Propuesta de Tesis de Grado de Ingeniería en Informática

Detección de Deadlocks en Rust en tiempo de compilación mediante Redes de Petri

Director: Ing. Pablo A. Deymonnaz

Alumno: Horacio Lisdero Scaffino, *(Padrón # 100.132)*
hlisdero@fi.uba.ar

Facultad de Ingeniería, Universidad de Buenos Aires

6 de marzo de 2023

Índice

1. Introducción

1.1. El problema de la correctitud en programación concurrente

En el área de computación concurrente, uno de los desafíos principales es probar la correctitud de un programa concurrente. A diferencia de un programa secuencial donde para cada entrada se obtiene siempre la misma salida, en un programa concurrente la salida puede depender de cómo se intercalaron las instrucciones de los diferentes procesos o hilos durante la ejecución.

La correctitud de un programa concurrente se define entonces en términos de propiedades del cómputo realizado y no solamente en términos del resultado obtenido. En la bibliografía [Ben-Ari, 2006, Coulouris et al., 2012, van Steen and Tanenbaum, 2017] se definen dos tipos de propiedades de correctitud:

- Propiedades de *safety*: Propiedades que se deben cumplir *siempre*.
- Propiedades de *liveness*: Propiedades que se deben cumplir *eventualmente*.

Dos de las propiedades de tipo *safety* deseables en un programa concurrente son:

- **Exclusión mutua**: dos procesos no deben acceder a recursos compartidos al mismo tiempo.
- **Ausencia de *deadlock***: un sistema en ejecución debe poder continuar realizando su tarea, es decir, avanzar produciendo trabajo útil.

Usualmente se utilizan primitivas de sincronización tales como mutexes, semáforos, monitores y *condition variables* para implementar el acceso coordinado de los procesos o hilos a los recursos compartidos. No obstante, el uso correcto de estas primitivas es difícil de lograr en la práctica y se pueden introducir errores difíciles de detectar y corregir. Actualmente la mayoría de lenguajes de uso general, ya sean compilados o interpretados, no permiten detectar estos errores en todos los casos.

Dada la creciente importancia de la programación concurrente debida a la proliferación de sistemas de hardware multihilo y multiproceso, reducir el número de *bugs* ligados a la sincronización de los procesos o hilos es de vital importancia para la industria. Evitar los *deadlocks* es un requerimiento ineludible en el desarrollo, especialmente en sistemas embebidos y sistemas de misión crítica como vehículos autónomos o aeronaves.

1.2. Redes de Petri

Las redes de Petri son una herramienta gráfica y matemática ampliamente utilizada para describir sistemas distribuidos, introducidas por el investigador alemán Carl Adam Petri en su tesis

de doctorado [Petri, 1962]. Un resumen conciso de la teoría de redes de Petri, sus propiedades, análisis y aplicaciones se encuentra en [Murata, 1989].

Una red de Petri consiste en un grafo dirigido bipartito, el cual cuenta con dos tipos de nodos: lugares (*places*) y transiciones (*transitions*). Únicamente pueden existir arcos dirigidos entre un lugar y una transición o entre una transición y un lugar. A los lugares se les asignan marcas (*tokens*) los cuales representan el estado actual del sistema o un recurso en particular.

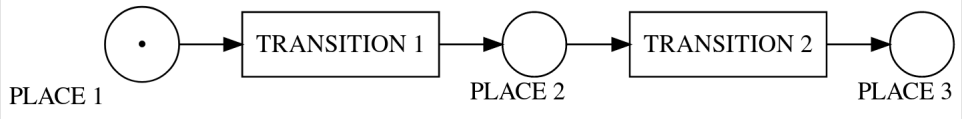

Figura 1: Ejemplo de una red de Petri. `PLACE 1` contiene un *token*.

Las transiciones se disparan siguiendo la siguiente regla:

- Se consume un *token* de los lugares cuyos arcos entran a la transición.
- Se crea un token en cada lugar al cual llega un arco saliente de la transición.

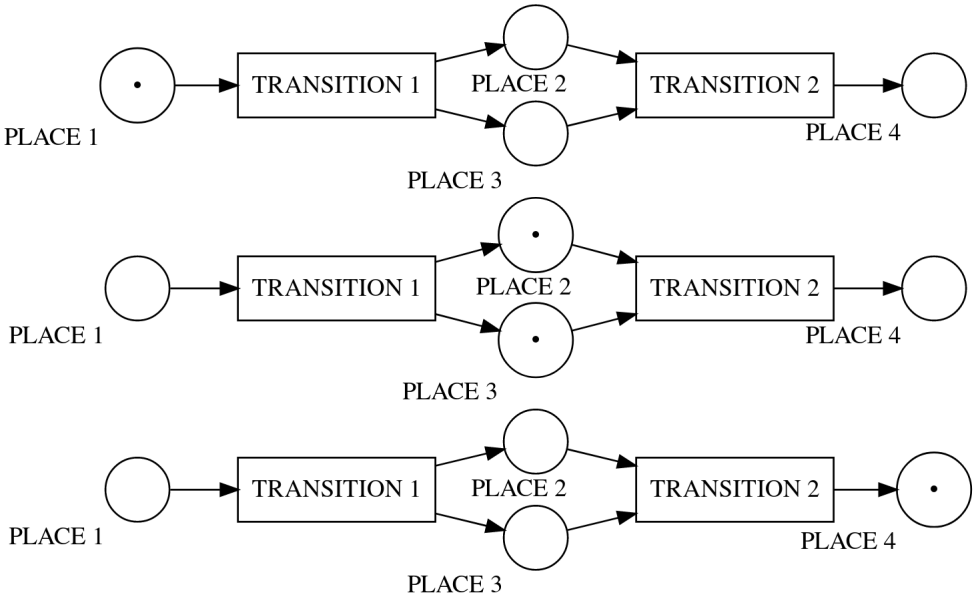

Figura 2: Ejemplo de disparo de transiciones. Primero se dispara la transición 1 y luego la transición 2.

Las redes de Petri pueden ser vistas como una versión generalizada de las máquinas de estado que permite modelar la concurrencia y el paralelismo. Su uso como método formal de validación de software está establecido desde finales de los años 1980 y existen redes de Petri que permiten

modelar primitivas de sincronización como enviar un mensaje o esperar a la recepción de un mensaje [Heiner, 1998]. Se pueden utilizar también para validar requerimientos de software expresados mediante casos de uso [Silva and dos Santos, 2004], para modelar procesos industriales [van der Aalst, 1994] o para especificar y analizar sistemas de tiempo real [Kavi et al., 2011].

Existen varias técnicas que permiten encontrar *deadlocks* en una red de Petri. El método más conocido se denomina análisis de alcance (*reachability analysis*) [Murata, 1989]. En este se construye un grafo dirigido que representa los estados posibles que la red de Petri puede alcanzar durante su ejecución. En este grafo cada nodo representa un estado posible de la red y cada arista representa una transición que permite pasar de un estado al otro. Se puede demostrar matemáticamente que la existencia de un nodo sin aristas salientes implica la existencia de un *deadlock* en la red de Petri.

La desventaja de este análisis es su elevado costo computacional, el cual crece de forma exponencial con la cantidad de nodos de la red de Petri cuando no se puede hacer uso de hipótesis o heurísticas adicionales. La complejidad computacional del llamado *reachability problem* es de hecho NP-completo para ciertas clases de redes de Petri. Sin embargo, existen clases de redes para las cuales la complejidad es menor, incluso polinomial. En [Esparza and Nielsen, 1994] se detallan los resultados teóricos más importantes obtenidos hasta 1998.

Además existen métodos alternativos para detección de *deadlocks* como el método basado en sifones (una estructura específica que ocurre en muchas redes de Petri) [Hu et al., 2011] y el método basado en jerarquías de abstracción [Kungas, 2005]. En particular, este último autor propone un método muy prometedor de orden polinomial para evitar el problema de la explosión de estados que subyace al algoritmo *naïve* de detección de *deadlocks*. A través de un algoritmo que abstrae una red de Petri dada a una representación más simple, se obtiene una jerarquía de redes de tamaño creciente para las cuales la verificación de ausencia de *deadlocks* resulta sustancialmente más rápida. Es, dicho de una forma burda, una estrategia del tipo "divide y vencerás"que verifica la ausencia de *deadlocks* en partes de la red para luego ir construyendo la verificación del todo final agregando partes a la red pequeña inicial.

1.3. Motivación

En el presente trabajo nos proponemos estudiar la detección de *deadlocks* y *lost signals* en el lenguaje de programación Rust. Se utilizará un modelo teórico basado en redes de Petri para encontrar los errores en el código fuente. Mediante una traducción del código fuente en tiempo de compilación, se obtendrá una red de Petri que luego podrá ser analizada mediante métodos de verificación de modelos para garantizar la ausencia de *deadlocks*.

El objetivo es contribuir a la comunidad de Rust aportando una primera versión de esta herramienta que podría luego ser extendida para soportar casos más complejos. Se busca que el uso de la herramienta sea lo más sencillo y accesible posible para fomentar su uso y aplicación a proyectos reales de software. Por esta razón la tesis contará con una primera implementación

del traductor que podrá ser utilizado como *plugin* del gestor de paquetes estándar de Rust *cargo*[1].

A largo plazo se podría incorporar la herramienta al compilador como un pase adicional opcional en el proceso de compilación. Este pase verificaría que no se pueden producir ciertas clases de *deadlocks*, lo que haría de Rust un lenguaje de programación aún más seguro y confiable.

Con el propósito de que el trabajo pueda ser leído por el mayor número de personas posible y que forme parte de la documentación de los respectivos repositorios de código publicados en GitHub, resulta imprescindible redactar el manuscrito en idioma inglés.

Por otra parte, nos proponemos implementar una funcionalidad de exportado de la red de Petri obtenida a partir del código fuente a formatos estandarizados a fin de facilitar la interoperabilidad con otras herramientas de verificación de modelos y visualización de redes de Petri. Como parte de la investigación previa se encontró que dos formatos son particularmente relevantes en este sentido:

- DOT[2]: un lenguaje de visualización de gráficos de código abierto, parte de la suite Graphviz [Gansner et al., 2015].
- Petri Net Markup Language (PNML)[3]: un lenguaje estandarizado para redes de Petri basado en XML [Jüngel et al., 2000]. Es parte del estándar ISO/IEC 15909-1:2019 que resultó de un trabajo arduo de muchos años para unificar la notación [Hillah and Petrucci, 2010].

2. Estado del arte / Literatura relacionada

2.1. El lenguaje de programación Rust

Uno de los lenguajes de programación modernos más prometedores para programación concurrente es Rust[4]. Su modelo de memoria basado en el concepto de *ownership* y su expresivo sistema de tipos permite eliminar una amplia variedad de errores relacionados al manejo de memoria y a la programación concurrente en tiempo de compilación:

- *Double free* [Klabnik and Nichols, 2022, Cap. 4.1]
- *Use-after-free* [Klabnik and Nichols, 2022, Cap. 4.1]
- Referencia colgante (*dangling pointers*) [Klabnik and Nichols, 2022, Cap. 4.2]
- *Data races* [Klabnik and Nichols, 2022, Cap. 4.2]

[1] https://doc.rust-lang.org/stable/cargo/
[2] https://graphviz.org/
[3] https://www.pnml.org/
[4] https://www.rust-lang.org/

- Pasaje de variables de tipo *non-thread-safe* entre hilos [Klabnik and Nichols, 2022, Cap. 16.4]

La importancia de estas ventajas para la industria no puede ser subestimada. Diversas investigaciones empíricas han llegado a la conclusión que 70 % de las vulnerabilidades encontradas en proyectos grandes en C/C++ ocurren debido a errores en el manejo de la memoria. Esta cifra elevada se puede observar en proyectos tales como Android [Stepanov, 2020], los componentes Bluetooth y media de Android [Stoep and Zhang, 2020], Chrome [The Chromium Projects, 2015], el componente CSS de Firefox [Hosfelt, 2019], iOS y macOS [Kehrer, 2019], productos de Microsoft [Miller, 2019, Fernandez, 2019] y Ubuntu [Gaynor, 2020].

Numerosas herramientas se han dedicado a tratar de resolver estas vulnerabilidades causadas por el uso incorrecto de la memoria en *codebases* ya establecidas. Sin embargo, su utilización conlleva una notable pérdida de performance y no todas las vulnerabilidades se pueden prevenir [Szekeres et al., 2013].

En los últimos años, varios proyectos de gran importancia en el ambiente Open Source han decidido incorporar Rust a fin de reducir el número de *bugs* relacionados al manejo de la memoria. Entre ellos podemos nombrar al Android Open Source Project [Stoep and Hines, 2021] y al kernel Linux que desde su versión 6.1 introduce soporte para programar componentes en Rust [Simone, 2022, Corbet, 2022]. Por otra parte, Meta aprueba y fomenta el uso de Rust como lenguaje para desarrollo *server-side* desde el 2022 [Garcia, 2022]. La popularidad del lenguaje Rust es innegable, ya que ha sido elegido durante 7 años consecutivos como el lenguaje de programación más querido por los programadores en la encuesta anual de Stack Overflow [Stack Overflow, 2022].

Cabe destacar que la generación de código en Rust incluye además una serie de mitigaciones a *exploits* de diversos tipos [Rust Project, 2023, Cap. 11]. Asimismo, si bien la librería estándar no está exenta de errores [Davidoff, 2018], los procesos de gobierno de código abierto y transparentes basados en el modelo RFC (*Requests for Comments*)[5] aseguran una mejora continua del lenguaje y su funcionalidad.

El ciclo de lanzamientos del compilador oficial de Rust, *rustc*, es por otra parte sumamente veloz. Cada 6 semanas se publica una nueva versión estable del compilador [Klabnik and Nichols, 2022, Appendix G]. Esto es posible gracias a un complejo sistema de tests automatizado que compila incluso todos los paquetes disponibles en *crates.io* mediante un programa llamado *crater* para verificar que la nueva versión del compilador no falla al compilar ni causa errores en los tests de los paquetes existentes [Albini, 2019].

Por estas razones Rust es una excelente opción para estudiar los *deadlocks* y *lost signals* porque se los puede estudiar por separado, sabiendo que otros errores ya son evitados en tiempo de compilación. La estabilidad y la seguridad del lenguaje proveen una base firme sobre la que construir una herramienta que detecte más errores en tiempo de compilación.

[5] https://rust-lang.github.io/rfcs/

2.2. Herramientas de verificación formal de código

Existen varias herramientas de verificación automática en Rust. Una primera aproximación recomendable es el resumen producido por Alastair Reid, investigador en Intel. En ella se lista explícitamente que la mayoría de las herramientas formales de verificación no soportan concurrencia [Reid, 2021].

El intérprete *Miri*[6] desarrollado por el *Rust project* en GitHub es un intérprete experimental para la representación intermedia del lenguaje Rust (*mid-level intermediate representation*, conocida comúnmente por la sigla "MIR") que permite ejecutar binarios de proyectos de *cargo* de forma granularizada, instrucción a instrucción, para verificar la ausencia de *Undefined Behaviour* (UB) y otros errores en el manejo de la memoria. Detecta *memory leaks*, accesos no alineados a memoria, *data races* y violaciones de precondiciones o invariantes en código marcado como *unsafe*.

Es conocido que en la actualidad las herramientas de verificación formal de software son aplicadas en unos pocos ámbitos muy específicos donde se requiere una demostración formal de correctitud del sistema. Usualmente se trata de sistemas donde la seguridad es un factor crítico. En [Reid et al., 2020] se discute la importancia de acercar las herramientas de verificación a los desarrolladores a través de un enfoque que buscar maximizar la relación costo-beneficio de su uso. Se propone mejorar la usabilidad de las herramientas existentes e incorporar su uso a la rutina del desarrollador partiendo de la base que la verificación puede ser vista como un tipo diferente de test unitario o de integración.

En [Toman et al., 2015] se introduce un verificador formal para Rust que no requiere modificaciones en el código fuente y se lo ejecuta sobre módulos de la librería estándar de Rust. Como resultado se detectaron errores en el uso de la memoria en código *unsafe* que tardaron meses en ser descubiertos de forma manual por el equipo de desarrollo. Esto ejemplifica la importancia del uso de herramientas de verificación automática para complementar las reviews manuales del código.

2.3. Detección de *deadlocks*

La detección de *deadlocks* es una de las estrategias clásicas para abordar este problema crucial en programación concurrente. Las estrategias restantes (*deadlock avoidance* y *deadlock prevention*) y algunos algoritmos se describen brevemente en [Singhal, 1989]. El problema principal con el enfoque de detectar los *deadlocks* antes de su aparición es probar que se detecta el tipo de *deadlock* deseado en todos los casos y que no se producen falsos negativos en el proceso. El enfoque basado en redes de Petri, tratándose de un método formal, satisface estas condiciones. No obstante, la dificultad de adopción radica mayoritariamente en la practicabilidad de la solución debido al gran número de estados posibles en un proyecto de software real.

[6] https://github.com/rust-lang/miri

En [Kavi et al., 2002] y [Moshtaghi, 2001] se describe una traducción de algunas de las primitivas de sincronización de la librería POSIX de threads (`pthread`) en C a redes de Petri. En particular se modela:

- La creación de threads con la función `pthread_create` y el manejo de la variable de tipo `pthread_t`.
- La operación de *thread join* con la función `pthread_join`.
- La operación de adquisición del lock de un mutex (`pthread_mutex_lock`) y su posterior desbloqueo (`pthread_mutex_unlock`).
- Las funciones `pthread_cond_wait` y `pthread_cond_signal` para manejo de *condition variables*.

En [Karatkevich and Grobelna, 2014] se propone un método para reducir el número de estados explorados durante la detección de *deadlocks* mediante el análisis de alcance. Estas heurísticas ayudan a mejorar la performance del enfoque basado en redes de Petri.

En su tesis de máster, [Meyer, 2020] establece las bases para una semántica de redes de Petri para el lenguaje de programación Rust. Se concentra no obstante en código con solo un hilo, limitándose a la detección de deadlocks causados por ejecutar la operación de *lock* dos veces sobre el mismo mutex en el hilo principal. Asimismo, el código disponible como parte del trabajo ya no es válido para la nueva versión del compilador *rustc*, ya que el funcionamiento interno del compilador cambió significativamente en los últimos dos años.

2.4. Bibliotecas de redes de Petri disponibles en Rust

Como parte del desarrollo de la traducción del código fuente a una red de Petri será necesario utilizar una biblioteca de redes de Petri para el lenguaje de programación Rust. Una búsqueda rápida en los paquetes disponibles en *crates.io*, GitHub y GitLab reveló que desafortunadamente no existe una biblioteca bien mantenida.

Se encontraron algunos simuladores de redes de Petri como:

- pns[7]: Programado en C. No ofrece la opción de exportar la red resultante a un formato estándar.
- PetriSim[8]: Un simulador antiguo para DOS/PC programado en Bordland Pascal.
- WOLFGANG[9]: Un editor de redes de Petri en Java, mantenido por el Departamento de Ciencias de la Computación de la Universidad de Freiburg, Alemania.

[7] https://gitlab.com/porky11/pns
[8] https://staff.um.edu.mt/jskl1/petrisim/index.html
[9] https://github.com/iig-uni-freiburg/WOLFGANG

Lamentablemente ninguno satisface los requerimientos del trabajo.

Dado que una red de Petri es un grafo, nos planteamos la posibilidad de usar una biblioteca de grafos y adaptarla a los objetivos del trabajo. Se encontraron dos bibliotecas de grafos en Rust:

- petgraph[10]: La biblioteca más utilizada para grafos en *crates.io*. Ofrece una opción de exportado al formato DOT.
- gamma[11]: Inestable y sin cambios desde el 2021. No ofrece la posibilidad de exportar el grafo.

Ninguna de las posibilidades satisface el requerimiento de exportar la red resultante al formato PNML. Además, de usarse una biblioteca para grafos, se deberían implementar las operaciones propias de una red de Petri como un *wrapper* alrededor de un grafo, lo que reduce la posibilidad de optimizaciones para nuestro caso de uso y dificulta la extensibilidad a largo plazo del proyecto.

En conclusión, es conveniente implementar una biblioteca de redes de Petri en Rust desde cero como un proyecto separado. Esto aporta una herramienta más a la comunidad que podría reutilizarse en trabajos posteriores.

3. Objetivos

El objetivo general de la tesis consiste en estudiar la posibilidad de extender el compilador de Rust para detectar *deadlocks* en tiempo de compilación debidos a un uso incorrecto de mutexes y señales perdidas debidas a un uso incorrecto de *condition variables*.

Los objetivos particulares son:

1. Diseñar e implementar un sistema de traducción del código Rust a una red de Petri.
2. Conectar la salida del sistema con un *model checker* para verificar la ausencia de *deadlocks* y *lost signals*. Se utilizarán entre otros los formatos de exportado especificados en 1.3.
3. Integrar la herramienta al ecosistema Rust, haciendo su uso lo más simple posible para el usuario.
   a) Implementar un plugin para el gestor de paquetes *cargo* y publicarlo.
   b) Documentar la herramienta y sus limitaciones. Esto incluye el manuscrito, el cual será redactado en inglés como se enunció en 1.3.

[10] https://docs.rs/petgraph/latest/petgraph/
[11] https://github.com/metamolecular/gamma

4. Cronograma de trabajo

Se establece el siguiente cronograma estimativo para el desarrollo de la tesis:

Tareas	Meses								
	1	2	3	4	5	6	7	8	9
Lectura de bibliografía									
Diseño del sistema									
Implementación de la biblioteca de redes de Petri									
Desarrollo									
Redacción del manuscrito									

La carga de trabajo estimada total es de alrededor de 900 horas.

- Lectura de bibliografía [130 horas]: Lectura de publicaciones científicas, libros de texto, artículos y documentación de herramientas existentes.
- Diseño del sistema [80 horas]: Familiarizarse con la arquitectura del compilador de Rust. Lectura de la documentación pertinente. Diseño de una solución extensible y confiable.
- Implementación de la biblioteca de redes de Petri [140 hs]: Considerando lo mencionado en 2.4, implementación de una biblioteca acorde a las necesidades de la solución.
- Desarrollo [400 hs]:
  - Implementación de la traducción de código fuente Rust a una red de Petri.
  - Desarrollo de un plugin para el gestor de paquetes *cargo*.
  - Incorporación de tests unitarios y de integración.
  - Documentación de la herramienta.
- Redacción del manuscrito de la tesis [150 horas]: El idioma a utilizar será inglés por lo explicado en 1.3.

Referencias

[Albini, 2019] Albini, P. (2019). RustFest Barcelona - Shipping a stable compiler every six weeks. https://www.youtube.com/watch?v=As1gXp5kX1M. Accedido: 2023-02-24.

[Ben-Ari, 2006] Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming*. Pearson Education, 2nd edition.

[Corbet, 2022] Corbet, J. (2022). The 6.1 kernel is out. https://lwn.net/Articles/917504/. Accedido: 2023-02-24.

[Coulouris et al., 2012] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012). *Distributed Systems, Concepts and Design*. Pearson Education, 5th edition.

[Davidoff, 2018] Davidoff, S. (2018). How Rust's standard library was vulnerable for years and nobody noticed. https://shnatsel.medium.com/how-rusts-standard-library-was-vulnerable-for-years-and-nobody-noticed-aebf0503c3d6. Accedido: 2023-02-20.

[Esparza and Nielsen, 1994] Esparza, J. and Nielsen, M. (1994). Decidability Issues for Petri Nets. *BRICS: Basic Research in Computer Science*, 1(8).

[Fernandez, 2019] Fernandez, S. (2019). A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. Accedido: 2023-02-24.

[Gansner et al., 2015] Gansner, E. R., Koutsofios, E., and North, S. C. (2015). *Drawing Graphs With Dot*.

[Garcia, 2022] Garcia, E. (2022). Programming languages endorsed for server-side use at Meta. https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/. Accedido: 2023-02-24.

[Gaynor, 2020] Gaynor, A. (2020). What science can tell us about C and C++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/. Accedido: 2023-02-24.

[Heiner, 1998] Heiner, M. (1998). Petri Net Based Software Validation - Prospects and Limitations. *ICSI Technical Report TR-92-022*.

[Hillah and Petrucci, 2010] Hillah, L. M. and Petrucci, L. (2010). Standardisation des réseaux de Petri : état de l'art et enjeux futurs. *Génie logiciel : le magazine de l'ingénierie du logiciel et des systèmes*, 93:5–10.

[Hosfelt, 2019] Hosfelt, D. (2019). Implications of Rewriting a Browser Component in Rust. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/. Accedido: 2023-02-24.

[Hu et al., 2011] Hu, W., Zhu, Y., and Lei, J. (2011). The Detection and Prevention of Deadlock in Petri Nets. *2011 International Conference on Physics Science and Technology (ICPST 2011)*, 22.

[Jüngel et al., 2000] Jüngel, M., Kindler, E., and Weber, M. (2000). The Petri Net Markup Language. *7. Workshop Algorithmen und Werkzeuge für Petrinetze*, 2-3.

[Karatkevich and Grobelna, 2014] Karatkevich, A. and Grobelna, I. (2014). Deadlock detection in Petri nets: one trace for one deadlock? *7th International Conference on Human System Interactions (HSI)*.

[Kavi et al., 2002] Kavi, K. M., Moshtaghi, A., and Chen, D.-J. (2002). Modeling Multithreaded Applications Using Petri Nets. *International Journal of Parallel Programming*, 30(5).

[Kavi et al., 2011] Kavi, K. M., Sheldon, F. T., and Reed, S. (2011). Specification and Analysis of Real-Time Systems Using CSP and Petri Nets. *International Journal of Software Engineering and Knowledge Engineering*, 6(2).

[Kehrer, 2019] Kehrer, P. (2019). Memory Unsafety in Apple's Operating Systems. https://langui.sh/2019/07/23/apple-memory-safety/. Accedido: 2023-02-24.

[Klabnik and Nichols, 2022] Klabnik, S. and Nichols, C. (2022). The Rust Programming Language. https://doc.rust-lang.org/book/. Accedido: 2023-02-20.

[Kungas, 2005] Kungas, P. (2005). Petri Net Reachability Checking Is Polynomial with Optimal Abstraction Hierarchies. *6th International Symposium, SARA 2005*, 6.

[Meyer, 2020] Meyer, T. (2020). A Petri-Net semantics for Rust. Master's thesis, Universität Rostock — Fakultät für Informatik und Elektrotechnik.

[Miller, 2019] Miller, M. (2019). Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ. Accedido: 2023-02-24.

[Moshtaghi, 2001] Moshtaghi, A. (2001). Modeling Multithreaded Applications Using Petri Nets. Master's thesis, The University of Alabama in Huntsville.

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4).

[Petri, 1962] Petri, C. A. (1962). Kommunikation mit Automaten. *Institut für Instrumentelle Mathematik*, 3.

[Reid, 2021] Reid, A. (2021). Automatic Rust verification tools (2021). https://alastairreid.github.io/automatic-rust-verification-tools-2021/. Accedido: 2023-02-20.

[Reid et al., 2020] Reid, A., Church, L., Flur, S., de Haas, S., Johnson, M., and Laurie, B. (2020). Towards making formal methods normal: meeting developers where they are. Accepted at HATRA 2020.

[Rust Project, 2023] Rust Project (2023). The rustc book. https://doc.rust-lang.org/rustc/. Accedido: 2023-02-20.

[Silva and dos Santos, 2004] Silva, J. R. and dos Santos, E. A. (2004). Applying Petri Nets to Requirements Validation. *IFAC Information Control Problems in Manufacturing*, 37(4).

[Simone, 2022] Simone, S. D. (2022). Linux 6.1 Officially Adds Support for Rust in the Kernel. https://www.infoq.com/news/2022/12/linux-6-1-rust/. Accedido: 2023-02-24.

[Singhal, 1989] Singhal, M. (1989). Deadlock detection in distributed systems. *Computer*, 22(11).

[Stack Overflow, 2022] Stack Overflow (2022). 2022 Developer Survey. https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages. Accedido: 2023-02-22.

[Stepanov, 2020] Stepanov, E. (2020). Detecting Memory Corruption Bugs With HWASan. https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html. Accedido: 2023-02-24.

[Stoep and Hines, 2021] Stoep, J. V. and Hines, S. (2021). Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Accedido: 2023-02-22.

[Stoep and Zhang, 2020] Stoep, J. V. and Zhang, C. (2020). Queue the Hardening Enhancements. https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html. Accedido: 2023-02-24.

[Szekeres et al., 2013] Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy*.

[The Chromium Projects, 2015] The Chromium Projects (2015). Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Accedido: 2023-02-24.

[Toman et al., 2015] Toman, J., Pernsteiner, S., and Torlak, E. (2015). CRUST: A Bounded Verifier for Rust. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[van der Aalst, 1994] van der Aalst, W. (1994). Putting high-level Petri nets to work in industry. *Computers in Industry*, 25.

[van Steen and Tanenbaum, 2017] van Steen, M. and Tanenbaum, A. S. (2017). *Distributed Systems*. Pearson Education, 3rd edition.