Herramienta gráfica para guiar a investigadores a realizar experimentos bajo radiación memorias

Autor:

Jesús Alguacil González

Director:

Juan Antonio Clemente Barreira

Trabajo de Fin de Grado – Grado en Ingeniería Informática

Facultad de informática – Curso 2020/2021
Universidad Complutense de Madrid

Graphic tool to guide researchers to perform experiments with memories under the effects of radiation

Author:

Jesús Alguacil González

Director:

Juan Antonio Clemente Barreira

Trabajo de Fin de Grado – Grado en Ingeniería Informática

Facultad de informática – Curso 2020/2021

Universidad Complutense de Madrid

Agradecimientos

A mi familia y a mis amigos que siempre dan su apoyo para que dé lo mejor de mí mismo.

Índice

RESUMEN **5**
Palabras clave 5
SUMMARY **6**
Keywords 6
1. INTRODUCCIÓN **7**
1.1 Efectos de la radiación en los dispositivos 7
1.2 Fuentes de radiación 8
1.3 Métodos de mitigación frente a los efectos de la radiación 9
1.3.1 Mitigación por tecnología 9
1.3.2 Mitigación por diseño 11
1.3.3 Mitigación por redundancia 12
1.4 Motivación de este Trabajo de Fin de Grado 13
2. ESTIMACIÓN DEL NÚMERO DE FALSOS MCUS **14**
2.1 La "paradoja del cumpleaños" 14
2.2 Ecuaciones de detección en función de la distancia entre celdas 15
3. DESARROLLO DE LA APLICACIÓN **18**
3.1 Plan de trabajo 18
3.2 Plataforma de desarrollo 18
3.3 JavaFX 18
3.4 Configuración del proyecto 21
3.5 Estructura del proyecto 23
3.6 Código 24
3.6.1 Formulario 24
3.6.2 Control de escenas 27
3.5.3 Cálculo de errores 28
3.7 Portabilidad de la aplicación 31
4. RESULTADOS **32**
4.1 Estudio de las ecuaciones de probabilidad 32
4.2 Estudio de experimentos previos con memorias en aceleradores de partículas 36
4.2.1 Primer experimento 36
4.2.2 Segundo experimento 38
4.2.3 Tercer experimento 40
5. CONCLUSIONES DE ESTE TRABAJO DE FIN DE GRADO **42**
5. CONCLUSIONS OF THIS BACHELOR'S THESIS **43**
BIBLIOGRAFÍA **44**

Resumen

Según avanza la tecnología de fabricación de memorias, éstas se vuelven cada vez más sensibles a los efectos de la radiación natural del entorno que las rodea. Diversas fuentes de radiación (protones, neutrones, partículas alfa…) son capaces de inducir cambios en la lógica combinacional de estos dispositivos o incluso afectar al contenido de las celdas de memoria, corrompiendo la información que almacenan. Esto es debido a que las partículas incidentes, al impactar con las celdas de memoria, depositan la suficiente energía como para inducir cambios en su contenido (de 0 a 1 o viceversa).

Aunque hay técnicas a la hora de fabricación para reducir la aparición de estos efectos, hasta ahora no es posible evitarlos por completo. Por ello, los equipos de investigación deben realizar pruebas simulando estos efectos, y una de las técnicas más populares y realistas de conseguirlo es exponer los dispositivos a la radiación en un acelerador de partículas. Cuando una partícula impacta con el dispositivo, ésta puede causar un evento simple, que consiste en el cambio de un único bit de información; o un evento múltiple que consiste en el cambio de varios bits de información físicamente cercanos. Cuando se analizan los resultados obtenidos tras la exposición de un dispositivo a la radiación, una problemática que surge es la probabilidad de aparición de "falsos eventos múltiples", que se manifiestan como eventos múltiples, pero realmente se produjeron por acumulación de eventos simples que casualmente afectaron a celdas vecinas o cercanas. La probabilidad de aparición de dichos falsos eventos múltiples es mayor cuanto mayor sea el tiempo de exposición del dispositivo frente a la radiación y el flujo de partículas incidente.

De este modo, el objetivo de este trabajo es crear una herramienta que ayude a los investigadores a visualizar el tiempo óptimo de exposición donde se obtendrán el máximo de resultados minimizando, o al menos controlando, la probabilidad de aparición de falsos eventos múltiples en el experimento.

Palabras clave

Radiación, Memorias, Interfaz gráfica de usuario (GUI), Investigación, Optimización, Hardware.

Summary

As memory fabrication technology scales down, memories have become more and more sensitive to the effects of natural radiation of the surrounding environment. Different sources of radiation (protons, neutrons, alpha particles…) are able to induce changes in the combinational logic of these devices or even capable of affecting the contents of the memory cells, corrupting the information that they store. This phenomenon occurs when charged particles impact with memory cells and they deposit enough energy to flip their contents (from 0 to 1 or vice versa).

Even though there exist techniques that mitigate the appearance of these effects, this is still a not fully resolved problem that needs to be addressed. Thus, research teams must simulate these effects, and one of the most popular and realistic techniques of attaining this objective is to expose devices against the radiation by using a particle accelerator. When a particle hits the device, it may cause a simple event, which consists in the alteration of one unique bit of information, or a multiple event, which consists in the alteration of several bits of information that are allocated physically close. When the results of exposing the device to the radiation are analyzed, an issue that arises is the appearance of “false multiple events” that manifest as multiple events but were, in fact, produced by accumulation of various simple events that, coincidentally, affected neighbor cells. The greater the time of exposure to the radiation and to incidental flux of particles, the greater is the probability of the aforementioned events to appear.

Thus, the goal of this project is to create a tool that would help the researcher visualize the optimal time in which he/she will get the maximum number of results while minimizing, or at least, controlling the probability of appearance of false multiple events in the experiment.

Keywords

Radiation, Memories, Graphical User Interface (GUI), Research, Optimization, Hardware.

1. Introducción

1.1 Efectos de la radiación en los dispositivos

Los dispositivos electrónicos de nueva generación son cada vez más sensibles a los efectos de la radiación ambiental que les rodea. Según se expone en [1], estos efectos varían en función de la fuente de la radiación. Se pueden distinguir dos clases de fallos dependiendo del fenómeno que afecte a la memoria.

El primero consiste en la creación de defectos Frenkel en la estructura cristalina del silicio del que están compuestos estos chips. Este fenómeno también es conocido como daño por desplazamiento o “displacement damage” [1].

En otro tipo de situaciones, la radiación Gamma, los rayos X y otras fuentes de radiación son capaces de generar una carga eléctrica dentro del circuito integrado de la memoria, llegando a veces a afectar a los transistores. Esta carga dura poco pero su intensidad puede provocar cambios en el estado eléctrico y alterar su contenido.

En este último escenario, se producen los conocidos como Single Event Effects (SEEs). Estos eventos se pueden clasificar en diferentes categorías (Figura 1), pero los principales que van a ser estudiados a lo largo del trabajo serán los Single Event Upsets (SEUs). Los SEUs consisten en la modificación de la información almacenada en una o varias celdas de la memoria, lo que también se conoce con el nombre de bitflips. Los SEUs se clasifican en función de su multiplicidad entre Single Bit Upsets (SBUs) y Multiple Cell Upsets (MCUs). Otros tipos de SEEs (Single Event Transients (SETs), Single Event Functional Interrupts (SEFIs) y errores permanentes) también están incluidos en [1]. En la Figura 1 se proporciona la clasificación de estos eventos y se resalta cuáles son de interés en este trabajo.

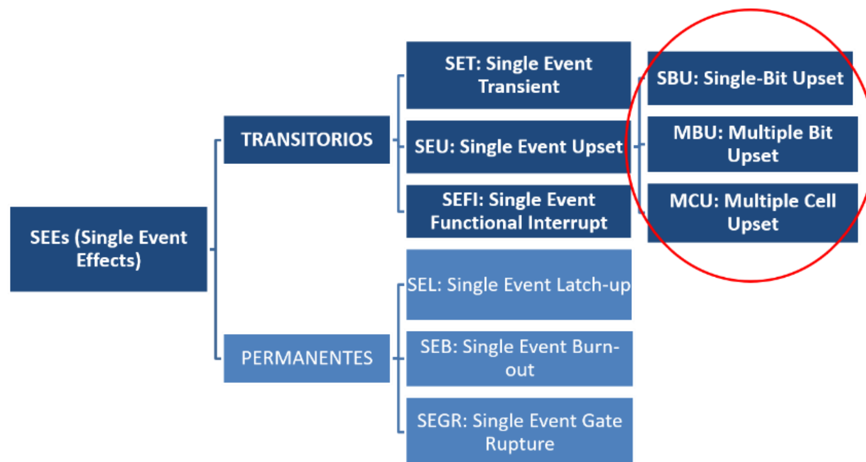

Figura 1. Clasificación de los diferentes tipos de SEEs

Los SBUs consisten en un evento aislado que solo afecta a un bit de la memoria. Alternativamente, se encuentran los Multiple Bit Upsets (MBUs) y los ya mencionados MCUs que, como sus nombres indican, son eventos que afectan a varios bits o varias celdas de la memoria. Al ser provocados por la misma partícula, estos bits o celdas se encuentran muy próximas y, en el caso de los MBUs, afectan a varios bits de una palabra, dificultando de esta forma la recuperación de su contenido original [1]. La diferencia principal de los MBUs con los MCUs es que, en el primer caso, los bits

afectados por la radiación pertenecen a la misma palabra, mientras que en los MCUs son de palabras diferentes.

Para evitar la aparición de MBUs y que aparezcan MCUs en su lugar, los fabricantes implementan la técnica de “bit interleaving” o “entrelazado de memoria” [2], [3] [4]. Esta técnica consiste en intercalar bits pertenecientes a diferentes palabras consiguiendo que, cuando dos o más bits vecinos se vean afectados por un evento múltiple, éste no afecte a varios bits de una palabra sino a un bit de varias palabras, permitiendo que los “Error Correcting Codes (ECC)” estándar puedan recuperar los datos que han sido modificados (Figura 2) [4] [5].

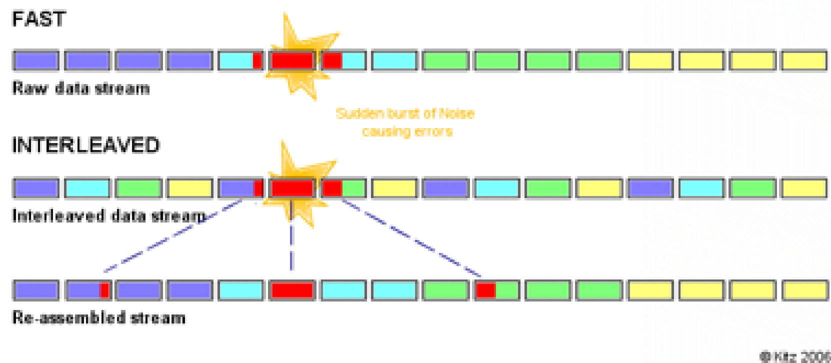

Figura 2. Funcionamiento del "bit interleaving".

En un entorno donde existe radiación natural, y en particular, en un acelerador de partículas, donde es esperable observar una gran cantidad de eventos para obtener resultados estadísticamente válidos, ocurren de forma frecuente los conocidos como falsos MCUs. Este fenómeno se manifiesta cuando aparentemente se observa un MCU, pero que en realidad fueron dos o más SBUs diferentes los que modificaron celdas vecinas o muy próximas, en vez de ser sólo una partícula la que provocase el cambio múltiple. A la hora de estudiar los eventos múltiples, la aparición de estos eventos falsos dificulta enormemente el avance de la investigación ya que provocan que los resultados obtenidos no sean absolutamente fiables.

1.2 Fuentes de radiación

Existen dos principales fuentes de radiación que provocan la aparición de los SEEs. Una fuente son las impurezas radioactivas en los componentes de los dispositivos electrónicos y la otra son los rayos cósmicos provenientes del espacio exterior [6].

El primer tipo de radiación se genera cuando los materiales que se usan para construir los componentes de la memoria no son debidamente purificados a la hora de su extracción. Elementos como el estaño y el plomo suelen contener trazas de materiales radioactivos como el uranio y el torio que, si no son purificados apropiadamente, pueden generar la radiación necesaria como para desencadenar un SEE [7].

Por otro lado, están los rayos cósmicos que proceden del espacio exterior y que interfieren con los equipos electrónicos que se usan en las naves espaciales [8] y en los satélites [9], y en algunas ocasiones en situaciones cotidianas [10]. En este último caso se han observado varios casos en los que los rayos han provocado alteraciones en los chips de memoria. A pesar de que muchas de las partículas de estos rayos, mayoritariamente protones, son repelidas por el campo magnético terrestre, algunos

llegan a atravesarlo y provocar multitud de eventos. Por otro lado, los protones también interactúan con los átomos de la atmósfera, provocando “duchas” de partículas secundarias (Figura 3), entre ellas neutrones de varias energías [11]. Éstos, a pesar de no poseer carga eléctrica, son capaces de interactuar con elementos como el boro [6] el cual, desafortunadamente, tiene una gran presencia en la electrónica en componentes como la capa BPSG (“Borophosphosilicate Glass”).

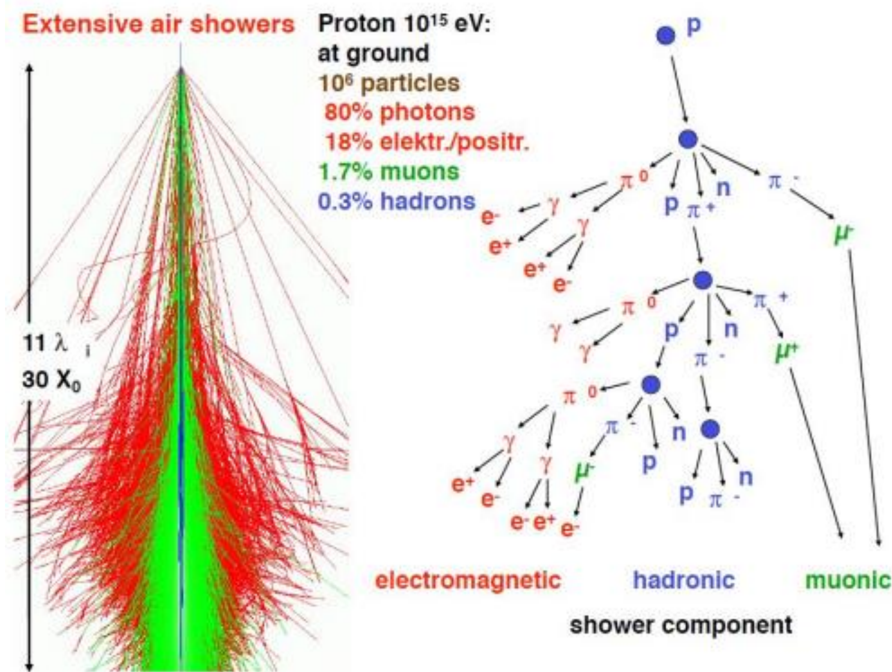

Figura 3. Extensive Air Showers. Imagen extraída de [1]

1.3 Métodos de mitigación frente a los efectos de la radiación

Una vez conocidos qué son y cómo se pueden generar los SEEs, en la literatura se han propuesto y desarrollado métodos para la mitigación de sus efectos. Ninguno de los métodos que se van a exponer a continuación son infalibles ya que, dependiendo del tipo de electrónica con la que se esté trabajando, pueden tener una eficacia diferente. Además, cabe mencionar que factores como la fuente de alimentación, la temperatura o la frecuencia del reloj del dispositivo pueden afectar a la sensibilidad de los dispositivos ante los SEEs [6].

1.3.1 Mitigación por tecnología

El primero de los métodos es la mitigación usando la tecnología, cuyo objetivo es el de mejorar la tecnología de fabricación de chips para que aumente su tolerancia a la aparición de SEEs. Con este propósito se han se han producido una serie de avances tecnológicos entre los que destacan los siguientes:

---

1 https://indico.cern.ch/event/763013/contributions/3358878/attachments/1817838/2971899/GroupD_CosmicRays.pdf

1. **Eliminación de la capa BPSG**: La capa BPSG, utilizada en tecnologías anteriores a 180-nm, es un componente que formaba parte de gran variedad de memorias, estaba formada por boro y su uso en los años 90 se popularizó debido a que reducía el estrés que sufre el silicio durante el proceso de metalización. Sin embargo, uno de sus isótopos, en concreto el ^{10}B, presente de forma natural con una abundancia [12] del 20% del boro natural, es altamente sensible a la radiación cósmica, ya que interactúa con los neutrones liberando partículas alfa e iones pesados (Figura 4) que pueden desencadenar eventos SET y como consecuencia, dar lugar a SBUs o MCUs dentro de la memoria. Por lo tanto, esto presenta una inconveniencia a la hora de construir chips con esta capa BPSG. En tecnologías de 180-nm y subsiguientes, esta capa se reemplazó por una equivalente sin boro, mitigando así enormemente el efecto de esta problemática [12].

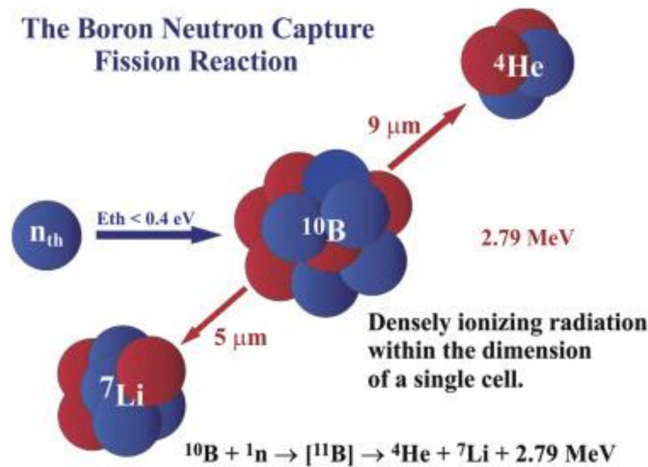

Figura 4. Reacción del boro al interactuar con neutrones [13].

2. **Tecnología SOI (Silicon-On-Insulator)**: Esta tecnología se basa en añadir una capa aislante entre el sustrato y la capa superior de los transistores (Figura 5). Varios estudios [14] [15] han demostrado que la tecnología SOI aporta una reducción drástica en el número de eventos generados en comparación con otras tecnologías de la misma generación. Como aspecto negativo, esta tecnología es altamente sensible a un tipo concreto de SEE conocido como "snapback" [16] [17]. Esta sensibilidad solo se encuentra en las versiones de esta tecnología que vacían parcialmente al semiconductor ("partially depleted") de portadores móviles (por ejemplo, electrones). Otras versiones de la tecnología en las que se consigue una capa de silicio mucho más pequeña consiguen vaciar por completo al semiconductor ("fully depleted") y eliminan la sensibilidad frente a los eventos "snapback".

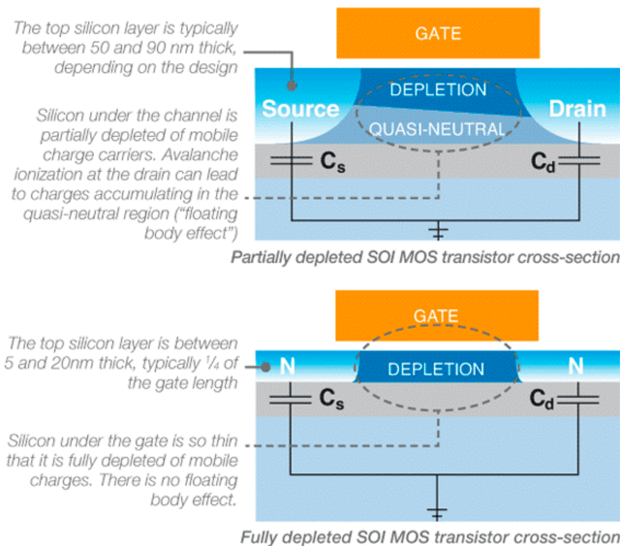

Figura 5. Diferencias entre SOI parcial y completamente vaciadas [2].

3. **Adición de impurezas**: En el caso de que ninguna de las tecnologías anteriores estuviese disponible, cabe la posibilidad de añadir una capa de impurezas debajo de los dispositivos internos con el objetivo de redirigir la carga de los SEEs lejos de los componentes sensibles a ella. Para ello se añaden condensadores que absorberán la carga pero que a su vez empeorarán el rendimiento general de la memoria a cambio de una mayor fiabilidad [12].

1.3.2 Mitigación por diseño

Otro método es la mitigación por diseño, el cual se basa en añadir más componentes (resistencias, transistores, etc.) que ayuden a la recuperación de datos tras un SEE. Este tipo de diseños están basados principalmente en las celdas de memoria de las SRAMs, las cuales están compuestas por dos inversores realimentados entre sí y un par de transistores de acceso. Con la finalidad de conservar los datos tras un SEE, se añaden transistores adicionales que conservarán una copia de los datos que se estén utilizando para poder ser recuperados cuando uno de estos eventos altere el contenido de la memoria, ya que la probabilidad de que todos los transistores que contengan cierta información (el original más las copias) es muy baja comparada con la probabilidad de que sólo sea uno afectado. Un famoso ejemplo son las celdas "Dual Interlocked storage CEll" (DICE)[3]. Fueron propuestas por T. Calin, M. Nicolaidis y R. Velazco [18] en 1996 y se popularizaron debido a su resistencia a los SEEs en una época donde no eran frecuentes los eventos múltiples ya que las arquitecturas de los chips rondaban los 350 nm. Estas celdas mitigaban bien los errores simples a cambio de usar más recursos hardware, pero no servían para mitigar errores múltiples que afectasen a dos de sus "latches". Es por ello que, según ha avanzado la tecnología y los chips han reducido su tamaño, los errores afectando a varios de sus "latches" simultáneamente se han vuelto más comunes y, por lo tanto, este tipo de celdas han perdido gran parte de su utilidad [19], aunque en su momento fueron una revolución importante.

[2] https://soiconsortium.org/2008/05/14/fully-depleted-fd-vs-partially-depleted-pd-soi/
[3] https://nepp.nasa.gov/files/25702/2013MRQW_Berg_n272.pdf

1.3.3 Mitigación por redundancia

Como último método se encuentra la mitigación por redundancia. Este método surge debido a que los enumerados anteriormente, a pesar de ser muy eficaces, dependen de la disponibilidad de los componentes o tecnologías correspondientes. Por lo tanto, surge un diverso número de tecnologías software que no dependen de la disponibilidad de ningún componente físico:

1. **Códigos de corrección de errores ("Error Correction Codes" o ECC)**: Consisten en añadir una serie de bits a cada palabra con la finalidad de detectar y corregir valores no válidos. Esta técnica es útil para proteger la información almacenada en memorias, pero conlleva una severa reducción en el área efectiva de la memoria [20] [21].
2. **Intercalado de bits**: Complementando a la técnica previa, se encuentra la popular técnica conocida como "bit interleaving", la cual ha sido explicada en la Sección "*1.1 Efectos de la radiación en los dispositivos*" y cuyo objetivo es el de minimizar la cantidad de MBUs y convertirlos en MCUs, reduciendo así el impacto de éstos en el funcionamiento normal de la memoria.
3. **Refresco o reinicio periódico**: Esta técnica está restringida a dispositivos reconfigurables como las FPGAs. Hay ocasiones en las que los SEEs pueden corromper datos sin tener una consecuencia inmediata. En otros casos, los dispositivos tienen copias que se pueden restaurar. En estas situaciones, un refresco o reinicio del dispositivo eliminará esos errores y se asegurará que la información almacenada es la correcta. Esta técnica también es conocida en la literatura científica como "scrubbing" [22].
4. **TMR ("Triple Modular Redundancy")**: En lugar de usar un solo sistema, el desarrollador implementa tres copias. Cuando se necesita acceder a la información, se realiza una votación de mayoría con las salidas de las tres copias. Gracias a este mecanismo, se asegura que la información es la apropiada en el caso de que, a lo sumo, solo una copia haya sido corrompida. En el caso de que dos de las copias sean erróneas este método será inválido, pero la probabilidad de que dos de tres copias de este sistema hayan sido afectadas por un SEE es muy reducida.
5. **Redundancia de tiempo**: Esta técnica consiste en comparar periódicamente las salidas de un sistema para detectar si ha ocurrido un tipo concreto de SEE en forma de pulso eléctrico, como los que se producen en caso de un SET. Las mediciones se realizan comparando una cierta salida con esa misma salida retrasada un cierto tiempo a través de una puerta XOR. Si estas dos son diferentes, significa que durante ese tiempo se ha producido un SET. Esta técnica acarrea dos inconvenientes, siendo el primero el caso en el que el pulso producido por el SET dure más que el tiempo entre medidas, y el segundo, el caso en el que se produzca en cambio intencionado en el valor de la salida durante el retardo entre medidas. Estas dos desventajas limitan la frecuencia a la que el dispositivo puede trabajar.
6. **Redundancia de software**: Finalmente y como se describe en [6], es posible añadir mecanismos de autodetección de errores dentro de un programa como la duplicidad de datos, redundancia temporal u otras técnicas similares a las descritas anteriormente. El mayor beneficio de esta técnica es que, al ser completamente por software, es válida para una inmensa variedad de dispositivos.

1.4 Motivación de este Trabajo de Fin de Grado

Debido a que ninguno de los métodos conocidos es infalible a la hora de evitar la ocurrencia de SEEs, es necesario que se sigan investigando la aparición de este tipo de eventos y los efectos que éstos implican. Con este objetivo en mente, los investigadores necesitan simular las situaciones en las que la radiación afecta a las memorias. Como se ha mencionado anteriormente, un método fácil y eficaz consiste en utilizar un acelerador de partículas que genere esas fuentes de radiación expuestas en la Sección *1.2 Fuentes de radiación*.

Para poder realizar estos estudios, es necesario alquilar el uso de uno de estos aceleradores por un alto coste. Por ello, es imprescindible hacer un uso eficiente de estas instalaciones para reducir costes y simplificar los experimentos, y que, aun así, se obtengan resultados estadísticamente válidos.

Cuando se desea estudiar los efectos de la radiación sobre un dispositivo o memoria, este estudio puede verse afectado a causa de los llamados falsos MCUs. Como se ha expuesto en puntos anteriores, los MCUs se producen cuando un solo SEE afecta a dos celdas de memoria físicamente vecinas. Sin embargo, los falsos MCUs se producen cuando dos bits o celdas vecinas son alteradas, pero por dos partículas de radiación diferentes, que casualmente afectaron a celdas vecinas o cercanas [23]. Esto provoca una seria problemática para la fiabilidad de los datos obtenidos ya que se pueden contabilizar múltiples SBUs como eventos múltiples.

Para ello, en este Trabajo de Fin de Grado se ha desarrollado una herramienta que sirva de apoyo al investigador a encontrar un tiempo idóneo durante el cual la exposición de una memoria a la radiación de un acelerador de partículas genere unos resultados lo más precisos posibles conociendo exactamente la probabilidad de aparición de los citados falsos MCUs. Con esta finalidad, la aplicación dibujará un gráfico con la probabilidad de aparición de falsos MCUs que se producirán en función del número de eventos totales observados, que a su vez dependerá del tiempo de exposición a la fuente de radiación usada en el experimento. Esto será de utilidad para poder ajustar la duración de las rondas de irradiación de manera que se observe una cantidad significativa de errores sin aumentar en exceso la probabilidad de aparición de falsos MCUs. Consideramos que esta herramienta tendrá un impacto positivo en los costes logísticos de los experimentos que se realizarán, y en la calidad de los resultados obtenidos.

2. Estimación del número de falsos MCUs

2.1 La “paradoja del cumpleaños”

Las ecuaciones utilizadas en la aplicación que se ha desarrollado para estimar el número de falsos MCUs, y que se describen en el siguiente apartado, tienen como origen la idea de la “paradoja del cumpleaños”. Propuesta inicialmente en [24] para estimar la población de peces en un lago, esta “paradoja” trata de calcular la probabilidad de que dos personas dentro de un grupo hayan nacido exactamente en el mismo día. Tras realizar los cálculos (Ecuación 1) se puede demostrar que, con un grupo de sólo 23 personas, ya hay una probabilidad superior al 50% de que haya dos personas dentro de ese grupo que compartan día de nacimiento (Figura 6).

$$1 - p = \begin{cases} 1 - \dfrac{365!}{365^n \, (365 - n)!}, & 1 \leq n \leq 365 \\ 1, & n > 365 \end{cases}$$

Ecuación 1. Cálculo de la probabilidad de que al menos dos personas compartan cumpleaños.

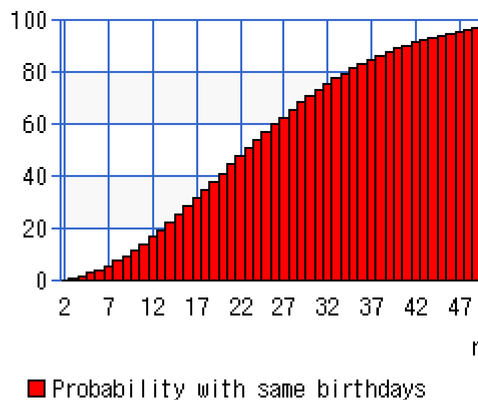

Figura 6. Resultados de la "paradoja del cumpleaños". El eje X contiene el número de personas del grupo y, el eje Y, la probabilidad de que dos personas compartan cumpleaños.

Esta teoría es extensible para analizar el caso de que dos personas cuyos cumpleaños estén separados por un número *k* de días. Por ejemplo, analizando los 46 presidentes de la historia de los Estados Unidos, se puede observar que hay 8 parejas de presidentes cuyos cumpleaños distan 1 día ($k = 1$) (Figura 7).

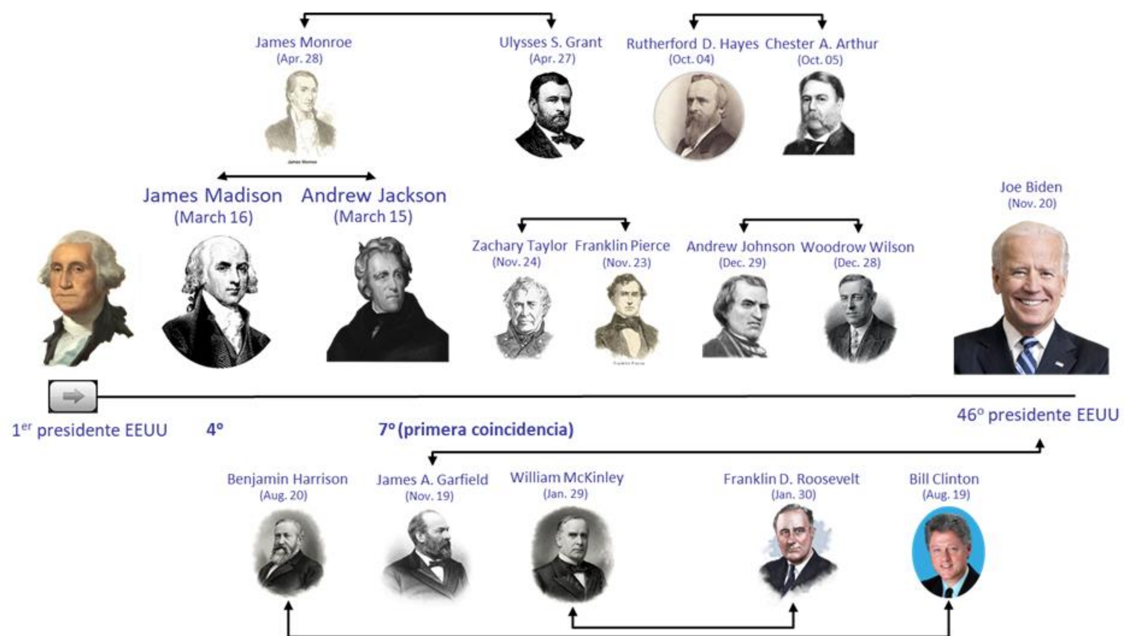

Figura 7. Presidentes de E.E.U.U. cuyos cumpleaños distan 1 día.

Esta idea se puede aplicar al caso de las memorias. Si sustituimos las personas que cumplen años por bitflips que afectan a una dirección de memoria, el número de días que distan (*k*) por la distancia a la que se encuentra un bitflip de otro y el número de días que tiene un año por el número de direcciones que tiene una memoria, se pueden adaptar entonces los cálculos de la "paradoja del cumpleaños" al cálculo del número de falsos MCUs que pueden aparecer durante un estudio. Tausch [25] fue el primero en aplicar este método para la detección de falsos MCUs.

2.2 Ecuaciones de detección en función de la distancia entre celdas

Dentro del artículo [26] se proponen una serie de técnicas de agrupación de bitflips en un experimento para identificar eventos múltiples; en concreto:

- **Métodos estadísticos**, que se utilizan asumiendo que no se conoce la estructura interna de la memoria bajo estudio y, por tanto, se desconoce la posición física dentro de la memoria de los bitflips observados en el experimento. Su estudio está totalmente fuera del alcance de este trabajo.
- **Métodos geométricos**, que se pueden utilizar en caso contrario; es decir, sí se conoce la estructura interna de la memoria bajo estudio, por tanto, se puede calcular la posición física en la memoria de cada bitflip observado, lo cual permite agruparlos con algún criterio basado en distancia. Estos métodos son los que se utilizarán en la herramienta desarrollada en este trabajo. En concreto, se utilizarán la distancia Manhattan ("Manhattan Distance" o MD) y la "Infinity Norm Distance" (IND). Las celdas que se encuentran a una MD o IND determinada con respecto a una celda de referencia se muestran en la Figura 8. Adicionalmente, estos métodos requieren que se determine una distancia umbral D que servirá para obtener el criterio de agrupación mencionado anteriormente. Esta distancia afectará a diferentes áreas según el método a usar. En la Figura 8 se muestra cómo se interpreta la distancia en los métodos MD e IND respectivamente para una distancia $D \leq 3$.

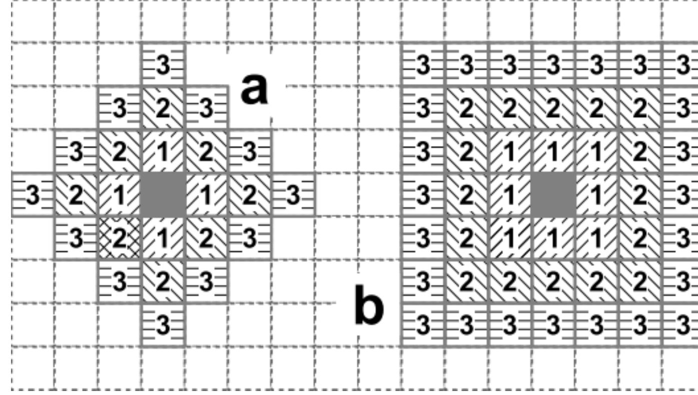

Figura 8. Celdas que se encuentran a una distancia D = 1, 2 o 3 con respecto a una celda central (coloreada en gris) en el método MD (a) y en el método IND (b) [27].

Estos métodos, junto con el parámetro D, se utilizarán para determinar si dos celdas afectadas por bitflips están a una distancia menor que esa distancia umbral. En el caso de usar el método MD, y de que se puedan asociar las direcciones lógicas a las físicas, estas últimas son colocadas en un plano XY $[a_i \rightarrow (x_i, y_i),\ a_j \rightarrow (x_j, y_j)]$ [26]. Si este nuevo par se encuentra a una distancia menor que D, se considera que ese par forma parte de un MCU. De tal forma que:

$$d_{MD}(a_i, a_j) = |x_i - x_j| + |y_i - y_j| \leq D$$

Ecuación 2. Manhattan Distance (MD). Extraída del artículo [26].

Por otro lado, y de forma similar, se encuentra el método IND, cuya ecuación es expresada de la siguiente manera:

$$d_{IND}(a_i, a_j) = \max(|x_i - x_j|, |y_i - y_j|) \leq D$$

Ecuación 3. Infinite Norm Distance (IND). Extraída del artículo [26].

Para poder utilizar estas ecuaciones, lo primero es calcular las posibles parejas de celdas de memoria afectadas por la radiación. Sabiendo que el número total de bitflips observados en un experimento es N_{BF}, se crea un conjunto con las posibles celdas de memoria que han sido afectadas, obteniendo $A = \{a_1, a_2, \ldots, a_{N_{BF}}\}$ [26]. A partir de este conjunto A, las parejas posibles de bitflips (N_P) se calculan obteniendo todas las combinaciones posibles del conjunto A:

$$N_P = \binom{N_{BF}}{2} = \frac{1}{2} \cdot N_{BF} \cdot (N_{BF} - 1)$$

Ecuación 4. Combinaciones del conjunto A. Ecuación extraída del artículo [26].

El siguiente paso es determinar cuáles de las parejas obtenidas se encuentran a una distancia menor que la distancia umbral, usando para ello las ecuacionesEcuación 2 y Ecuación 3. El uso de una u otra ecuación dependerá del investigador y del criterio que desee usar para determinar la distancia D.

Finalmente, utilizando la teoría de combinatoria, y más en concreto, ideas relacionadas con la resolución del problema de las urnas y las bolas[4], se pueden utilizar los parámetros N_P y D, definidos previamente, junto con el tamaño de la memoria (L_N), para obtener las ecuaciones finales que devolverán un valor aproximado del posible número de falsos MCUs de multiplicidad 2 que se estima se producirán para un determinado número de bitflips observados en un experimento. Conociendo este número de bitflips, se puede obtener N_P, y con ello, se utilizan el resto de los parámetros para así obtener las ecuaciones Ecuación 5 y Ecuación 6.

$$N_{FM2} \simeq {L_N}^{-1} \cdot N_P \cdot 2 \cdot D \cdot (D+1)$$

Ecuación 5. Ecuación final para estimar el número de falsos MCUs de multiplicidad 2, utilizando distancia MD. Extraída del artículo [26].

$$N_{FM2} \simeq {L_N}^{-1} \cdot N_P \cdot 4 \cdot D \cdot (D+1)$$

Ecuación 6. Ecuación final para estimar el número de falsos MCUs de multiplicidad 2, utilizando distancia IND. Extraída del artículo [26].

Estas dos ecuaciones serán las que se usen en la aplicación, la cual tendrá como parámetros el tamaño de la memoria, la distancia D y el método a emplear (MD o IND). Con estos datos, se mostrará una gráfica con la probabilidad de aparición de falsos MCUs de multiplicidad 2 (eje Y) para un determinado número de bitflips observados en un experimento (eje X). De esta forma, el investigador podrá estimar de forma más precisa cuántos bitflips son necesarios observar para obtener una cierta probabilidad de aparición de falsos MCUs. Nótese que este trabajo se focaliza únicamente en la aparición de falsos eventos múltiples de multiplicidad 2, que son los primeros que tienen mayor probabilidad de aparecer en experimentos con un gran número de bitflips. A pesar de que puede estimar la probabilidad de ocurrencia de falsos MCUs de multiplicidad 3 [26] o incluso superior, su probabilidad de aparición es muy inferior, por lo que se han excluido del estudio de este trabajo.

[4] https://www.statisticshowto.com/urn-model/

3. Desarrollo de la aplicación

Una vez explicada la función que cumplirá la aplicación, es hora de hablar de su desarrollo. A continuación, se expondrán los diferentes aspectos asociados al desarrollo como la plataforma, lenguaje de programación, configuración del proyecto y definición del código.

3.1 Plan de trabajo

El primer paso a la hora de realizar una aplicación es reunir los requisitos. Para este caso son:

- Elegir el lenguaje de programación que se va a usar.
- Implementar un formulario para la introducción de datos de entrada.
- Recoger los datos de ese formulario.
- Dibujar un gráfico que, dados unos parámetros de entrada, muestre los resultados esperados.

Por preferencia del autor, además de tener un gran número de librerías que son de ayuda para las interfaces gráficas, se usará Java como lenguaje para el desarrollo de la aplicación. Teniendo esto en cuenta, el punto de partida será desarrollar la funcionalidad básica del formulario y la comunicación del formulario con el gráfico. Una vez desarrollada esta funcionalidad, se deberán implementar los cálculos necesarios para que el gráfico muestre los datos apropiados. Y, por último, la interfaz de usuario, intentando obtener una interfaz que sea amable con el usuario y que sea consistente a lo largo de las diferentes pantallas de la aplicación.

3.2 Plataforma de desarrollo

El lenguaje de programación escogido para el desarrollo ha sido Java. Debido a su gran popularidad, el lenguaje está dotado de una gran variedad de librerías de código que agilizan el trabajo del programador, así como una gran cantidad de foros de ayuda donde buscar información referente al funcionamiento de éste.

Ya que el lenguaje elegido es Java, se ha decidido usar Eclipse (https://www.eclipse.org/) como plataforma de desarrollo. Esta plataforma aporta gran comodidad para el usuario automatizando tareas básicas y, por lo tanto, agiliza el desarrollo de la aplicación. Otras opciones como NetBeans o IntelliJ fueron descartadas debido a que el autor posee más experiencia con Eclipse que con las demás.

Como complemento a esta plataforma, se ha usado GitHub (https://github.com/) como herramienta de almacenamiento en la nube. Bien es conocido que GitHub es una de las plataformas predilectas para que los programadores guarden su código. Sus funcionalidades de gestionado de versiones, así como su detección automática de los cambios hacen que se merezca su popularidad. Además, ofrece la posibilidad de crear repositorios privados que pueden ser publicados con un solo clic.

3.3 JavaFX

Para el diseño gráfico de la aplicación se han utilizado las librerías externas de JavaFX (https://openjfx.io/). Estas librerías aportan clases con un diseño

predeterminado para determinadas funcionalidades. En este caso se ha utilizado principalmente para el gráfico resultante tras aplicar los cálculos.

Para el formulario que recopila los datos de entrada que se pedirán al usuario, se ha usado JavaFX también pero no se ha usado un diseño predeterminado. En su lugar se ha utilizado un diseño propio del autor creado mediante la herramienta SceneBuilder cuya integración con Eclipse es facilitada por JavaFX. Esta integración permite la creación de archivos FXML, que son archivos de código HTML pero adaptados para que puedan ser leídos por las clases de JavaFX.

Los archivos FXML son editados desde el programa SceneBuilder con una interfaz gráfica. Cuando el diseñador guarda los cambios dentro del programa SceneBuilder, se actualiza el fichero FXML con el código HTML asociado a los elementos que estén actualmente en la interfaz gráfica.

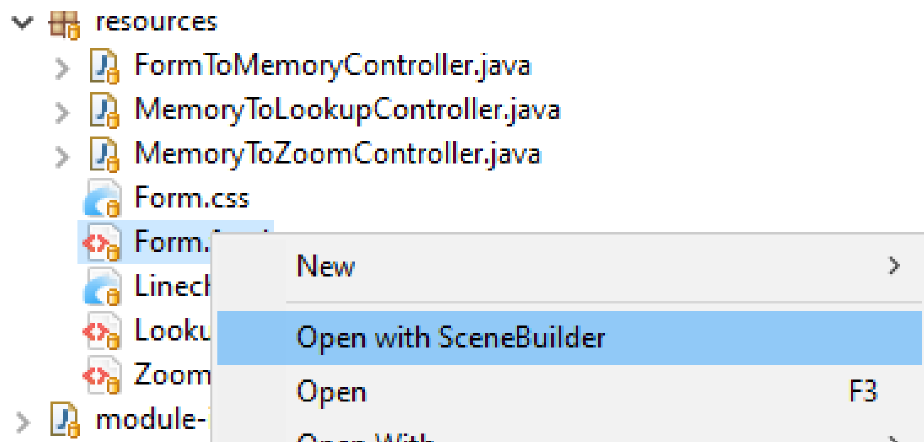

Figura 9. Menú de contexto de un fichero FXML.

Haciendo clic derecho en un fichero FXML, el menú de opciones muestra la opción de abrir con SceneBuilder. Esto abre la interfaz gráfica de desarrollo del elemento que hayamos escogido. En el caso de la Figura 10, el formulario que recoge los datos de entrada.

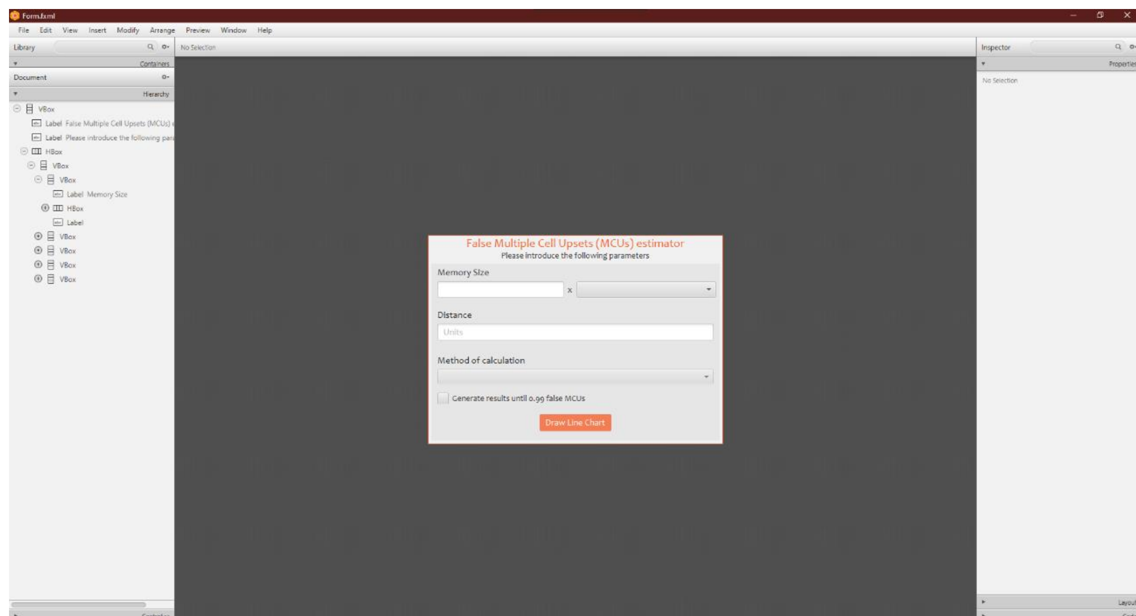

Figura 10. Interfaz gráfica de SceneBuilder.

En el panel de la izquierda se puede observar el panel de herramientas que utiliza SceneBuilder con todos los elementos que se pueden añadir al fichero FXML (Figura 11), así como los elementos que se están usando en el fichero actual (Figura 12).

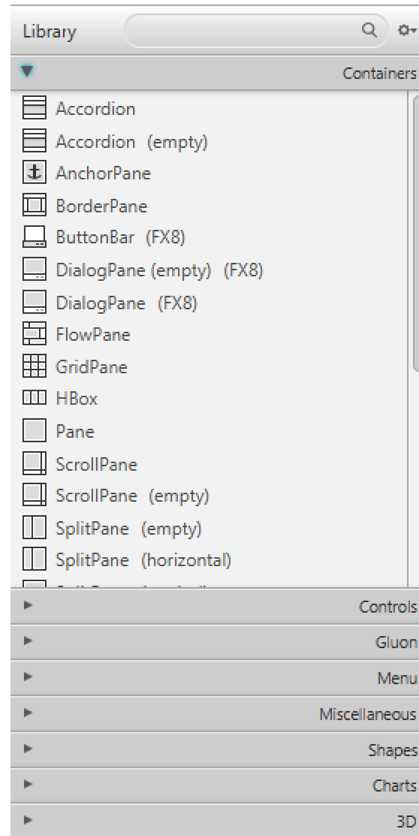

Figura 11. Menú de opciones de SceneBuilder.

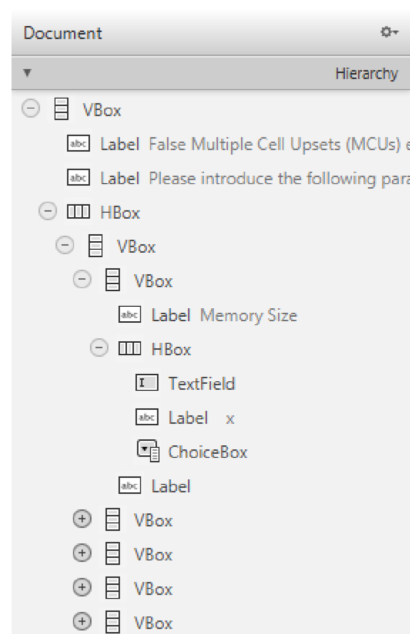

Figura 12. Lista de elementos usados en SceneBuilder.

En la derecha se sitúa el panel de opciones para un elemento. Al seleccionar un elemento (por ejemplo, una etiqueta de texto), se podrán gestionar desde este panel opciones de estilo, disposición, así como un identificador que posteriormente se podrá usar en el código JavaFX o un identificador de clase CSS.

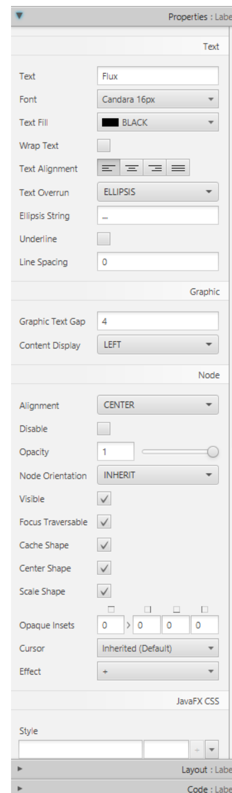

Figura 13. Menú de opciones de un elemento.

3.4 Configuración del proyecto

Al ser JavaFX un conjunto de librerías externas es necesario realizar cierta configuración al proyecto para poder obtener su funcionalidad.

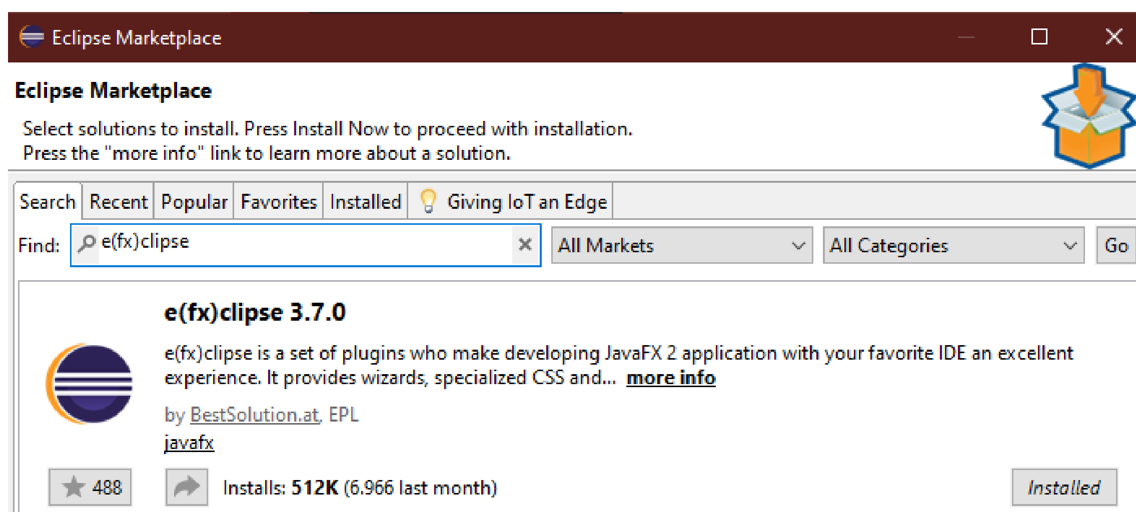

Figura 14. Extensión e(fx)clipse.

Lo primero es instalar una extensión desde el Marketplace de Eclipse llamada e(fx)clipse (Figura 14). Esta extensión permitirá a Eclipse crear proyectos de JavaFX y ficheros de tipo FXML.

Una vez creado el proyecto con la nueva opción que ha sido habilitada, es necesario añadir las librerías correspondientes a JavaFX al propio proyecto. En versiones anteriores del Java Development Kit (JDK), estas librerías venían incluidas, pero en versiones más recientes hay que instalarlas aparte. Para ello, es necesario descargarse el fichero SDK desde la página web de JavaFX y añadirlo como librerías de usuario al Build Path del proyecto.

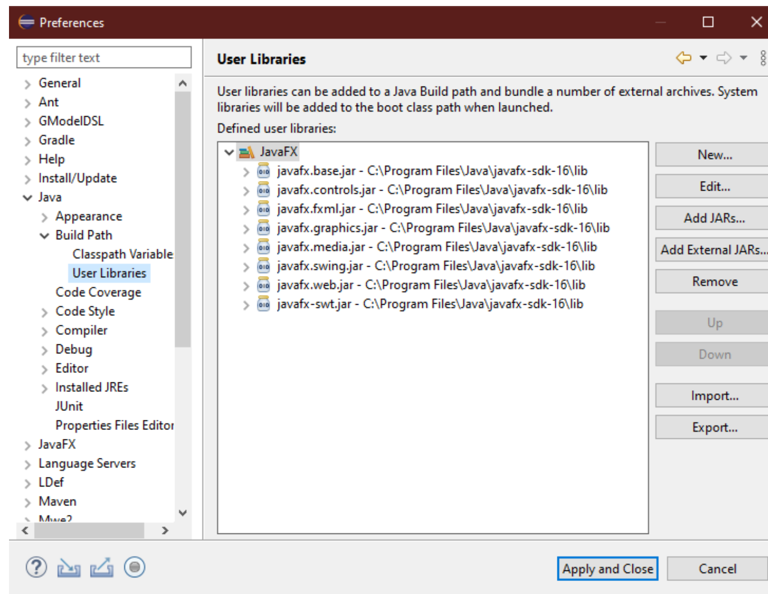

Figura 15. Librería de usuario para JavaFX.

Una vez creada la librería, por último, es necesario añadirla al "Modulepath" del proyecto junto con el propio SDK, el cual será añadido al Classpath (Figura 15 y Figura 16).

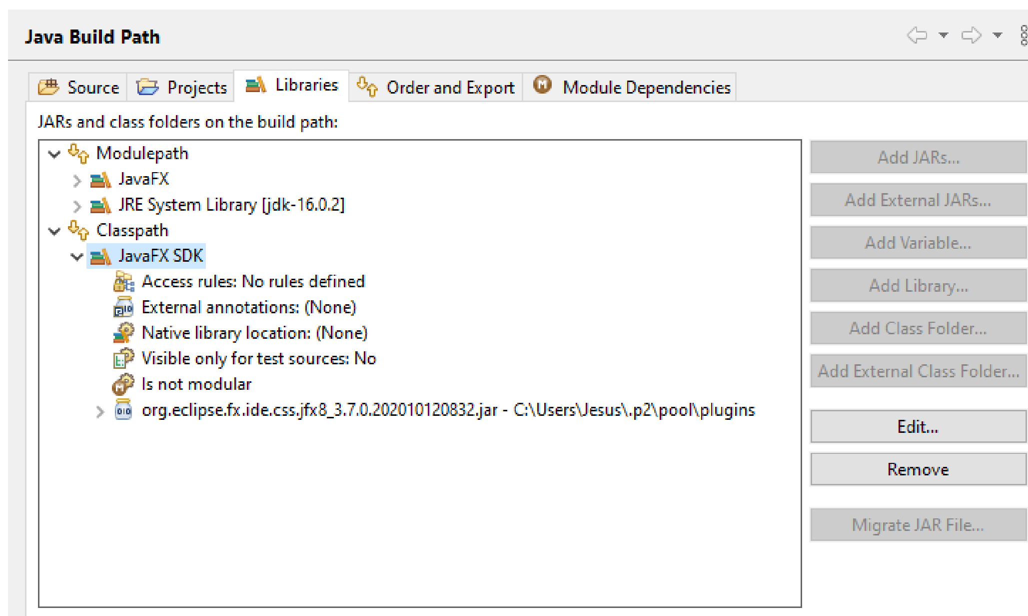

Figura 16. Java Build Path del proyecto.

3.5 Estructura del proyecto

La estructura del proyecto se compone de tres paquetes diferentes. El primer paquete, cuyo nombre es “form”, es el que contiene la clase “AlternativeMain.java” que ejecuta el programa y muestra el formulario dado un fichero “Form.fxml” y otro fichero “Form.css”, que aportan el estilo de la interfaz de usuario. Dentro de este paquete también se encuentra la clase “FormFX.java”, que controla los datos introducidos por el usuario en el formulario con apoyo de los enumerados “MethodSelection.java” y “SizeSelection.java”.

El segundo paquete, que tiene como nombre “linechart”, usa la clase Memory.java para realizar los cálculos y mostrar el gráfico.

Y, por último, el paquete “resources” que contiene los ficheros FXML y CSS, además de las clases controladoras de escenas. Estas clases controladoras se encargan de recoger los datos necesarios de una escena y enviarlos a la nueva escena que se va a mostrar al usuario. Estas clases, junto con los ficheros FXML y CSS, los cuales se encargan de aportar estilo a la interfaz, con las encargadas de crear estas escenas y mostrarlas al usuario por pantalla.

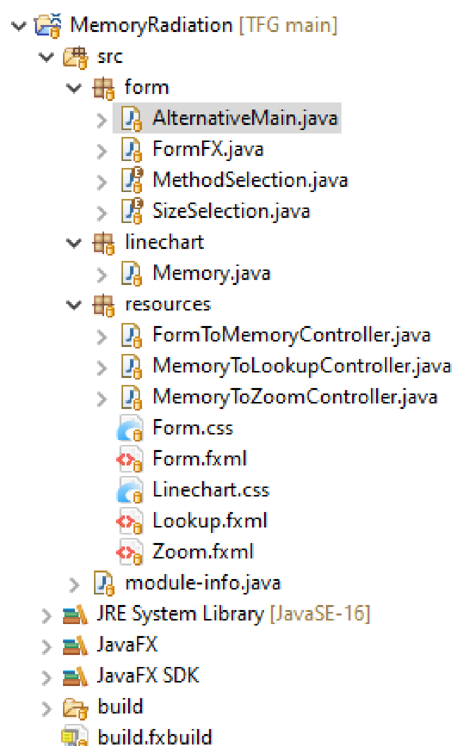

Figura 17. Estructura del proyecto en Eclipse.

Adicionalmente a estos paquetes, los proyectos de JavaFX requieren de un archivo module-info.java. Este fichero contiene requerimientos, así como permisos del módulo de JavaFX para que funcione correctamente. En este caso:

```
1  module HelloFX {
2      requires javafx.controls;
3      requires transitive javafx.graphics;
4      requires javafx.fxml;
5      requires transitive javafx.base;
6
7      opens linechart;
8      opens form;
9      opens resources;
10
11     exports linechart;
12     exports form;
13     exports resources;
14 }
```

Figura 18. Clase module-info.java.

El módulo HelloFX contiene los requerimientos de librerías de JavaFX que usa la aplicación: javafx.controls, javafx.graphics, javafx.fxml y javafx.base. Estas son las librerías que se usan en toda la aplicación y gracias a la instrucción "opens" (Figura 18), se permite el acceso de los paquetes de la aplicación a las librerías de JavaFX correspondientes. Adicionalmente, la instrucción "exports" sirve para que a la hora de exportar el proyecto a un formato JAR ejecutable, este ´sea exportado junto con las librerías necesarias.

3.6 Código

A continuación, se explicará el código desarrollado para el funcionamiento de la aplicación. En esta explicación se seguirá el flujo de ejecución normal del programa explicando la funcionalidad de cada clase y cuándo se ejecuta cada parte del código.

3.6.1 Formulario

Lo primero que el usuario puede observar al ejecutar la aplicación es el formulario de introducción de parámetros. Como ya se ha expuesto en apartados anteriores, estos parámetros son el tamaño de la memoria en Mbits, la distancia D, y el método de agrupación de bitflips. Además, también se da la opción de realizar los cálculos hasta que se llegue a una probabilidad del 99% de aparición de falsos MCUs.

Figura 19. Formulario de parámetros.

Como se ha explicado en el apartado "*3.3 JavaFX*", se ha usado la herramienta SceneBuilder para crear la interfaz de usuario de la Figura 19, resultando en el fichero FXML "Form.fxml". En la Figura 20 se muestra parte de este fichero, en concreto, el código asociado al campo de la distancia D. Este extracto muestra el código asociado al texto "Distance" (líneas 77 a 84), así como el del campo de introducción de datos (líneas 85 a 92) y el de la etiqueta de error que aparece cuando los datos introducidos no son válidos (líneas 93 a 100).

También se puede observar que dos de estos elementos tienen la propiedad *"fx:id"* (`"distanceTextField"` y `"distanceError"`). Estos identificadores servirán para poder conectar el código Java con el código FXML y permitir el traspaso de datos entre escenas.

```
75 <VBox prefHeight="46.0" prefWidth="488.0">
76    <children>
77       <Label alignment="CENTER" text="Distance">
78          <VBox.margin>
79             <Insets left="10.0" right="5.0" />
80          </VBox.margin>
81          <font>
82             <Font name="Candara" size="16.0" />
83          </font>
84       </Label>
85       <TextField fx:id="distanceTextField" prefHeight="25.0" prefWidth="433.0" promptText="Units" styleClass="input-text">
86          <VBox.margin>
87             <Insets bottom="5.0" left="10.0" right="10.0" top="5.0" />
88          </VBox.margin>
89          <font>
90             <Font name="Candara" size="15.0" />
91          </font>
92       </TextField>
93       <Label fx:id="distanceError" textFill="#f20000">
94          <font>
95             <Font size="13.0" />
96          </font>
97          <VBox.margin>
98             <Insets left="10.0" />
99          </VBox.margin>
100      </Label>
101   </children>
102 </VBox>
```

Figura 20. Fragmento de código extraído del fichero Form.fxml

Complementando al fichero FXML, es posible añadir un fichero CSS para definir el estilo gráfico de la ventana. Aunque SceneBuilder es capaz de configurar gran cantidad de parámetros de estilo, hay algunos como el color del fondo o los bordes que se han de añadir desde un fichero externo (Figura 21).

```
.anchor{
    -fx-background-color: transparent;
}

.vbox{
    -fx-background-color: #f1f1f1;
    -fx-border-color: #f3622d;
    -fx-border-width: 1px;
}

.hbox{
    -fx-background-color: #E6E6E6;
}

.form-btn{
    -fx-background-color: #f3622d;
}
```

Figura 21. Fichero CSS asociado al formulario principal.

Una vez creada la interfaz, sólo falta conectarla con el código para que se muestre al arrancar el programa. Esta tarea es desempeñada por la clase “FormFX.java” contenida en el paquete “form”. En el mismo paquete también se define la enumeración “MethodSelection.java”, la cual definirá los campos del menú desplegable del formulario.

Dentro de la clase “FormFX.java” se encuentra la función *main()* de la aplicación. Esta función realiza una llamada a la función *launch(args)* de la librería “javafx.application” y lanza el programa invocando a la función *start(primaryStage)*. Esta función *start()* genera la primera escena que se va a mostrar por pantalla, que en este caso será el formulario. Para ello es necesario cargar los dos ficheros “Form.fxml” y “Form.css” y mostrarlos por pantalla.

```
1  package form;
2
3⊕ import java.net.URL;[]
11
12 public class FormFX extends Application {
13
14⊖     /**
15      * Displays the form and fills it with default values
16      */
17⊖     @Override
18     public void start(Stage primaryStage) {
19         try {
20             URL fxmlocation = getClass().getResource("/resources/Form.fxml");
21             FXMLLoader loader = new FXMLLoader(fxmlocation);
22
23             Parent root = loader.load();
24             Scene scene = new Scene(root);
25             String css = getClass().getResource("/resources/Form.css").toExternalForm();
26             scene.getStylesheets().add(css);
27
28             FormToMemoryController controller = loader.getController();
29             controller.setMethod(MethodSelection.MD);
30             controller.setSizeMultiplier(SizeSelection.MBITS);
31
32             primaryStage.setTitle("False Multiple Cell Upsets (MCUs) estimator");
33             primaryStage.setScene(scene);
34             primaryStage.show();
35
36         }
37         catch(Exception e) {
38             e.printStackTrace();
39         }
40
41     }
42
43⊖     /**
44      * Main function that opens the form
45      */
46⊖     public static void main(String[] args) {
47         launch(args);
48     }
49 }
50
```

Figura 22. Clase FormFX.java.

En la Figura 22 se muestra la clase “FormFX.java” al completo. De las líneas 20 a la 30, se obtienen los recursos necesarios para generar la escena (archivos FXML y CSS, y asignación de valores por defecto a los menús desplegables) que se muestra en las líneas 32 a 34.

También cabe mencionar que la clase que contenga el *main* en una aplicación JavaFX ha de extender la interfaz “Application” que aportan estas librerías. Esta interfaz aporta los métodos *launch()* y *start()*. De esta forma, se consigue el punto inicial de la aplicación que muestra el formulario que verán los usuarios.

3.6.2 Control de escenas

Para el control de escenas se ha usado una clase diferente para cada transición. La aplicación tiene un total de cuatro escenas diferentes: el formulario principal, el gráfico y otras dos ventanas emergentes para funcionalidad adicional.

En el paquete "resources" de la aplicación, se encuentran las tres clases controladoras. Al tener estas clases un comportamiento muy similar entre ellas, solo se entrará en detalle de una de ellas ("FormToMemoryController.java") debido a que trata más transiciones que las otras clases controladoras y, por lo tanto, es la más representativa. Es por ello que también, el número de clases controladoras y de escenas no coincide.

En esta clase se recogen los datos introducidos en el formulario, se comprueba que sean válidos y, por último, se envían a la clase que calculará los datos que debe mostrar el gráfico.

Para recoger los datos, es necesario conectar el código Java con el código FXML que se ha creado con anterioridad. Para ello se requiere el uso de la etiqueta "@FXML" dentro del código Java tal y como se muestra en la Figura 23.

```
// Form parameters
@FXML
private TextField sizeTextField;
@FXML
private TextField distanceTextField;
@FXML
private ChoiceBox<MethodSelection> dropdown;
private MethodSelection[] method = {MethodSelection.MD, MethodSelection.IND};
@FXML
private ChoiceBox<SizeSelection> sizeChoice;
private SizeSelection[] sizeMultipliers = {SizeSelection.BITS, SizeSelection.KBITS, SizeSelection.MBITS};
@FXML
private CheckBox checkbox;
@FXML
private Label sizeError;
@FXML
private Label distanceError;
```

Figura 23. Atributos Java donde se recogerán los datos del formulario.

Como se puede observar en la Figura 23, los atributos *"distanceTextField"* y *"distanceError"* poseen el mismo nombre que la propiedad *"fx:id"* que se ha mencionado anteriormente. Esto, junto con la etiqueta "@FXML", consigue que se conecten los campos del formulario con los atributos que tengan el mismo nombre que el identificador.

Una vez recogidos los datos, se comprueba que sean válidos, se crea un objeto de la clase "Memory.java" que se encargará de realizar el cálculo de errores y se cambia la escena a la nueva escena que mostrará el gráfico. Toda esta funcionalidad se recoge en la función *switchToGraph(ActionEvent event)* de la clase controladora (Figura 24).

```java
public void switchToGraph(ActionEvent event) {
    boolean dataOk = true;
    sizeError.setText(null);
    distanceError.setText(null);
    Stage stage = (Stage)((Node)event.getSource()).getScene().getWindow();
    Memory mem = new Memory();

    try {
        mem.setSize(Integer.parseInt(sizeTextField.getText()), sizeChoice.getValue());
    }catch (NumberFormatException e) {
        sizeError.setText("Please enter a number");
        dataOk = false;
    }

    try {
        mem.setD(Integer.parseInt(distanceTextField.getText()));
    }catch (NumberFormatException e) {
        distanceError.setText("Please enter a number");
        dataOk = false;
    }

    mem.setMethod(dropdown.getValue());
    mem.setMultiplier(sizeChoice.getValue());
    mem.setGenerateUntilMax(checkbox.isSelected());

    if(dataOk)
        mem.start(stage);
}
```

Figura 24. Función switchToGraph() de la clase FormToMemoryController.java

3.5.3 Cálculo de errores

En la anterior clase, se crea el objeto *mem* de la clase "Memory.java". Este objeto posee los atributos que se muestran en la Figura 25.

```java
private int memorySize;
private SizeSelection multiplier;
private int D;
private MethodSelection method;
private boolean generateUntilMax;

private List<Double> errors;
private int nBitflips = 2000;
private XYChart.Series<Number, Number> data = new XYChart.Series<>();
LineChart<Number, Number> lineChart;
```

Figura 25. Atributos de la clase Memory.java.

Estos atributos se corresponden a los campos que se han rellenado en el formulario, tanto campos de texto, como menús desplegables y casillas marcables. Adicionalmente, el atributo *errors* almacenará los datos que se van a mostrar en el gráfico; es decir, la probabilidad de que aparezca un falso MCU en función de los bitflips observados.

Cuando en el método *switchToGraph()* se invoca a *start()* (Figura 26) y se envía la escena actual, se llama automáticamente a *init(stage)*, la cual inicializa el gráfico, calcula los datos y muestra los resultados por pantalla.

```java
@Override
public void start(Stage primaryStage) {
    init(primaryStage);
}
```

Figura 26. Método start().

Para generar el gráfico, lo primero es inicializar los ejes que se van a usar con el tipo de datos que se desea mostrar. Para ello, se crear los objetos *xAxis* e *yAxis* de tipo "NumberAxis" ya que vamos a representar números en ambos ejes. Con estos objetos, ya se puede crear el gráfico como se muestra en la Figura 27.

```java
33        NumberAxis xAxis = new NumberAxis();
34        xAxis.setLabel("Time");
35
36        NumberAxis yAxis = new NumberAxis();
37        yAxis.setLabel("False MCUs");
38
39        LineChart<Number, Number> lineChart = new LineChart<Number, Number>(xAxis, yAxis);
40        lineChart.setTitle("Memory Radiation");
41
42        XYChart.Series<Number, Number> data = new XYChart.Series<>();
43        data.setName("False MBUs / Time");
```

Figura 27. Creación del gráfico.

Una vez inicializado el gráfico, según el tipo de criterio de agrupación de bitflips que se haya elegido (MD o IND), se ejecutan los cálculos correspondientes. Para cada tipo de criterio se ha creado un método que se selecciona según se muestra en la Figura 28.

```java
if(this.method == MethodSelection.MD)
    this.calculateMD(this.data);
else
    this.calculateIND(this.data);

this.lineChart.getData().add(this.data);
```

Figura 28. Selección del método.

Los resultados obtenidos se almacenan en la variable "*data*" que se pasa por parámetro, la cual luego ha de ser añadida al gráfico para que los muestre. Esto se hace en la última línea de código de la Figura 28.

En las Figura 29 y Figura 30, se muestra el código que realiza los cálculos tanto para el criterio MD como para IND. Los resultados se guardan en dos variables. La primera es la variable *errors*, que es una lista de decimales para poder operar con los datos cuando se necesite, y en la variable *data* que es la que utiliza el gráfico para mostrar los datos por pantalla.

Una vez creado el gráfico, lo único que falta es mostrar la escena final para que el investigador que use la herramienta pueda observar la cantidad de errores esperados que se van a observar en su experimento. Esto se hará de la misma forma que se han mostrado las escenas en otras clases y que ya se ha explicado previamente.

```java
private void calculateMD(XYChart.Series<Number, Number> data) {
    this.errors = new ArrayList<Double>();
    double Np, Nbf, Nfm2;
    int i = 0;
    double currentError = 0;

    if(this.generateUntilMax) {
        while(currentError < 0.99) {
            Nbf = i;
            Np =  (Nbf * (Nbf - 1))/2;
            Nfm2 = Math.pow(this.memorySize, -1) * Np * 2 * this.D * (this.D + 1);
            currentError = 1 - Math.pow(Math.E, -Nfm2);
            this.errors.add(currentError);
            data.getData().add(new XYChart.Data<Number, Number>(i, currentError));
            ++i;
        }

    }
    else {
        for(i = 0; i <= this.nBitflips; ++i) {
            Nbf = i;
            Np =  (Nbf * (Nbf - 1))/2;
            Nfm2 = Math.pow(this.memorySize, -1) * Np * 2 * this.D * (this.D + 1);
            this.errors.add(1 - Math.pow(Math.E, -Nfm2));
            data.getData().add(new XYChart.Data<Number, Number>(i, this.errors.get(i)));
        }
    }

}
```

Figura 29. Código que calcula la probabilidad de falsos MCUs usando el criterio MD.

```java
private void calculateIND(XYChart.Series<Number, Number> data) {
    this.errors = new ArrayList<Double>();
    double Np, Nbf, Nfm2;

    int i = 0;
    double currentError = 0;

    if(this.generateUntilMax) {
        while(currentError < 0.99) {
            Nbf = i;
            Np =  (Nbf * (Nbf - 1))/2;
            Nfm2 = Math.pow(this.memorySize, -1) * Np * 4 * this.D * (this.D + 1);
            currentError = 1 - Math.pow(Math.E, -Nfm2);
            this.errors.add(currentError);
            data.getData().add(new XYChart.Data<Number, Number>(i, currentError));
            ++i;
        }
    }
    else {
        for(i = 0; i <= this.nBitflips; ++i) {
            Nbf = i;
            Np =  (Nbf * (Nbf - 1))/2;
            Nfm2 = Math.pow(this.memorySize, -1) * Np * 4 * this.D * (this.D + 1);
            currentError = 1 - Math.pow(Math.E, -Nfm2);
            this.errors.add(currentError);
            data.getData().add(new XYChart.Data<Number, Number>(i, currentError));
        }
    }

}
```

Figura 30. Código que calcula la probabilidad de falsos MCUs usando el criterio IND.

3.7 Portabilidad de la aplicación

Para convertir la aplicación a un formato portable y que se pueda ejecutar en las plataformas necesarias, se ha decido usar la utilidad `jpackage` incluida dentro del Java Development Kit o JDK.

Esta herramienta permite empaquetar el código de la aplicación junto con un entorno en el cual se pueda ejecutar accediendo a las librerías utilizadas en el mismo, en este caso, JavaFX.

Para utilizarla, lo primero es crear una “runtime image” de Java que juntará las librerías básicas de Java, contenidas en el JDK, y las librerías externas de JavaFX (Figura 31).

```
C:\Users\Jesus>jlink --output jdk-16+fx --module-path <path-to-javafx-jmods> --add-modules javafx.controls,
javafx.graphics,javafx.fxml,javafx.base
```

Figura 31. Uso del comando `jlink` *para preparar la imagen "runtime".*

A este comando se le debe indicar el nombre de la imagen (opción “output”), el directorio a los jmods de JavaFX que se encargarán de aportar las librerías necesarias y, por último, se debe de indicar cuáles de esas librerías son necesarias; en este caso javafx.controls, javafx.graphics, javafx.fxml, javafx.base, que fueron incluidas previamente en el fichero “module-info.java”.

Una vez creada la imagen, es posible utilizarla para crear el instalador usando `jpackage`. Esta herramienta da la opción de personalizar la instalación con utilidades como la creación de un acceso en el escritorio, añadir la aplicación al menú del sistema o añadir un icono a la aplicación, además de otras muchas funcionalidades. Para ejecutar el comando, se ha de introducir en la consola de comandos la instrucción de la Figura 32.

```
C:\Users\Jesus>jpackage -t exe --name <App name> --description <App description> --vendor <Owner name> --app-versio
n <version> --input <folder for input files> --dest <folder to store output files> --main-jar <Jar file of the app>
 --icon <path to icon> --runtime-image <path to image> --win-shortcut --win-menu
```

Figura 32. Uso del comando `jpackage`*.*

La mayoría se explican con su nombre, pero otras, merece la pena mencionarlas. Entre ellas está la opción `--input` que indica donde están almacenados tanto el fichero JAR ejecutable como las librerías que usará. La opción `--runtime-image` indica la dirección de la imagen que se ha creado previamente con el comando `jlink`. Por último, la opción `--main-jar` contiene la dirección al fichero JAR extraído del proyecto de Java.

Una vez ejecutado, se creará un fichero instalador que se podrá utilizar en cualquier plataforma, siempre y cuando se use el mismo sistema operativo que la máquina donde se creó. En caso de necesitar el instalador en otras plataformas, se tendría que crear una nueva imagen, desde dicha plataforma, con el comando `jlink` y seguir el resto del procedimiento con la nueva imagen.

4. Resultados

Una vez desarrollada la aplicación, se ha procedido al análisis de los resultados aportados por la misma dando diferentes valores a los datos de entrada que se le solicitan al usuario. El objetivo es compararlos con diversos experimentos realizados con anterioridad en campañas de irradiación realizadas por el grupo de investigación GHADIR de la Universidad Complutense de Madrid y analizar si, en aquellos casos, el uso de la herramienta hubiese conllevado una mayor eficiencia y un mayor ahorro de tiempo y recursos.

4.1 Estudio de las ecuaciones de probabilidad

Las figuras Figura 33 - Figura 38 muestran los resultados obtenidos por la aplicación para memorias de tamaños 1Mx8 bits y 2Mx8 bits que ya han sido estudiadas y cuyos resultados son conocidos. Estas memorias están fabricadas por Infineon Technologies y fueron estudiadas bajo radiación en las publicaciones [28] - [30] .

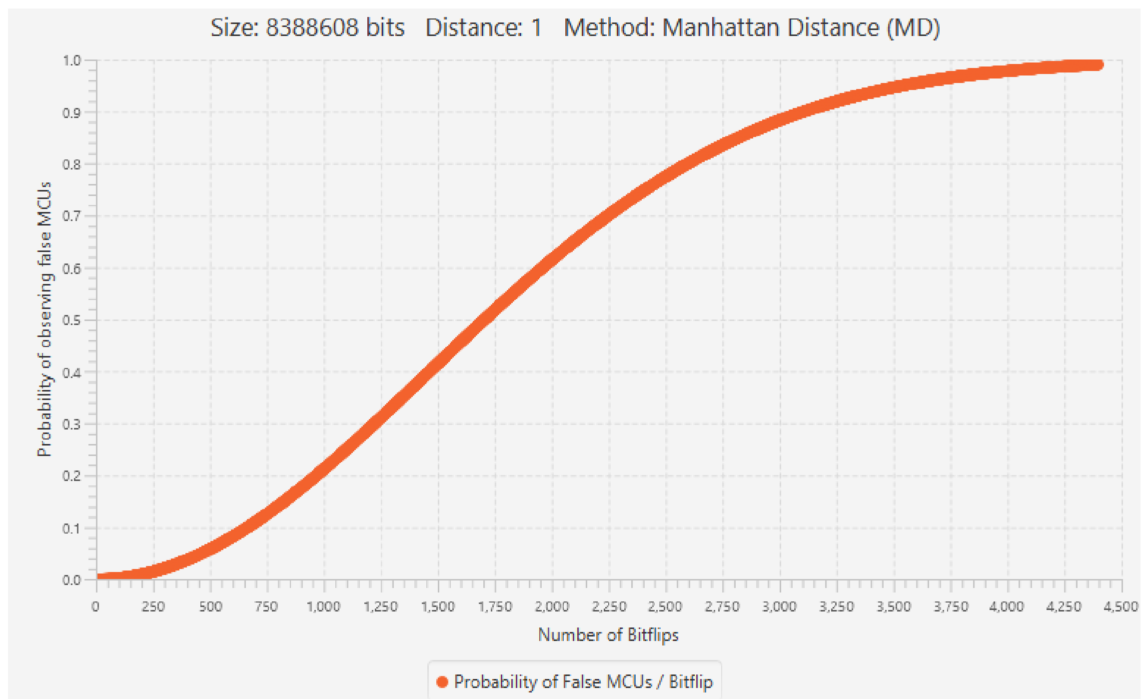

Figura 33. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 1MX8 bits y D = 1. Método MD.

Como se puede observar, al usar el método MD (Figura 33), que considera un menor rango de celdas como celdas vecinas, la probabilidad de aparición de falsos MCUs aumenta más lentamente que en el caso de la Figura 34, llegando a un 99% de probabilidad de aparición de falsos MCUs con 4396 bitflips, en comparación con los 3109 que requiere una memoria de idéntico tamaño, pero utilizando el método IND.

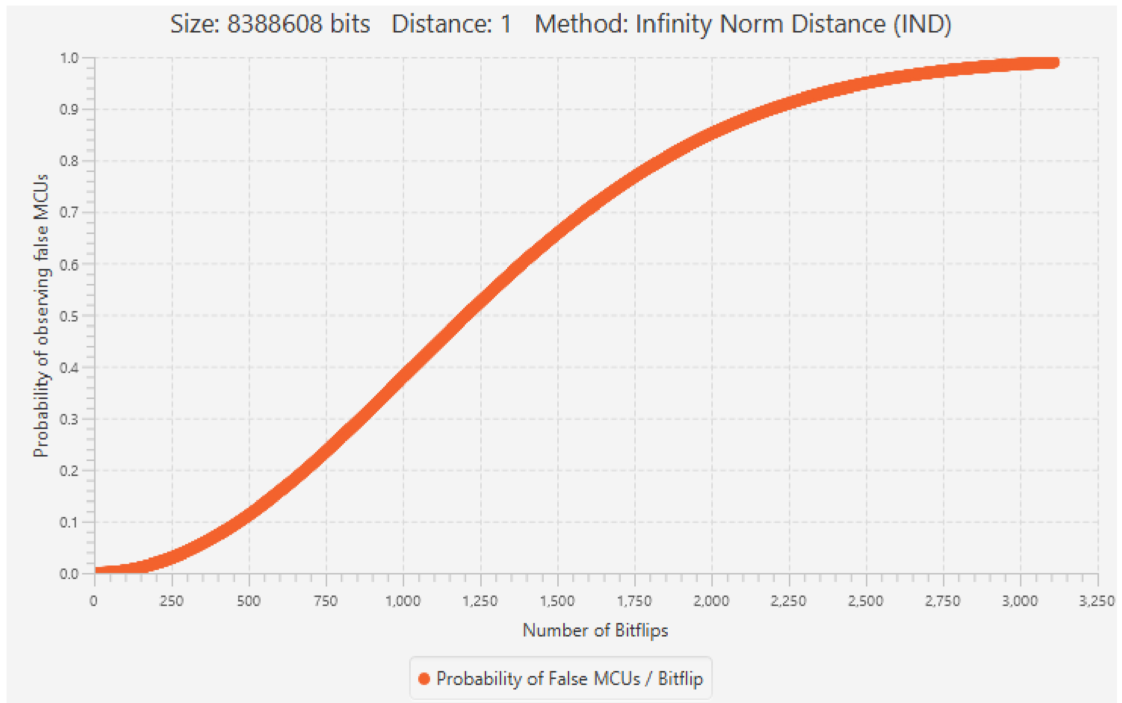

Figura 34. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 1MX8 bits y D = 1. Método IND.

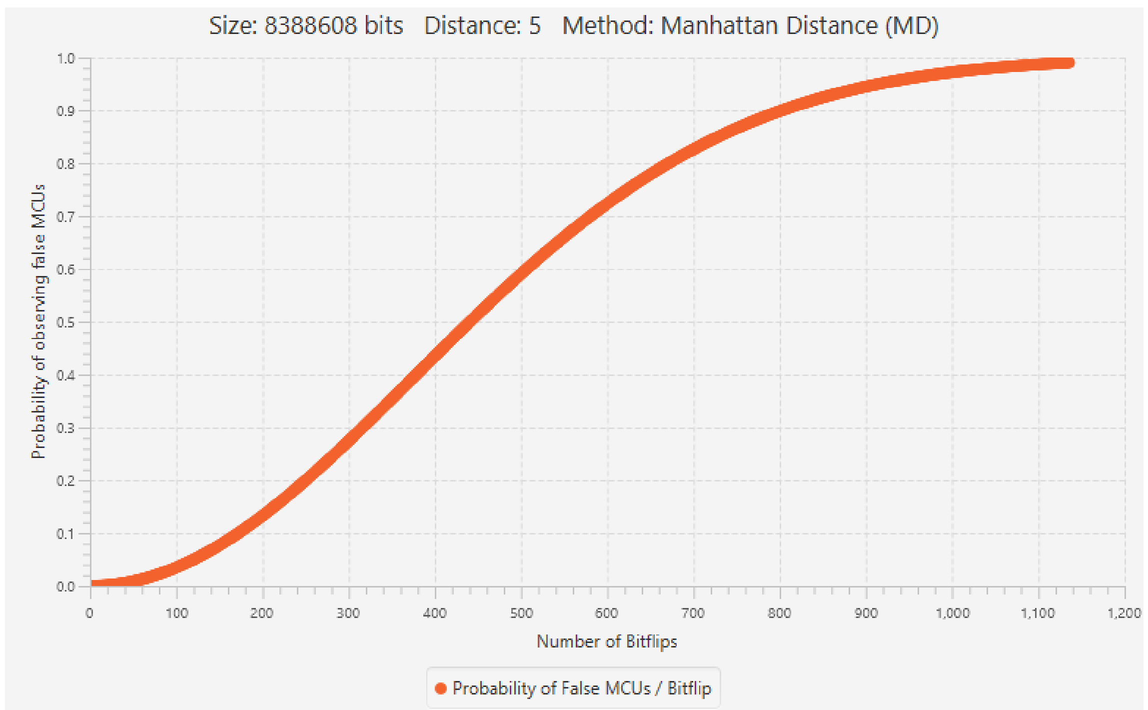

Figura 35. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 1MX8 bits y D = 5. Método MD.

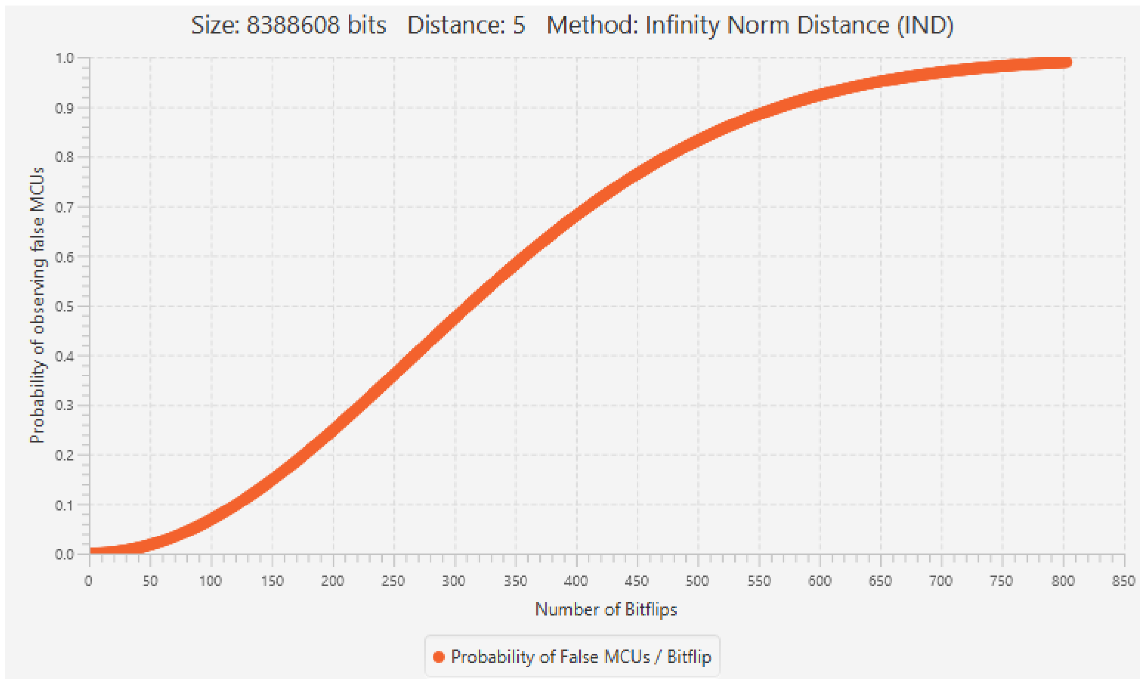

Figura 36. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 1MX8 bits y D = 5. Método IND.

A continuación, en las Figura 35 y Figura 36, se observa que, al aumentar la distancia *D* de 1 a 5, se incrementa de forma drástica la probabilidad de aparición de falsos eventos múltiples para una memoria del mismo tamaño, ya que el número de celdas vecinas que se agruparán bajo el mismo evento múltiple aumenta con el parámetro *D*. Esto tiene un claro impacto en las Figura 33 y Figura 34. En lo que antes se necesitaban 4396 y 3109 bitflips respectivamente, en estos casos se necesitan 1136 y 803 respectivamente.

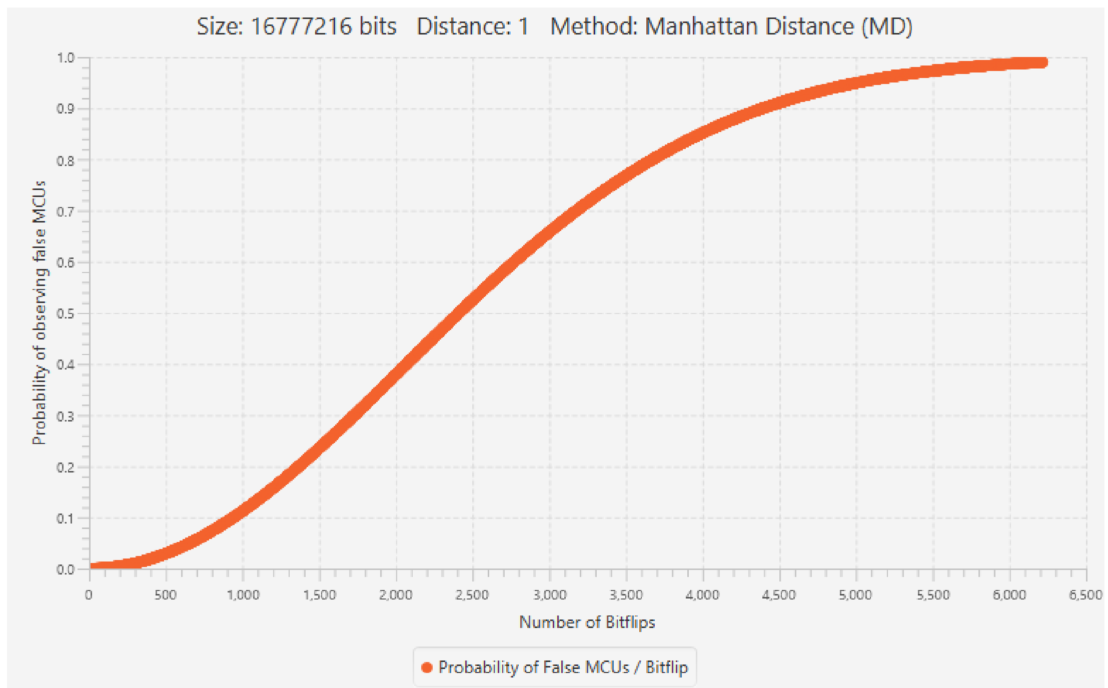

Figura 37. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 2MX8 bits y D = 1. Método MD.

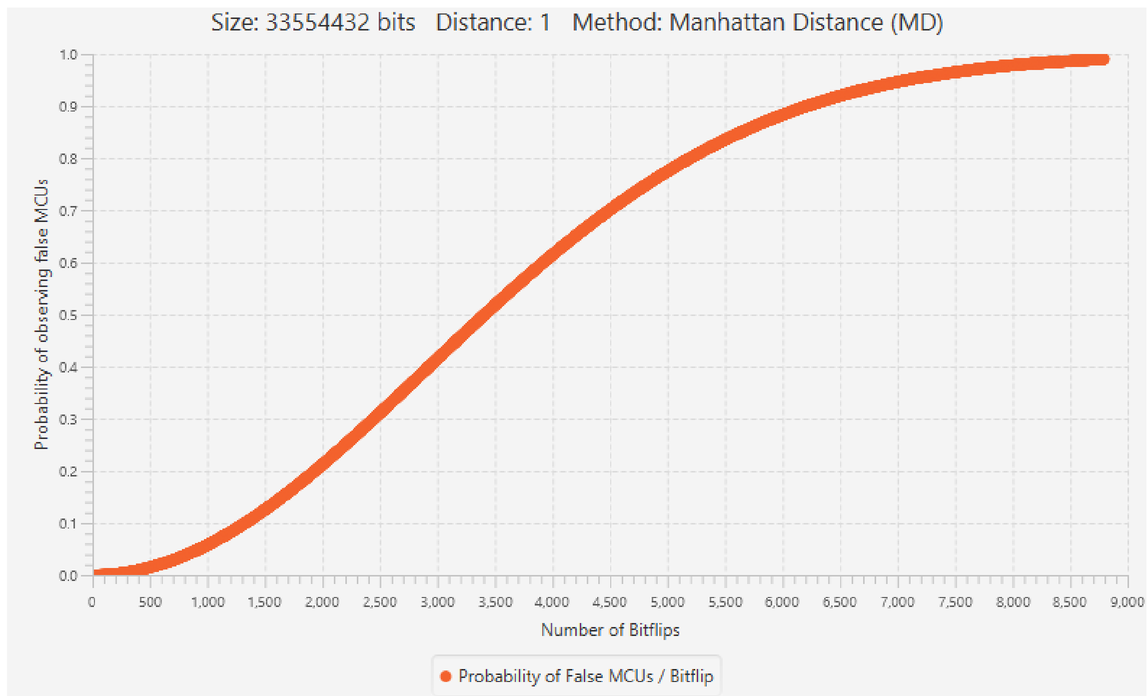

Figura 38. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 4MX8 bits y D = 1. Método MD.

Finalmente, en las Figura 37 yFigura 38, se compara lo obtenido para una memoria de tamaño 2Mx8 bits y otra cuyo tamaño es el doble (4Mx8 bits). Al aumentar el tamaño de la memoria, se aumenta el número de direcciones que se pueden ver afectadas por los SEEs. Por tanto, para un mismo número de bitflips repartidos entre toda la memoria, es menos probable que se vean afectadas dos celdas vecinas y, por lo tanto, aumenta el número de bitflips necesarios para llegar a la misma probabilidad de aparición de falsos MCUs.

En todos estos casos se puede observar que la curva de probabilidad tiene una forma similar, ya que la pantalla en la que se muestra se adapta de forma proporcional a la cantidad de resultados que ha de mostrar. Esto ayuda enormemente a ver la dependencia que tiene la aparición de los falsos MCUs con el tamaño de la memoria, el criterio de agrupación elegido y distancia que el investigador elija para agrupar bitflips en MCUs.

False Multiple Cell Upsets (MCUs) estimator

Please introduce the following parameters

Maximum probability of false MCUs occurrence

0.5

The maximum number of bitflips you should observe in your experiment is:
1706

Obtain result

Figura 39. Obtención del número máximo de bitflips que se deben observar para que la probabilidad de aparición de falsos MCUs sea menor o igual que *p.*

Adicionalmente, se ha implementado en la aplicación una funcionalidad de apoyo que consiste en que, dada una probabilidad p_0 de aparición de falsos MCUs, la aplicación devuelve el número de bitflips que se deben observar a partir de los cuales esta probabilidad es mayor o igual que p_0 (Figura 39). Esto servirá de apoyo para encontrar de forma rápida exactamente cuál es el número de bitflips que, como mucho se deben observar un experimento, si se desea moderar la probabilidad de aparición de falsos MCUs por debajo de un umbral deseado p_0.

4.2 Estudio de experimentos previos con memorias en aceleradores de partículas

Finalmente, en esta sección se comentan resultados experimentales realizados en investigaciones previas [28] – [30] y se compararán con los datos obtenidos de la aplicación. Al compararlos el objetivo principal es valorar si estos experimentos se realizaron bajo condiciones de alta probabilidad de falsos MCUs comprobando el número de bitflips observados en los mismos y comparándolos con los datos obtenidos de la aplicación.

En todos los casos se utilizaron memorias de 16 Mbits de tamaño, una distancia $D = 3$ y MD como método de agrupación. Al introducir estos datos en la aplicación se obtiene la curva de probabilidad representada en la Figura 41.

4.2.1 Primer experimento

El primero de los experimentos analizados [28], realizado en Mayo de 2015, se centró en el estudio de la sensibilidad a los SEEs de las memorias de tipo SRAM que sean comerciales o COTS (“Commercial-Off-The-Shelf”), ya que cada vez son más populares en los ámbitos aeronáuticos y espaciales en los que la radiación está muy presente. El motivo del uso de este tipo de memorias es su bajo coste y su fiabilidad ante estos eventos gracias a los códigos de corrección de errores ya mencionados en la sección *1.3.3* Mitigación por redundancia.

Muchos dispositivos actuales permiten el uso de la técnica “Dynamic Voltage Scaling” (DVS) para un uso óptimo de la energía. El objetivo del DVS es conseguir reducir el consumo de energía de los componentes que más consuman cuando no necesiten tanto voltaje y de esta forma ahorrar esa energía.

Como consecuencia, y como ya se ha observado en previos estudios [31] [32], al reducirse el voltaje, se aumenta la probabilidad de aparición de SBUs/MCUs, por lo tanto, tanto en este como en el resto de los experimentos analizados, se realizaron diferentes rondas de irradiación con diferentes voltajes para estudiar cómo afecta este factor a la aparición de SBUs y MCUs.

Voltaje (V)	SBU	2-bit	3-bit	4-bit	5-bit	6-bit	7-bit	8-bit	9-bit	10-bit
0.50	1645	96	12	8	2	0	0	0	0	1
0.60	1385	89	10	3	0	0	0	0	0	0
0.70	1215	96	13	3	1	1	0	0	0	0
0.80	1065	97	15	4	0	0	0	0	0	0
0.90	876	99	12	4	0	0	0	0	0	0
1.00	734	79	16	5	0	1	1	0	0	0
1.20	623	69	7	0	0	0	0	0	0	0
3.30	86	12	2	1	0	0	0	0	0	0

Tabla 1. Resultados del experimento [28].

Las pruebas de este primer experimento se realizaron en varias rondas de irradiación con neutrones a 14.2 MeV en el "*GEnerator of NEutrons Pulsed and Intense*" (GENEPI2) con una memoria de 90-nm fabricada por Infineon Technologies (CY62167EV30LL-45ZXI). Los resultados obtenidos se muestran en la Tabla 1.

En este caso, los datos aportados no muestran directamente el número de bitflips por cada ronda, pero sí el número total de SBUs y MCUs con diversas multiplicidades (de 2 a 10). Por tanto, para obtener el número de bitflips de cada experimento solo hace falta sumar todos los eventos multiplicados por su multiplicidad correspondiente obteniendo así el resultado de la Tabla 2.

Voltaje (V)	Número total de bitflips
0.50	1925
0.60	1605
0.70	1469
0.80	1320
0.90	1126
1.00	973
1.20	782
3.30	120

Tabla 2. Número total de bitflips observados en las rondas experimentales de [28].

Adicionalmente, cabe mencionar que la memoria utilizada estaba dividida en dos módulos iguales y que, debido a un error, uno de ellos quedó inoperativo, por lo que los resultados obtenidos serían los correspondientes a los de una memoria 1Mx8 bits. Al introducir estos parámetros en la aplicación, se obtienen los resultados de la Figura 40.

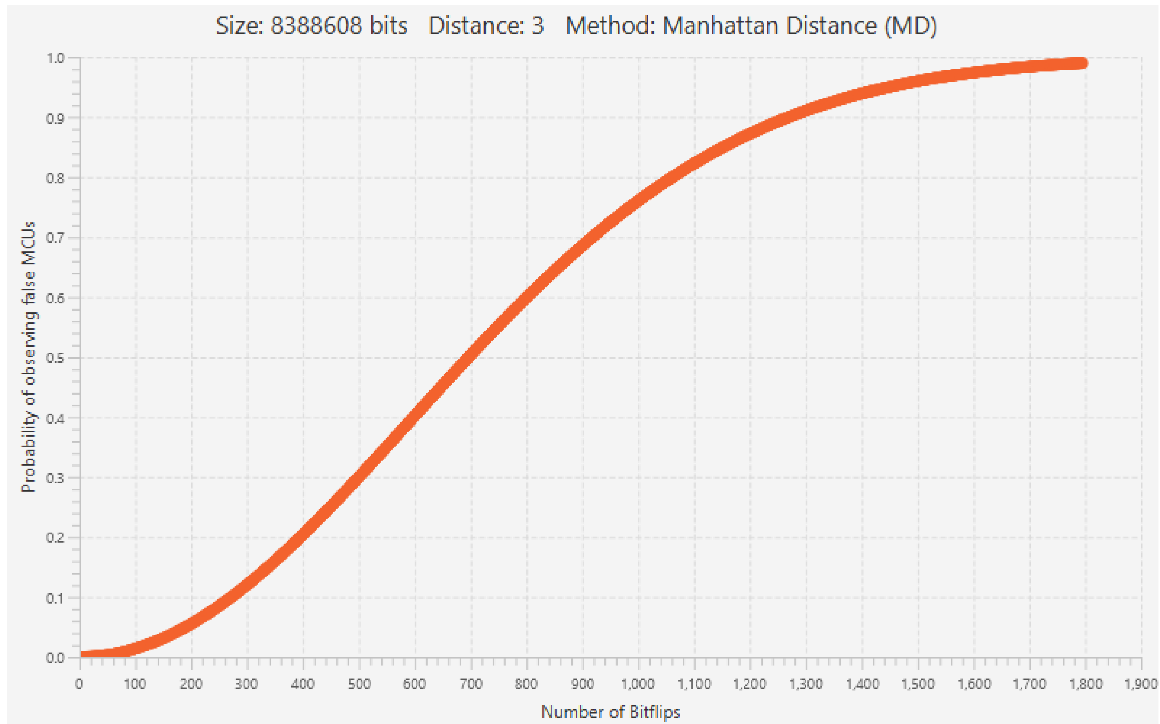

Figura 40. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 1MX8 bits y D = 3. Método MD.

Como se puede observar en la Figura 40, la probabilidad máxima de aparición de falsos MCUs (99%) se produce a partir de los 1795 bitflips, lo cual sitúa la mayoría de las rondas de irradiación por debajo de este umbral. En este caso se decidió realizar rondas de irradiación de muy poco tiempo para optimizar el escaso tiempo disponible del que se disponía en el acelerador. La consecuencia de ir "a ciegas" fue obtener, en algunas de las rondas, muy pocos resultados, lo cual limitó la calidad de las conclusiones obtenidas en ese experimento al no ser estadísticamente muy fuertes.

Esto se observa de forma muy clara en las pruebas correspondientes a 1.2V y a 3.3V, donde los eventos múltiples de más de 3 bits son prácticamente inexistentes. Si la aplicación que se ha desarrollado en este Trabajo de Fin de Grado hubiese estado disponible antes de la realización del experimento, habría sido de gran ayuda para que los investigadores pudieran ajustar la extensión del experimento para una utilización óptima del escaso tiempo del que se disponía en el acelerador.

4.2.2 Segundo experimento

La idea detrás de este segundo experimento fue reforzar los resultados obtenidos del primero evitando el error que se produjo en aquella ocasión. Estas pruebas [29] tuvieron lugar en Mayo del 2017 en el acelerador GENEPI2 exponiendo las memorias a flujos de neutrones a con una energía media de 14.2 MeV en un rango de 2.00×10^7 a $2.41 \times 10^7\ n/cm^2/s$. En este caso se usaron 3 memorias de Infineon Technologies de 65-nm (CY 62167GE30-4 5ZXI), 90-nm (CY 62167EV 30LL 45ZXI) y 130-nm (CY 62167DV 30LL 55ZXI) respectivamente.

Duración (mins.)	Voltaje (V)	Bitflips
3	0.72	4393
3	0.8	3876
3	0.9	3515
3	1	3163
3	1.1	2976
3	1.2	2878
3	1.3	2565
3	1.4	2342
3	1.5	2244
3	1.7	1670
3	2	1188
3	2.5	824
3	2.8	543
3	3.17	516
5	3.17	4160
3	3.17	475
5	3.17	3929

Tabla 3. Resultados del experimento [29] para la memoria de 130-nm.

En la Tabla 3 se muestran los resultados para la memoria de 130-nm, los cuales fueron muy similares a los de la memoria de 90-nm [29]. A continuación, en la Tabla 4 se muestran los resultados de la memoria de 65-nm, que era tecnológicamente muy diferente a las otras 2 y que, además, implementaba un código de corrección de errores que hubo que desactivar para realizar los experimentos.

Duración (mins.)	Voltaje (V)	Bitflips
2	0.8	1276
2	0.9	1109
2	1	876
2	1.1	885
5	1.3	1854
2	1.5	734
5	2	1756
5	2.5	1708
5	3.17	1850

Tabla 4. Resultados del experimento [29] para la memoria de 65-nm.

Al comparar estos datos con los obtenidos de la aplicación en la Figura 41, se puede observar que a partir de 2538 bitflips, la probabilidad de encontrarse con falsos MCUs es del 99% o superior. Por lo tanto, en el caso de la memoria de 130-nm la situación es la contraria a la del primer experimento, en la cual el resultado de desconocer la probabilidad de falsos MCUs conllevó que el investigador no pudiese ajustar la duración del experimento a una donde hubiese una probabilidad razonable de aparición de falsos MCUs y que, en 9 ocasiones se superase ese umbral de 2538 bitflips, dándose cifras muy superiores en algunas de ellas.

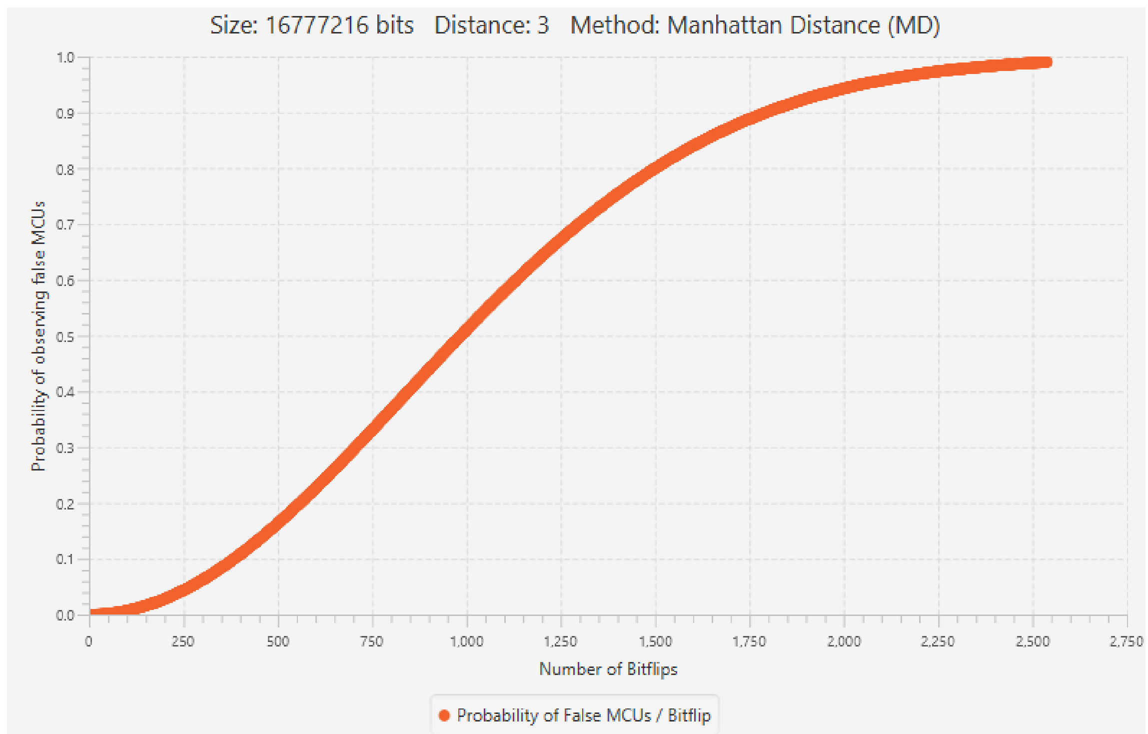

Figura 41. Análisis de la probabilidad de aparición de falsos MCUs, para una memoria de tamaño = 2MX8 bits y D = 3. Método MD.

Alternativamente, al contrastar los datos de la memoria de 65-nm, todos los resultados quedan en la zona de la gráfica donde esta probabilidad es más baja ya que el caso en el que más bitflips se observan son 1854 (para un voltaje de 1.3V) la probabilidad es de ocurrencia de falsos MCUs en ese caso fue del 91,4%.

4.2.3 Tercer experimento

Como se ha observado en los anteriores experimentos, el voltaje de alimentación es un factor a tener en cuenta ya que afecta a la sensibilidad de la memoria frente a la radiación. Por ello, este tercer experimento [30] se centra en analizar el impacto que el DVS tiene en la fiabilidad de las SRAMs.

Las pruebas [30] fueron realizadas en Marzo del 2021 en el Institut Laue-Langevin (ILL) usando el acelerador de neutrones térmicos "Thermal and Epithermal Neutron Irradiation Station" (TENIS) usando las 3 mismas memorias de Infineon Technologies que en el experimento 2. Para cada memoria se realizaron pruebas con rangos de voltajes entre 0.5 y 3.23 V exponiéndolas a un flujo de $2.86 \times 10^{9}\ n/cm^{2}/s$. Los resultados se muestran en la Tabla 5 y la Tabla 6.

Voltaje (V)	Bitflips observados		
	65-nm	90-nm	130-nm
3.23	184	716	519
3	189	744	562
2.5	174	696	552
2	153	804	564
1.5	160	766	810
1	446	1044	1606
0.9	565	1392	2110
0.85	710	1498	2396
0.8	975	1756	2938
0.75	1148	1976	3474
0.7	-	2564	4230
0.65	-	3252	4862
0.6	-	3752	6098
0.55	-	4834	7800
0.5	-	6004	9296

Tabla 5. Resultados del experimento [30] (1).

Voltaje (V)	Bitflips observados		
	65-nm	90-nm	130-nm
3.23	181	624	510
3	182	646	478
2.5	175	646	428
2	177	654	750
1.5	161	682	788
1	437	926	1620
0.9	645	1254	2042
0.85	760	1438	2376
0.8	935	1722	2852
0.75	1228	2056	3318
0.7	-	2610	3860
0.65	-	3090	4714
0.6	-	3848	5956
0.55	-	4972	7320
0.5	-	6066	9142

Tabla 6. Resultados del experimento [30] (2).

En las tres columnas de la derecha de la Tabla 5 y la Tabla 6 se representan el número de bitflips observados en cada prueba que se realizó. Al igual que en los 2 experimentos anteriores, si esta aplicación hubiese estado disponible antes de la realización de este experimento, los investigadores podrían haber consultado de forma preventiva cuál es el máximo número de bitflips que se deberían observar para moderar la probabilidad de aparición de falsos MCUs bajo un cierto umbral. Sin embargo, en aquella ocasión (y en las anteriores, que ya se han explicado), estas pruebas fueron realizadas a ciegas.

En este experimento, la situación es la misma a la de la memoria de 130-nm del segundo experimento, donde un gran porcentaje de las pruebas realizadas superan la probabilidad del 99% de falsos MCUs (que, igual que en el experimento 2, se consigue con 2538 bitflips). Por lo tanto, el investigador podría haber utilizado, en ambas situaciones, la información de la aplicación para moderar el tiempo de exposición frente a la radiación y así obtener datos experimentales con menos errores por ronda de irradiación, pero con una probabilidad de aparición de falsos eventos múltiples totalmente controlada. En aquel caso, esta situación se corrigió a posteriori, utilizando las ecuaciones (5) y (6) (explicadas en la sección 2.2) para estimar el número de falsos MCUs y restándolo al número total de MCUs observados. No obstante, esta no resulta una metodología ideal para un análisis exhaustivo de los resultados, ya que, si bien así se puede estimar con precisión el número de MCUs que verdaderamente ocurrieron en el experimento, la existencia de falsos MCUs limita la capacidad de un análisis más profundo de los eventos que ocurrieron, como por ejemplo su “forma” (es decir, si afectaron a celdas organizadas horizontal o verticalmente), u otros aspectos de interés.

5. Conclusiones de este Trabajo de Fin de Grado

En este Trabajo de Fin de Grado se ha propuesto una nueva herramienta que permitirá a los investigadores obtener una previsión de la probabilidad de aparición de falsos eventos múltiples de tipo MCU en memorias expuestas a radiación natural.

Para evaluar la fiabilidad de estas memorias frente a los efectos causados por la radiación, es habitual hacer uso de aceleradores de partículas. El uso de estas instalaciones requiere un esfuerzo económico considerable y la aparición de estos falsos MCUs dificultan un uso óptimo de la fuente de radiación disponible ya que entorpecen la obtención de unos resultados fiables. Por lo tanto, esta herramienta proveerá al investigador información de apoyo que servirá para ajustar el tiempo de exposición de la memoria al requerido para cada experimento, teniendo bajo control la probabilidad de aparición de dichos eventos múltiples. De esta forma se conseguirá optimizar el uso del acelerador.

Tal y como se ha observado en la sección *4.2 Estudio de experimentos previos con memorias en aceleradores de partículas,* en experimentos previos donde los investigadores tuvieron que ir “a ciegas”, resultó en que muchas de las rondas de irradiación superaron el umbral del 99% de probabilidad de aparición de falsos MCUs. Si bien esto no invalida por completo los resultados obtenidos, sí que provoca que sean menos fiables. Con este factor en mente, la herramienta podrá servir de apoyo a los investigadores en futuros experimentos para reducir el impacto que tienen los falsos MCUs en los resultados. Los investigadores podrán consultar la información proporcionada por la aplicación antes de la realización del experimento y, de esta forma, ajustar el tiempo de exposición al tiempo requerido para cada prueba sin obtener ni demasiados resultados en los que haya falsos MCUs, ni escasos resultados que requieran volver a realizar rondas de irradiación para reforzarlos, invirtiendo así tiempo adicional.

La herramienta se ha desarrollado en Java haciendo uso de las librerías de JavaFX destinadas al desarrollo web. La aplicación contiene la función principal del cálculo de falsos MCUs además de funcionales adicionales para un mejor acceso a la información como la posibilidad de hacer zoom en la gráfica donde se muestran los datos y un buscador para encontrar rápidamente un valor dentro de la misma gráfica.

Gracias a la herramienta `jpackage`, y como se expone en la sección 3.7 Portabilidad de la aplicación, se ha creado un instalador para facilitar la portabilidad de la aplicación. Tanto este instalador como el código de la aplicación se pueden encontrar en la web de GitHub https://github.com/jalguacilgon/TFG-MemoryRadiation-JesusAlguacilGonzalez. Para la obtención del instalador es necesario descargar el archivo “MemoryRadiation-1.6.2.exe” que contiene el fichero ejecutable para plataformas Windows. El código usado se encuentra bajo la carpeta “TFG” dentro del directorio.

Finalmente cabe mencionar que este trabajo se enmarca en un proyecto de investigación del Plan Nacional I+D+i del año 2020, con referencia PID2020-112916GB-I00, actualmente realizado por el Grupo en Gestión del Hardware Dinámicamente Reconfigurable (GHADIR) de la Facultad de Informática de la UCM, y será utilizado en los experimentos futuros que haga el grupo de investigación en este proyecto. También se pondrá a disposición de la comunidad científica en la web del grupo para su libre utilización.

5. Conclusions of this bachelor’s thesis

In this bachelor’s thesis, a new tool has been proposed that will allow researchers to know in advance the probability of appearance of the so-called “false MCUs” in memories, when they are exposed to natural radiation.

To evaluate the reliability of these memories against the effects caused by the radiation, it is very usual to use particle accelerators. The use of these facilities requires a considerable economic effort and the appearance of said false MCUs complicates the optimal use of the available radiation source because they hinder obtaining reliable results. Thus, this tool will provide the researcher supporting information which will be helpful to adjust the memory’s exposure time to the time required for each experiment, maintaining the probability of false MCUs appearance under control. By doing this, it will be possible to optimize the use of the accelerator.

As shown in Section *4.2 Estudio de experimentos previos con memorias en aceleradores de partículas*, in previous radiation ground experiments where researchers were “blindfolded”, many of the irradiation rounds that were performed surpassed the 99% probability of appearance of said false MCUs. Although this does not invalidate the results, it considerably reduces their statistical validity. With this fact in mind, this tool may provide support to researchers in future experiments to reduce the impact of false MCUs in the results. Researchers will be able to consult the information provided by the application beforehand to adjust the exposure time to the time required for each test without obtaining too many results in which false MCUs are present, nor too few ones that require additional irradiation rounds to strengthen the results and thus, investing additional time.

The tool has been developed in Java language using JavaFX libraries conceived for web development. It contains the main functionality of false MCUs calculations plus some extra functionalities to provide better access to the information such as a built-in zoom-in/zoom-out option for the plots shown to the user and a search functionality to easily find a value within the same graph.

Thanks to the `jpackage` tool, and as explained in *3.7 Portabilidad de la aplicación*, an installer to facilitate the app’s portability has also been created. This installer, as well as the code developed, can be found in the GitHub’s repository of this BsC. Thesis [5]. To obtain the installer, it is required to download the file under the name “MemoryRadiation-1.6.2.exe” which contains the executable file for Windows platforms. The developed code can be found under the “TFG” folder.

To conclude, it is worth mentioning that the context of this BsC. Thesis is a research project belonging to the 2020 Spanish “Plan Nacional I+D+i”, financed by the Spanish Ministry of Science, with reference PID2020-112916GB-I00. This project is now under development by the “Grupo en Gestión del Hardware Dinámicamente Reconfigurable (GHADIR)” of the Computer Science Faculty at UCM, and this tool will be used by this research team in future experiments within the scope of this project. It will also be available to the Community in the group’s website so any researcher will be free of using it.

[5] https://github.com/jalguacilgon/TFG-MemoryRadiation-JesusAlguacilGonzalez

Bibliografía

[1] A. Dixit y A. Wood, «The impact of new technology on soft error rates,» de *International Reliability Physics Symposium*, Monterey, CA, USA, 2011.

[2] S. Tsai y P. Schmied, «Interleaving and Error-Burst Distribution,» *IEEE Transactions on Communications,* vol. 20, nº 3, pp. 291-296, 1972.

[3] S. Baeg, S. Wen y R. Wong, «SRAM Interleaving Distance Selection With a Soft Error Failure Model,» *IEEE Transactions on Nuclear Science,* vol. 56, nº 4, pp. 2111-2118, 2009.

[4] M. Wirthlin, D. Lee, G. Swift y H. Quinn, «A Method and CaseStudy on Identifying Physically Adjacent Multiple-Cell Upsets Using28-nm, Interleaved and SECDED-Protected Arrays,» *IEEE Transactions on Nuclear Science,* vol. 61, nº 6, pp. 3080-3087, 2014.

[5] R. W. Hamming, «Error detecting and error correcting codes,» *The Bell System Technical Journal,* vol. 29, nº 2, pp. 147-160, 1950.

[6] R. Velazco y F. J. Franco, «Single Event Effects on Digital Integrated Circuits: Origins and Mitigation Techniques,» de *2007 IEEE International Symposium on Industrial Electronics*, Vigo, Spain, 2007.

[7] J. F. Ziegler y H. Puchner, «SER – History, Trends and Challenges. A guide for Designing with Memory ICs,» Cypress, 2004.

[8] B. E. Pritchard, G. M. Swift y A. H. Johnston, «Radiation effects predicted, observed, and compared for spacecraft systems,» de *IEEE Radiation Effects Data Workshop*, Phoenix, AZ, USA, 2002.

[9] J. Olsen, P. E. Becher, P. B. Fynbo, P. Raaby y J. Schultz, «Neutron-induced single event upsets in static RAMS observed a 10 km flight attitude,» *IEEE Transactions on Nuclear Science,* vol. 40, nº 2, pp. 74-77, 1993.

[10] C. Enguehard y J.-D. Graton, «Electronic Voting: the Devil is in the Details,» 2008.

[11] M. Rao, Extensive Air Showers, World Scientific, 1998.

[12] R. Baumann, «Single-Event Effects in Advanced CMOS Technology,» de *Section II of the short course in the frame of the 2005 IEEE Nuclear and Space Radiation Effects Conference*, Seattle, USA, July 2005.

[13] H. Kumada, «8.13 - Neutron Sources,» de *Comprehensive Biomedical Physics*, Oxford, Elsevier, 2014, pp. 197-217.

[14] G. Gasiot, V. Ferlet-Cavrois, J. Baggio, P. Roche, P. Flatresse, A. Guyot, P. Morel, O. Bersillon y J. du Port de Pontcharra, «SEU sensitivity of bulk and SOI

technologies to 14-MeV neutrons,» *IEEE Transactions on Nuclear Science,* vol. 49, nº 6, pp. 3032-3037, 2002.

[15] P. Ramos, V. Vargas, M. Baylac, F. Villa, S. Rey, J. A. Clemente, N.-E. Zergainoh, J.-F. Méhaut y R. Velazco, «Evaluating the SEE Sensitivity of a 45 nm SOI Multi-Core Processor Due to 14 MeV Neutrons,» *IEEE Transactions on Nuclear Science,* vol. 63, nº 4, pp. 2193 - 2200, 2016.

[16] J. R. Schwank, V. Ferlet-Cavrois, M. R. Shaneyfelt, P. Shaneyfelt y P. E. Dodd, «Radiation effects in SOI technologies,» *IEEE Transactions on Nuclear Science,* vol. 50, nº 3, pp. 522-538, 2003.

[17] P. Dodd, M. Shaneyfelt, D. Walsh, J. Schwank, G. Hash, R. Loemker, B. Draper y P. Winokur, «Single-event upset and snapback in silicon-on-insulator devices and integrated circuits,» *IEEE Transactions on Nuclear Science,* vol. 47, nº 6, pp. 2165-2174, 2000.

[18] T. Calin, M. Nicolaidis y R. Velazco, «Upset hardened memory design for submicron CMOS technology,» *IEEE Transactions on Nuclear Science,* vol. 43, nº 6, pp. 2874-2878, 1996.

[19] J. Furuta, K. Kobayashi y H. Onodera, «An area/delay efficient dual-modular flip-flop with higher SEU/SET immunity,» IEICE Transactions, vol. 93, nº 3, pp. 340-346, 2010.

[20] S. Lin y D. J. Costello, «Error control coding: fundamentals and applications,» Upper Saddle River, NJ: Pearson/Prentice Hall, 2004.

[21] A. Neale, M. Jonkman y M. Sachdev, «Adjacent-MBU-Tolerant SEC-DED-TAEC-yAED Codes for Embedded SRAMs,» *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 62, nº 4, pp. 387-391, 2015.

[22] G. He, S. Zheng y N. Jing, «A Hierarchical Scrubbing Technique for SEU Mitigation on SRAM-Based FPGAs,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 28, nº 10, pp. 2134-2145, 2020.

[23] F. J. Franco, J. A. Clemente, M. Baylac, S. Rey, F. Villa, H. Mecha, J. A. Agapito, H. Puchner, G. Hubert y R. Velazco, «Statistical Deviations From the Theoretical Only-SBU Model to Estimate MCU,» *IEEE Transactions on Nuclear Science,* vol. 64, nº 8, pp. 2152-2160, 2017.

[24] Z. E. Schnabel, «The Estimation of the Total Fish Population of a Lake,» *American Mathematical Monthly,* vol. 45, pp. 348–352, 1938.

[25] M. Abramson y W. O. Moser, «More Birthday Surprises,» *American Mathematical Monthly,* vol. 77, pp. 856–858, 1970.

[26] F. J. Franco, J. A. Clemente, G. Korkian, J. C. Fabero, H. Mecha y R. Velazco, «Inherent Uncertainty in the Determination of Multiple Event Cross Sections in Radiation Tests,» *IEEE Transactions on Nuclear Science,* vol. 67, nº 7, pp. 1547 - 1554, 2020.

[27] F. J. Franco, J. A. Clemente, H. Mecha y R. Velazco, «Influence of Randomness During the Interpretation of Results From Single-Event Experiments on SRAMs,» *IEEE Transactions on Device and Materials Reliability,* vol. 19, nº 1, pp. 104 - 111, 2019.

[28] (datos obtenidos en Junio de 2015 en el Laboratorio LPSC con neutrones a 14.2 MeV con una memoria de 1Mx8 bits) J. A. Clemente, G. Hubert, F. J. Franco, F. Villa, M. Baylac, H. Mecha, H. Puchner y R. Velazco, «Sensitivity Characterization of a COTS 90-nm SRAM at Ultra Low Bias Voltage,» IEEE Transactions on Nuclear Science (TNS), vol. 64, nº 8, pp. 2188-2195, 2017.

[29] (datos obtenidos en Junio de 2017 en el LPSC con neutrones a 14.2 MeV con 3 memorias de 2Mx8 bits) J. A. Clemente, G. Hubert, J. Fraire, F. J. Franco, F. Villa, S. Rey, M. Baylac, H. Puchner, H. Mecha y R. Velazco, «SEU Characterization of Three Successive Generations of COTS SRAMs at Ultralow Bias Voltage to 14.2-MeV Neutrons,» IEEE Transactions on Nuclear Science (TNS), vol. 65, nº 8, pp. 1858-1865, 2018.

[30] (datos obtenidos en marzo de 2021 en el acelerador TENIS del laboratorio ILL bajo neutrones térmicos con 3 memorias de 2Mx8 bits) M. Rezaei, F. J. Franco, J. C. Fabero, H. Mecha, H. Puchner y J. A. Clemente, «Impact of DVS on Power Consumption and SEE Sensitivity of COTS Volatile SRAMs,» de 22nd IEEE Latin-American Test Symposium (LATS), 2021, pp.1-6.

[31] R. C. Baumann, «Radiation-induced soft errors in advanced semiconductor technologies,» *IEEE Transactions on Device and Materials Reliability,* vol. 5, nº 3, pp. 305-316, 2005.

[32] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong y S. Borkar, «Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-/spl mu/m to 90-nm generation,» de *IEEE International Electron Devices Meeting*, Washington, DC, USA, 2003.