplicar una estrategia monolítica en modelos con varias componentes. La idea es calcular una función de importancia para cada componente por separado, y luego combinarlas de algún modo para tener la función del modelo completo. Entonces, se debe definir cómo se calcula la función de importancia dentro de un módulo y cómo se combinan los valores resultantes.

La función de importancia dentro de cada componente puede ser la monolítica, ya que las componentes suelen tener una cantidad manejable de estados antes de la composición paralela. En este caso, se debe definir cuáles serían los "estados raros" de cada componente, para luego aplicar el BFS a partir de los mismos. Una forma simple de definir estos estados raros es escribir la fórmula lógica que define el conjunto B en forma normal disjuntiva (o DNF, por sus siglas en inglés), es decir: disyunción de cláusulas conjuntivas. Luego de esto, marcamos como "raros" todos los estados que satisfagan algún literal de los presentes en la fórmula, ignorando los literales que correspondan a variables que no pertenecen a la componente.

Una vez que tenemos los valores de importancia para cada componente, definimos el valor de importancia de un estado del modelo composicional como la suma de los valores de importancia de los estados correspondientes a cada componente.

Se puede tener en cuenta la estructura de la fórmula en DNF, usando dos operadores distintos para combinar el $\vee$ y el $\wedge$. Usualmente se usan operadores que forman un semi-anillo (por ejemplo $(max, +)$, $(+, *)$), donde el $\vee$ se combina con la suma (del semi-anillo) y el $\wedge$ con el producto. Mencionamos este enfoque a modo de complemento, pero nos inclinamos por simplemente sumar lo valores de importancia para cada componente, ya que es más simple de usar en la herramienta en la que nos basamos. Además, en los experimentos que haremos, las fórmulas correspondientes a las probabilidades de interés son relativamente simples, por lo que no habría una diferencia significativa entre ambos enfoques.

Capítulo 5

Implementación

En este capítulo hacemos una breve descripción de la herramienta sobre la cuál se trabajó, qué se modificó y algunas dificultades que surgieron.

FIG

La herramienta FIG (Finite Improbability Generator) fue introducida en [8] como parte de la investigación sobre técnicas de derivación automática de funciones de importancia. Toma modelos en el lenguaje de descripción de IOSAs (descriptos en el Capítulo 3), y permite estimar propiedades transient y steady state (descriptas en el Capítulo 4).

La herramienta implementa dos motores de simulación: *NoSplit* (Montecarlo estándar) y *Restart*. En ambos casos, calcula un intervalo de confianza para la propiedad en cuestión y permite parar las simulaciones cuando se alcanzó cierto tiempo límite de corrida, o cuando el intervalo de confianza para cierto nivel de confianza tiene amplitud relativa menor a cierto valor. El intervalo de confianza se construye ejecutando muchas instancias de la simulación, y aproximando la media de todos los estimadores obtenidos como una distribución normal, por Teorema Central del Límite.

Se pueden usar funciones de importancia ad-hoc, es decir, definidas por el usuario, como también funciones derivadas automáticamente: monolítica y composicional, descriptas brevemente en el capítulo anterior. La cantidad de niveles de importancia y los umbrales correspondientes se pueden fijar a mano, pero también se implementa una técnica automática para fijar los niveles de importancia. Esta técnica se conoce como *Sequential Monte Carlo* y se presenta en [9]. Nos inclinamos en usar los thresholds derivados automáticamente, dada la dificultad de fijar a mano niveles de importancia que resulten en buenos estimadores.

Cambios introducidos

Se implementaron dos nuevos motores: *FixedEffort* y *FixedSuccess*. También se implementaron cambios en la interfaz para poder seleccionar estas técnicas

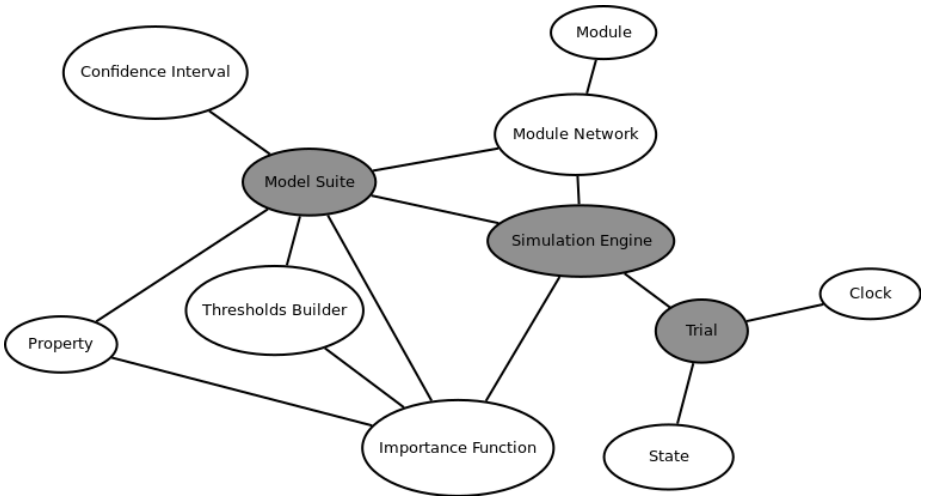

Figura 5.1: Diseño simplificado de FIG

de simulación, y configurar el esfuerzo (cantidad de ejecuciones por nivel) para *FixedEffort* y la cantidad de ejecuciones exitosas por nivel para *FixedSuccess*.

Se incluye como parte del trabajo el código de la herramienta con los motores nuevos, más un README con instrucciones de cómo utilizarlos y una lista más detallada de los cambios realizados en el código.

Se muestra el la Figura 5.1 un esquema simplificado del diseño de FIG, que muestra los módulos más importantes y las comunicaciones más relevantes entre los mismos. Están coloreadas las componentes en donde se introdujeron las mayores modificaciones.

El módulo sobre el que se trabajó más es *Simulation Engine*. Los motores que ya estaban implementados no se modificaron, pero se agregaron los motores nuevos que implementan *fixed-effort* y *fixed-success*. El motor de simulación recibe la descripción del modelo IOSA (*Module Network*), y la función de importancia, y realiza una única simulación usando la técnica correspondiente. Todos los motores de simulación hacen uso de la estructura *Trial*, que representa una instancia de ejecución del modelo, o sea, guarda el estado del IOSA y los tiempos de expiración de los relojes.

El módulo central del diseño es *Model Suite*, que se encarga de coordinar los pasos previos a la simulación, invocando a otros módulos: construcción de la función de importancia, determinación de los umbrales, invocación del motor de simulación y control de las condiciones de finalización de la simulación. Sobre este módulo se introdujeron algunos cambios para poder incorporar los dos nuevos motores de simulación, permitiendo configurar cuál se desea usar.

Problemas que surgieron

El problema más relevante que surgió durante la implementación, que se manifestó cuando se realizaron los experimentos, tiene que ver con la limitación que tiene *fixed-success* respecto a la función de importancia y los umbrales:

En el caso ideal, no debe poder ocurrir que una simulación que suba a un determinado nivel de importancia tenga probabilidad 0 de llegar al siguiente nivel. Trae problemas este caso, porque se puede dar que todas las ejecuciones que tenemos en un nivel tengan probabilidad 0 de llegar al siguiente, en cuyo caso la simulación no termina nunca. Como ya mencionamos, en la práctica (y en nuestra implementación) se puede aligerar el problema determinando un máximo de ejecuciones por nivel, que una vez que se excede esta cantidad se termina y se estima que la probabilidad es 0. Esta solución no es la ideal, ya que introduce un sesgo en el estimador y se desperdicia mucho tiempo de cómputo cuando se ingresa a uno de estos casos (la elección del máximo de ejecuciones por nivel es un *tradeoff* entre estos dos problemas).

Este problema se daba en nuestros experimentos, porque la forma que tiene FIG de representar los estados es mediante el estado del IOSA y los tiempos de finalización de los relojes activos. Esta información ya es suficiente para saber cuál es la próxima transición a ejecutarse y, por consiguiente, cuál será el siguiente estado del IOSA. Dado que lo que se usa para la función de importancia es sólamente el estado del IOSA, sin tener en cuenta el estado de los relojes, era muy común la situación de subir un nivel de importancia, pero que quede determinado que en el próximo paso se baja de nivel sí o sí.

Dado que todas las variables que usamos para los relojes en nuestros experimentos son exponenciales, en nuestro caso no es necesario que los estados incluyan los valores de los relojes, dado que los que están activos pueden reiniciarse después de cada transición y la simulación se comporta de la misma forma.

Luego, se realizó una modificación para que los relojes que son variables exponenciales se reinicien después de cada transición. Ésto mejoró considerablemente el rendimiento de la estrategia *fixed-success* en nuestros experimentos.

Este cambio produjo un impacto positivo en el rendimiento de, no sólo la técnica *fixed-success*, sino también las demás técnicas de splitting. Para estas técnicas, el problema de las ejecuciones que bajan siempre de nivel en el próximo paso, si bien no causaba que el proceso se detuviera como en *fixed-success*, provocaba que el estimador tuviera mayor varianza.

En el caso de Monte Carlo estándar, lo único que hace esta modificación es hacer más muestreos de variables exponenciales, y por lo tanto, hacer menos eficiente en tiempo cada paso de simulación. Por consiguiente, hacemos que al tratarse de una simulación Monte Carlo no se haga el remuestreo de los relojes exponenciales.

Si bien este cambio resuelve el problema para algunos de nuestros experimentos, no aplica al caso general de los modelos IOSA, en los que los relojes pueden tener asociados distribuciones arbitrarias. Una solución más general consistiría en considerar los tiempos de finalización de los relojes como parte del estado, a la hora de evaluar la función de importancia, quedando como motivación de un posible trabajo futuro.

Surgieron otras complicaciones, relacionadas a ciertas decisiones de diseño de la herramienta, las cuales no mencionamos dado que esto llevaría a hablar en demasiado detalle de lo que es el código en sí.

Capítulo 6

Experimentos y Resultados

En este capítulo presentaremos los casos de estudio que usamos para comparar las técnicas *fixed-effort* y *fixed-success* respecto a la técnica previamente implementada: RESTART. Primero explicaremos cada caso de estudio por separado y luego presentamos los resultados en términos de la precisión del estimador obtenido para un tiempo fijo de ejecución.

Casos de estudio

Tomamos tres casos de estudio. Los dos primeros son sistemas de colas y están basados en experimentos de [5]. El tercero representa un sistema de transporte de petróleo, basado en un experimento de [8]. En los dos primeros, los modelos son equivalentes a una CTMC, por lo que se puede validar que la estimación obtenida sea correcta contra el valor obtenido analíticamente por la herramienta PRISM [17]. Los códigos de los modelos correspondientes, tanto en lenguaje IOSA como en PRISM se presentan junto al código, como parte del trabajo.

Colas tandem

El primer experimento consta de dos colas (queues) en tandem, es decir, conectadas de modo que los eventos que ingresan a la primera, luego de ser procesados son agregados a la segunda, y cuando la segunda cola termina de procesarlo, este desaparece. Las dos colas procesan sus eventos independientemente una de la otra. Consideraremos que el tiempo transcurrido hasta que llega un evento a la primera cola, y los tiempos que demora cada cola no vacía en procesar eventos son variables exponenciales. El estado del sistema se puede representar por dos variables enteras q_1, q_2, que representan la cantidad de eventos que contiene cada una de las colas.

El modelo tiene los siguientes parámetros:

- El tamaño de cada una de las colas (c_1, c_2).
- El estado inicial del sistema (es decir, cuantos elementos contiene cada cola al inicio) (i_1, i_2).

- La tasa de llegada de eventos a la primera cola (λ) y las tasas de procesamiento de eventos de ambas (μ_1, μ_2). Estos son los parámetros de las variables exponenciales respectivas.

En un modelo como este, la propiedad de interés puede ser la probabilidad de que cierta cola se llene. En nuestro caso, tomaremos como evento de interés cuando la segunda cola se llena.

En cuanto a la condición de reinicio, una elección razonable puede ser que la segunda cola se vacíe. Entonces, nos interesa calcular la siguiente propiedad PCTL:

$$P(q_2 > 0 \ U \ q_2 = c_2)$$

Obviamente, para que esta elección tenga sentido, se requiere que $i_2 > 0$.

Usaremos los parámetros $(i_1, i_2) = (0, 1)$, $(\lambda, \mu_1, \mu_2) = (3, 2, 6)$ y variamos los tamaños de las colas (probamos $8, 12, 16, 20$) para comparar las técnicas para diferentes valores de la probabilidad a estimar. Es importante remarcar el hecho de que la tasa de vaciado de la segunda cola es significantemente mayor que el de la primera. Por este motivo, el cuello de botella del sistema se da en la primera cola, siendo así muy raro que se llegue a saturar la segunda.

Colas con rupturas

Este experimento, como el primero, puede verse como una CTMC, dado que modelamos todos los tiempos de transición como variables exponenciales.

Consiste en una cola de longitud K que recibe eventos de 10 fuentes de 2 tipos distintos (5 de cada uno). Tanto la cola como las fuentes pueden estar en dos estados: activa o inactiva. El estado de cada componente es independiente del estado de las demás. Mientras están en estado inactivo, las fuentes no generan eventos, y la cola no los procesa. Inicialmente todas las componentes están en estado inactivo, excepto una fuente de tipo 2 que se encuentra activa.

La cola procesa eventos con una tasa μ (mientras está activa), pasa de activa a inactiva con tasa γ y de inactiva a inactiva con tasa δ. Las fuentes de tipo i generan un evento con tasa λ_i (mientras están activas), pasan de activas a inactivas con tasa β_i y pasan de inactivas a activas con tasa α_i.

Como en el caso de las dos colas tandem, nos interesa la probabilidad de que la cola se llene antes de llegar a vaciarse, dado que empieza con un elemento.

Elegimos los parámetros $(\mu, \gamma, \delta) = (100, 3, 4)$, $(\lambda_1, \beta_1, \alpha_1) = (3, 2, 3)$, $(\lambda_2, \beta_2, \alpha_2) = (6, 4, 1)$. Similarmente al caso anterior, probamos varios valores para K: 40, 80, 120, 160.

Sistema de transporte de petróleo

Este experimento, basado en un experimento de [8], es un sistema un poco más complejo que los sistema de colas presentados previamente, donde además las transiciones pueden ocurrir en intervalos de tiempos con distribuciones distintas la exponencial, con lo cuál ilustramos la mayor expresividad de IOSA respecto a las CTMC.

El experimento estudia un sistema de transporte de petróleo, que consiste en una cañería donde están instaladas n bombas de forma equidistante. Cada bomba puede estar en 3 estados: funcional, averiada o siendo reparada. Inicialmente todas se encuentran en estado funcional. El sistema falla en transportar

el petróleo cuando hay un intervalo de k bombas consecutivas que no están en estado funcional. El sistema contiene también un reparador, que cuando está libre y hay alguna bomba averiada, empieza a repararla. El reparador sólo puede reparar una bomba a la vez, y cuando está libre y hay varias bombas averiadas elige la de menor índice (esto puede ocurrir cuando varias bombas se rompen mientras el reparador está reparando otra bomba).

En [8] se considera la proporción de tiempo que el sistema se encuentra en un estado fallido (propiedad del estado estacionario). En nuestro caso, tomamos como probabilidad de interés la probabilidad de que el sistema falle, antes de volver de nuevo a un estado donde todas las bombas se encuentran funcionales.

Los parámetros de este modelo son n, k y las distribuciones que representan el tiempo que demora cada máquina en fallar y en ser reparada. Tomamos el tiempo de falla como una exponencial con $\lambda = 10^{-3}$, el tiempo de reparación como una log-normal con media 1,21 y varianza 0,8 (tenemos que el tiempo promedio de falla es mucho mayor al tiempo medio de reparación, contribuyendo a la rareza de la propiedad a estimar). Tomamos $k = 4$ y variamos el valor de n: $20, 40, 60$ para experimentar varios escenarios.

Notemos que a diferencia de los casos de estudio anteriores, en este la cantidad de estados es inmanejable, ya que es exponencial en n. Por este motivo será imposible aplicar automáticamente la técnica monolítica para este experimento.

Resultados

Para cada uno de los casos de estudio, se utilizaron las funciones de importancia monolítica y composicional, con los umbrales de importancia determinados automáticamente por FIG. Para RESTART, se usa un factor de splitting de 4, que es el que mejor rendimiento mostró en [5], para *fixed-effort* se usa un esfuerzo de 500 por nivel, y para *fixed-success* un factor de 20 ejecuciones exitosas por nivel. El valor real de cada probabilidad fue calculado para los modelos de sistemas de colas mediante la herramienta PRISM, mientras que para el caso del sistema de transporte de petróleo tomamos como estimador de la probabilidad la media de los estimadores para todas las técnicas que tuvieron éxito. En todos los experimentos de sitemas de colas, el valor real de la probabilidad cayó dentro del intervalo de confianza obtenido.

Los resultados se muestran en el Cuadro 6.1. Para cada modelo y cada técnica estimamos un intervalo de confianza del 95 %. El valor que se muestra en el cuadro es la relación entre el tamaño del intervalo de confianza obtenido y el valor de la probabilidad objetivo, expresada como porcentaje. Es decir, un menor valor refleja un menor error relativo de los estimadores, para un mismo tiempo de ejecución. No se muestra ningún valor en los casos en los que no se pudo estimar la probabilidad.

Se observa una mejora muy significativa de todos los métodos respecto de Montecarlo estándar, que para eventos de probabilidad muy baja falla en estimarla, sin siquiera llegar al evento raro una vez.

Tenemos que las técnicas *fidex-effort* y *fixed-success* funcionan mejor que Restart para ambos sistemas de colas. Esto puede explicarse debido a que estas técnicas controlan de mejor forma el esfuerzo que se hace por nivel, mientras que RESTART puede resultar muy costoso en los niveles altos y es más susceptible a la elección del factor de splitting y los umbrales de importancia. Además, las

Modelo	Parámetro	p	MC	Restart		F. Effort		F. Success	
				mono	comp	mono	comp	mono	comp
Tandem	8	5.6e-06	4	3	3	2	2	2	2
	12	1.86e-08	65	12	15	3	5	4	12
	16	7.16e-11	-	48	39	5	16	6	6
	20	2.99e-13	-	29	86	8	53	9	9
Breakdown	40	4.59e-04	2	2	2	2	3	2	2
	80	3.72e-07	82	6	5	6	6	6	3
	120	3.01e-10	-	14	14	8	10	6	6
	160	2.45e-13	-	103	112	14	14	9	9
Oil pipeline	20	2.56e-06	-	-	57	-	143	-	-
	40	3.07e-06	-	-	155	-	76	-	-
	60	7.16e-06	-	-	131	-	85	-	-

Cuadro 6.1: Error relativo para cada experimento

técnicas de *fixed-effort* y *fixed-success* están especializadas para hacer análisis de propiedades transitorias, mientras que RESTART se usa sobre todo para hacer análisis del estado estacionario, debido a que una misma ejecución puede dividirse muchas veces al cruzar el mismo nivel, por lo que se pueden hacer ejecuciones largas donde siga ocurriendo la división al acercarse al estado raro.

Fixed-effort se comportó mejor en el experimento de las colas tandem, mientras que en el caso de las colas con rupturas, *fixed-success* tuvo el mejor rendimiento, aunque en ambos casos no hay una diferencia muy significativa entre ambas técnicas.

Vemos que en general el cambio de función de importancia de monolítica a composicional tiene un impacto negativo en el rendimiento, y este impacto es considerablemente menor en el caso de *fixed-success*. Vale la pena aclarar que este impacto es de esperar, dado que la función composicional puede verse como una simplificación de la función monolítica, que tiene la ventaja de que evita el problema de la explosión de estados, cuestión que no es de relevancia en nuestros experimentos dado que los espacios de estados son relativamente pequeños.

En el caso del sistema de transporte de petróleo, para todos los parámetros tanto Montecarlo como *fixed-success* fallaron en estimar la probabilidad. En el caso de *fixed-success* se dio el problema que reportamos en el capítulo 5, ya que al involucrar tiempos muestreados de variables no exponenciales, era muy común el caso de que resulte imposible llegar al nivel siguiente a partir de los estados disponibles en el nivel actual. En este experimento no se pudo usar la función de importancia monolítica debido a la gran cantidad de estados de los modelos. En los casos en los que sí se pudo lograr un estimador de la probabilidad (con *fixed-effort* y RESTART con función composicional), se obtuvieron estimadores de gran error, con relación a los demás experimentos, a pesar de que la probabilidad del evento raro es relativamente alta para todos los parámetros. Esto puede deberse a que las ejecuciones que llevan al evento raro o a la condición de parada tienen mayor número de transiciones en promedio, por lo que cada simulación es más costosa. Esto explica también la incapacidad de Montecarlo para lograr la ocurrencia del evento raro, a pesar de que en los sistemas de colas pudo estimar probabilidades aún más bajas.

Capítulo 7

Conclusiones

En este trabajo, implementamos las técnicas de *Importance splitting fixed-effort* y *fixed-success* en una herramienta de *splitting* basada en modelos IOSA, que es más expresivo que otros tipos de modelos comúnmente usados como las CTMC. Esta herramienta implementaba únicamente la técnica RESTART, la cuál es muy útil para analizar propiedades del estado estacionario, pero al analizar propiedades transitorias es superada en muchos casos por las técnicas implementadas en este trabajo. Esta suposición fue corroborada mediante la realización de una serie de experimentos, comparando las tres técnicas para varios casos de estudio, con diferentes probabilidades para la ocurrencia del evento raro, y varias funciones de importancia. A su vez, todas las técnicas de *splitting* resultaron en un rendimiento muchísimo mayor al de la simulación por Monte Carlo, ilustrando la utilidad de estas técnicas para el análisis de eventos raros.

Además, se mejoró el rendimiento para las tres técnicas de *splitting* en IOSA, mediante el reinicio de los relojes exponenciales antes de cada transición. La causa de esta mejora fue el hecho de que las funciones de importancia que se manejaron no tenían en cuenta el valor de los relojes a la hora de calcular la importancia. Se desprende entonces, como posible trabajo futuro, la investigación en técnicas de derivación de funciones de importancia para IOSA, que tomen en consideración los relojes como parte del estado. Este problema tenía más impacto en la técnica *fixed-success*, que es muy susceptible a casos en los que no se puede subir de nivel de importancia desde un estado, por lo que se puede investigar en variaciones de *fixed-success* que alivien esto, o en formas de detectar tal situación.

Otro posible trabajo futuro podría ser la implementación de otra técnica de *splitting*, adecuada para el análisis del estado estacionario, para que exista una alternativa a RESTART en esos casos.

Bibliografía

[1] Michael Amrein and Hans R Künsch. A variant of importance splitting for rare event estimation: Fixed number of successes. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21(2):13, 2011.

[2] Sigrún Andradóttir, Daniel P Heyman, and Teunis J Ott. On the choice of alternative measures in importance sampling with markov chains. *Operations research*, 43(3):509–519, 1995.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[5] Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. Better automated importance splitting for transient rare events. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 42–58, Cham, 2017. Springer International Publishing.

[6] Carlos E Budde, Pedro R D'Argenio, and Holger Hermanns. Rare event simulation with fully automated importance splitting. In *European Workshop on Performance Engineering*, pages 275–290. Springer, 2015.

[7] Carlos E Budde, Pedro R D'Argenio, and Raúl E Monti. Compositional construction of importance functions in fully. *European Transactions on Telecommunications*, 13(4):363–371, 2002.

[8] CE Budde. *Automation of importance splitting techniques for rare event simulation*. PhD thesis, Ph. D. thesis, Universidad Nacional de Córdoba, Córdoba, Argentina, 2017.

[9] Frédéric Cérou, Pierre Del Moral, Teddy Furon, and Arnaud Guyader. Sequential monte carlo for rare event estimation. *Statistics and computing*, 22(3):795–808, 2012.

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[11] Pedro R D'Argenio, Matias David Lee, and Raúl E Monti. Input/output stochastic automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 53–68. Springer, 2016.

[12] Marnix Joseph Johann Garvels. The splitting method in rare event simulation. 2000.

[13] Peter W Glynn and Donald L Iglehart. Importance sampling for stochastic simulations. *Management Science*, 35(11):1367–1392, 1989.

[14] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[15] Herman Kahn and Theodore E Harris. Estimation of particle transmission by random sampling. *National Bureau of Standards applied mathematics series*, 12:27–30, 1951.

[16] M. Kwiatkowska, G. Norman, and D. Parker. Advances and challenges of probabilistic model checking. In *Proc. 48th Annual Allerton Conference on Communication, Control and Computing*, pages 1691–1698. IEEE Press, 2010.

[17] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[18] Pierre L'Ecuyer, Valérie Demers, and Bruno Tuffin. Splitting for rare-event simulation. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 137–148. Winter Simulation Conference, 2006.

[19] Pierre L'Ecuyer, Valérie Demers, and Bruno Tuffin. Rare events, splitting, and quasi-monte carlo. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 17(2):9, 2007.

[20] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *IEEE computer*, 26(7):18–41, 1993.

[21] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[22] Gerardo Rubino and Bruno Tuffin. *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009.

[23] Brian J Sauser, Richard R Reilly, and Aaron J Shenhar. Why projects fail? how contingency theory can provide new insights–a comparative analysis of nasa's mars climate orbiter loss. *International Journal of Project Management*, 27(7):665–679, 2009.

[24] Antti Valmari. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification*, pages 156–165. Springer, 1990.

[25] Manuel Villen-Altamirano and Jose Villen-Altamirano. Restart: A method for accelerating rare event simulations. *Analysis*, 3(3), 1991.

[26] Manuel Villén-Altamirano and Jose Villen-Altamirano. Analysis of restart simulation: Theoretical basis and sensitivity study. *European Transactions on Telecommunications*, 13(4):373–385, 2002.

[27] Manuel Villén-Altamirano and José Villén-Altamirano. On the efficiency of restart for multidimensional state systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 16(3):251–279, 2006.

[28] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[29] Håkan LS Younes and Reid G Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *International Conference on Computer Aided Verification*, pages 223–235. Springer, 2002.