Automatización del flujo de diseño de un nanochip elaborado con librerías de diseño de TSMC de tecnología de 65 nanómetros

Juan Pablo Mora Argueta

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Automatización del flujo de diseño de un nanochip elaborado con librerías de diseño de TSMC de tecnología de 65 nanómetros

Trabajo de graduación presentado por Juan Pablo Mora Argueta para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2025

Vo.Bo.:

(f) ________________________________

MSc. Jonathan de los Santos

Tribunal Examinador:

(f) ________________________________

MSc. Jonathan de los Santos

(f) ________________________________

MSc. Carlos Esquit

(f) ________________________________

Ing. Juan Ricardo Girón Rubio

Fecha de aprobación: Guatemala, 24 de junio de 2025.

Prefacio

Mi nombre es Juan Pablo Mora, próximo a ser ingeniero mecatrónico de la Universidad del Valle de Guatemala, dado que será aprobado este mismo documento. Mi carrera universitaria data de diez años atrás, enero 2015. Primero que nada, debo agradecerle a todas las personas que por tanto tiempo me motivaron a seguir en el camino sin importar lo largo que fuese, entre estas personas, mi mamá, Alma Argueta, mi bisabuelo José León Pensamiento, mis amigos, Jorge y Sandy Martínez, mi hermano, Juan José Mora, el Vicerrector Fernando Paiz Mendoza y Carlos Esquit y finalmente a mí mismo, por tomar la decisión correcta constantemente a pesar de las incertidumbres y a veces incluso de los incentivos.

Para el lector, ofrezco una pequeña introducción a mi persona, lo que espero también sirva como introducción a este documento y a su motivación por mi parte. Para empezar, quisiera resaltar ser uno de los pocos, de no ser el único mecatrónico destinado al proyecto, comparado con el resto de excelentes ingenieros quienes son electrónicos. Hace un año aproximadamente, tuve la oportunidad de iniciar un proyecto de graduación. Buscando consejo fui a la oficina del MSc. Carlos Esquit, quien gustosamente me presentó los proyectos que la Universidad del Valle tenía para la carrera de mecatrónica, todas excelentes opciones y llenas de retos, desde investigación de materiales hasta desarrollos enfocados en inteligencia artificial. Esperando a ver qué clases y qué proyectos podría tomar el siguiente año, comenzamos a platicar, lo que llevó a conversar sobre un artículo de interés mutuo. En enero del 2022, Sam Zeloof, quien por segunda vez había logrado crear sus propios circuitos integrados utilizando litografía desde el garaje de su casa, demostraba su éxito en su *blog* personal, hablamos de esto un buen rato y fue cuando Carlos decidió contarme sobre el proyecto de la universidad y mi introducción a «El Gran Jaguar».

Parte de las razones porque mi cierre de pénsum fuese tan prolongado, es porque mi curiosidad se enfocó en las computadoras desde una temprana edad, por lo que para el 2016, después de haberme reducido la carga de estudios, comencé a trabajar como desarrollador de *software*. Durante todo este tiempo y desde mucho antes, probablemente desde los trece años, estuve expuesto al uso de Linux como sistema operativo, al romper y descubrir cosas a través de prueba y error. Durante el tiempo en la universidad, Carlos Esquit estuvo informado sobre mis quehaceres y mi apego al *software*, incluso al punto de sugerirme un cambio de carrera. Fue durante aquella conversación que, mi *expertise* de muchos años trabajando con Linux demostraba ser de gran ventaja para contribuir en un proyecto donde la mayor dificultad no

era exactamente el chip a crear, sino la dependencia de instalarse en Linux y aprender todo un nuevo sistema operativo antes de comenzar a trabajar.

Mi compromiso personal y ante la universidad es entregar en gran detalle todos los recursos posibles para facilitar todo detalle alrededor de las herramientas que se utilizan, para que los estudiantes que van después de mí amplifiquen sus fuerzas con la información y transferencia de recursos que este documento y cualquier trabajo posterior relacionado pueda aportar. Espero también que los documentos que se escriban citando este, compartan esta misma visión. Finalmente invito al lector a la actividad, a la acción, Guatemala tiene suficientes soñadores, es momento que tengamos ejecutores.

Gracias.

Índice

Prefacio iv

Lista de figuras viii

Lista de cuadros x

Resumen xi

Abstract xii

1. Introducción 1

2. Antecedentes 2

3. Justificación 6

4. Objetivos 7
- 4.1. Objetivo general . . . 7
- 4.2. Objetivos específicos . . . 7

5. Alcance 8

6. Marco teórico 9
- 6.1. Virtualización e *hipervisor* . . . 9
  - 6.1.1. Aislamiento de memoria . . . 9
  - 6.1.2. Aislamiento de *hardware* e instrucciones . . . 10
- 6.2. Máquina virtual . . . 10
- 6.3. Contenedores . . . 10
- 6.4. *Pipelines* y CI/CD en el desarrollo de *software* . . . 11
  - 6.4.1. Definición y propósito de CI/CD . . . 11
  - 6.4.2. Impacto de los contenedores en CI/CD . . . 12
  - 6.4.3. Automatización con Jenkins . . . 12
  - 6.4.4. Aplicación práctica de CI/CD y Jenkins en el proyecto . . . 13
  - 6.4.5. *Pipeline* para síntesis lógica . . . 13

6.4.6. *Pipeline* para síntesis física . . . 13
6.4.7. Comparación con métodos tradicionales . . . 13
6.5. Uso de imágenes y volúmenes . . . 14
6.6. Entrega e integración continua . . . 14
6.7. Cadena de montaje (o *pipeline*) . . . 15
6.8. Git . . . 15
6.8.1. Conceptos básicos de Git . . . 16
6.8.2. Flujo de trabajo con Git . . . 16
6.9. Docker . . . 16
6.9.1. Creación de una imagen de Docker . . . 16
6.9.2. Inicializando una imagen de Docker para hacer un contenedor . . . 17
6.10. Creación de una cadena de montaje con Jenkins . . . 18
6.10.1. Creación de las instrucciones una cadena de montaje con Jenkins . . . 19
6.11. Distro . . . 20
6.12. SSH . . . 20
6.12.1. Configuración de SSH . . . 20
6.12.2. Configuración global de SSH . . . 21

7. Análisis y mejora del *script* de instalación para entornos Linux: evaluación de herramientas y mejores prácticas 22
7.1. Cambios en los estándares de seguridad y actualización del proceso de instalación . . . 23
7.2. Configuración de seguridad de SSH para la conexión con SFTP de Synopsys . 23
7.2.1. Configuración global . . . 23
7.2.2. Configuración individual . . . 23
7.3. Descarga FTP e instalación manual con interfaz gráfica . . . 24

8. Evaluación de Apptainer como alternativa a Docker: instalación, pruebas y comparación de características 27
8.1. Creación de contenedores en Apptainer . . . 28
8.1.1. Introducción a Apptainer . . . 28
8.1.2. Definiendo un contenedor personalizado . . . 30
8.1.3. Bootstrap y from . . . 31
8.1.4. Sección %setup . . . 31
8.1.5. Sección %environment . . . 31
8.1.6. Sección %post . . . 31
8.1.7. Sección %startscript . . . 31

9. Uso de programas de Synopsys dentro de Apptainer 32
9.1. Instalación de Synopsys Container . . . 32
9.2. creación de una imagen apta para ejecutar los programas de Synopsys . . . 36

10. Depuración de imágenes y contenedores de Apptainer 38
10.1. Creación de una imagen de Apptainer con permisos de *root* . . . 38

11. Depuración de *scripts* de Synopsys 40
11.1. Variables de entorno para depuración . . . 40

12. Instalación de servidor de Jenkins **42**
12.1. Instalación de Jenkins 42
12.2. Configuración inicial del servidor de Jenkins 43
12.2.1. Personalización de instalación y creación de nodos adicionales 46
12.3. Creación de un *pipeline* desde la interfaz de Jenkins 46
12.4. Explicación del *script* de *pipeline* 50
12.5. *Pipeline* de síntesis física 52
12.5.1. Agente con espacio de trabajo personalizado 54
12.5.2. Variables de entorno adicionales 54
12.5.3. Ejecución de síntesis física 55
12.5.4. Manejo del archivo GDS 55
12.5.5. Compresión y almacenamiento de resultados 55
12.5.6. Rama `sintesis-fisica` 55

13. Diseño de una estructura de archivos estandarizada para *pipelines* **56**
13.1. Estructura de librerías para diseño de electrónica automatizado (EDA) 56
13.2. Organización de *scripts* para su uso en *pipelines* de Jenkins 57

14. Conclusiones **58**
14.1. Principales logros 58
14.2. Limitaciones 59
14.3. Reflexión final 59

15. Recomendaciones **60**
15.1. Instalación de *software* de Synopsys y uso del *script* de descargas 60
15.2. Manejo de *artifacts* y resultados de procesos en Jenkins 60

16. Bibliografía **62**

17. Glosario **65**

Lista de figuras

1. Máquina virtual comparada con contenedores [16] 11
2. Anatomía de un contenedor que utiliza volúmenes [17] 14

3. Explorador de archivos por defecto de GNOME 25
4. Menú de *Other Locations* seleccionado. 25
5. Servidor EFT de Synopsys requiriendo contraseña. 26
6. Descarga iniciada con indicador visual directamente en el explorador de archivos 26

7. Explorador de archivos mostrando la carpeta de Container V2 33
8. Instalador de Synopsys mostrando la carpeta seleccionada. 33
9. Instalador de Synopsys mostrando la carpeta base de instalación. 34
10. Instalador de Synopsys mostrando la carpeta base de instalación. 34
11. Script de configuración de Synopsys Container. 35
12. Configuración generada por el *script* de Synopsys Container. 36
13. *Output* del proceso de instalación de la configuración creada por Synopsys Container. 36

14. *Output* del servicio de Jenkins mostrando los pasos de inicialización. 43
15. Interfaz *web* de Jenkins pidiendo la contraseña de configuración. 44
16. Interfaz *web* de Jenkins instalando los plugins recomendados. 44
17. Interfaz de Jenkins creando un usuario administrador nuevo 45
18. Interfaz de Jenkins mostrando la página inicial 45
19. Interfaz de Jenkins mostrando la página de configuraciones 46
20. Interfaz de Jenkins creando un usuario administrador nuevo 52

Lista de cuadros

Cuadro 6.1: Ejemplo de *Dockerfile* básico para Node.js . . . 17
Cuadro 6.2: Construcción de imagen a partir de Dockerfile . . . 17
Cuadro 6.3: Lanzamiento de contenedor a partir de una imagen . . . 18
Cuadro 6.4: Instalación de Java en Rocky . . . 18
Cuadro 6.5: Instalación de Jenkins en Rocky . . . 19
Cuadro 7.1: Ejemplo de Jenkinsfile básico . . . 19

Cuadro 7.1: Parámetros de configuración SSH globales . . . 23
Cuadro 7.2: Parámetros de configuración SSH locales. . . . 24
Cuadro 7.3: Demostración de configuracion de SSH aplicada funcionando correctamente. . . . 24

Cuadro 8.1: Creación de una imagen básica de Apptainer basado en Docker Hub. . . 29
Cuadro 8.2: Demostración de configuración por defecto de un contenedor en Apptainer. . . . 29
Cuadro 8.3: Definición demostrativa de una imagen de Apptainer. . . . 30

Cuadro 9.1: Uso de *script* para generar configuración de Synopsys Container. . . . 34
Cuadro 9.2: Uso de *script* para generar configuración de Synopsys Container con datos reales . . . 35
Cuadro 9.3: Uso de *script* para instalar configuración de Synopsys Container. . . . 35
Cuadro 9.4: Actualización de archivo de definición para instalar nuevas librerias o dependencias. . . . 37
Cuadro 9.5: Definición de Rocky 8.9 utilizada para generar la imagen compatible con Synopsys Container. . . . 37

Cuadro 10.1: Creación de imagen de Apptainer basado en su archivo de definición. 39

Cuadro 12.1: Actualizar los paquetes del sistema e instalar Java. . . . 42
Cuadro 12.2: Agregar el repositorio de Jenkins. . . . 42
Cuadro 12.3: Instalar Jenkins y registrar el servicio. . . . 43
Cuadro 12.4: Inicializar Jenkins. . . . 43
Cuadro 12.5: Definición del *Pipeline Script* utilizado para hacer síntesis lógica. . . . 50

Cuadro 12.6: Definición del *Pipeline Script* utilizado para hacer síntesis física y comprimir los archivos. 54

Cuadro 13.1: Organización de directorios sugerida para librerías. 57

Resumen

En el presente trabajo se abordó la mejora y automatización de procesos relacionados con la instalación y ejecución de herramientas de diseño electrónico y *software* de automatización, utilizando entornos Linux y contenedores. El objetivo fue optimizar la implementación de *pipelines* de integración continua y la configuración de entornos seguros para el uso de herramientas avanzadas de Synopsys y tecnologías relacionadas, como Docker y Apptainer.

Inicialmente, se evaluaron y actualizaron los estándares de seguridad en *scripts* de instalación, incluyendo configuraciones globales e individuales para SSH en conexiones SFTP, así como la descarga e instalación manual mediante interfaces gráficas. Posteriormente, se exploró el uso de Apptainer como alternativa a Docker, destacando la creación y personalización de contenedores capaces de ejecutar herramientas de Synopsys, incluyendo la depuración de imágenes con permisos de *root* y el ajuste de variables de entorno.

En la siguiente etapa, se configuró un servidor Jenkins, abordando tanto su instalación como la personalización inicial y la creación de *pipelines* desde su interfaz. Se diseñó una estructura de archivos estandarizada para la automatización de procesos, enfocándose en la organización eficiente de librerías y *scripts* para *pipelines* de diseño electrónico automatizado (EDA).

Los resultados incluyeron la optimización del flujo de instalación y ejecución de programas de Synopsys dentro de Apptainer, una configuración más segura de los entornos de trabajo y una metodología estructurada para el diseño y uso de *pipelines*. Como propuestas futuras, se planteó una mayor automatización de procesos, incluyendo la generación dinámica de *scripts* y una mejora en la interoperabilidad entre herramientas. Esto permitirá reducir la intervención manual y aumentar la eficiencia en proyectos de diseño automatizado.

Abstract

This work addressed the improvement and automation of processes related to the installation and execution of electronic design tools and automation *software*, using Linux environments and containers. The objective was to optimize the implementation of continuous integration *pipelines* and the configuration of secure environments for the use of advanced Synopsys tools and related technologies such as Docker and Apptainer.

Initially, security standards in installation *scripts* were evaluated and updated, including global and individual configurations for SSH in SFTP connections, as well as the manual download and installation through graphical interfaces. Subsequently, the use of Apptainer as an alternative to Docker was explored, highlighting the creation and customization of containers capable of running Synopsys tools, including the debugging of images with *root* permissions and the adjustment of environment variables.

In the next stage, a Jenkins server was configured, addressing both its installation and initial customization, as well as the creation of *pipelines* from its interface. A standardized file structure was designed for process automation, focusing on the efficient organization of libraries and *scripts* for automated electronic design (EDA) *pipelines*.

The results included the optimization of the installation and execution flow of Synopsys programs within Apptainer, a more secure configuration of working environments, and a structured methodology for designing and using *pipelines*. Future proposals include further automation of processes, such as the dynamic generation of *scripts* and improvements in tool interoperability. This approach will reduce manual intervention and increase efficiency in automated design projects.

CAPÍTULO 1

Introducción

La automatización de procesos en diseño electrónico requiere herramientas eficientes y bien configuradas, especialmente en entornos de desarrollo complejos y colaborativos como los de la Universidad del Valle de Guatemala. Este proyecto se enfoca en mejorar el uso de herramientas de Synopsys mediante la integración de un *pipeline* automatizado en Jenkins, con contenedores y entornos de virtualización, optimizando el flujo de trabajo en el diseño y simulación de circuitos.

El documento aborda cada aspecto esencial del proyecto, comenzando con el análisis y mejora del *script* de instalación en sistemas Linux, donde se evalúan herramientas y mejores prácticas para configurar el entorno de desarrollo de forma eficiente. A continuación, en el Capítulo 8, titulado Evaluación de Apptainer como alternativa a Docker, se explora la instalación y características de Apptainer, comparándola con Docker para determinar su viabilidad en entornos académicos y de alta computación.

En el Capítulo 9, Uso de programas de Synopsys dentro de Apptainer, se detalla cómo ejecutar herramientas de Synopsys en contenedores Apptainer, optimizando la ejecución y evitando conflictos entre versiones de dependencias. Luego, el Capítulo 10, Depuración de imágenes y contenedores de Apptainer, describe técnicas y herramientas para identificar y resolver problemas en las imágenes y contenedores utilizados, asegurando un entorno estable y funcional.

El Capítulo 11, Depuración de *scripts* de Synopsys, se enfoca en la resolución de errores y optimización de *scripts* para mejorar la funcionalidad y el rendimiento de las simulaciones y síntesis. Finalmente, el Capítulo 12, Instalación de servidor de Jenkins, ofrece una guía detallada para configurar Jenkins en un entorno Linux, permitiendo la integración y entrega continua del flujo de trabajo automatizado.

Este documento busca proporcionar una guía completa y replicable para la automatización de herramientas de Synopsys en un entorno de contenedorización y CI/CD. La implementación de estos procesos mejora la eficiencia del flujo de trabajo y facilita su uso en proyectos futuros en sistemas Linux.

CAPÍTULO 2

Antecedentes

En la era de la miniaturización y la creciente demanda de dispositivos electrónicos más pequeños y eficientes, el desarrollo de nanochips ha tomado un papel crucial. En este contexto, la Universidad del Valle de Guatemala desea destacar en el campo de la nanoelectrónica, siendo pionera en la creación del primer nanochip en Centroamérica.

El diseño y fabricación de nanochips es un proceso extenso y complejo que requiere el uso y *expertise* en *software* especializado como el de Synopsys. Además, este conocimiento exige una base sólida en el uso del sistema operativo Linux, específicamente en una versión no muy conocida que busca continuar con el mantenimiento del proyecto CentOS. Todos estos desafíos se suman, limitando significativamente la cantidad de personas que podrían fabricar un chip con estas herramientas, calidad y escala. Estos conocimientos son fundamentales y preliminares a cualquier otro necesario en el diseño eficiente de un nanochip.

Durante los trabajos y detalles compartidos a continuación se refuerza la idea de que hacer un nanochip es un proceso extremadamente difícil.

La universidad ha obtenido acceso a licencias de distintos programas de la empresa Synopsys. Sin embargo, la instalación, actualización y simulación de estos programas representan un desafío. El primer trabajo dedicado a la creación y diseño de un nanochip fue propuesto por Jonathan de los Santos [1] hace diez años. En 2014, Jonathan propuso la creación de un simple sumador/restador con tecnología de fabricación de 28 nm utilizando *software* de Synopsys. Este primer trabajo sirve como fundamento para los procesos de instalación de los programas de Synopsys que se utilizan en la actualidad. Este trabajo entrega extensos avances y documentación sobre cómo instalar y utilizar las herramientas de *software* de Synopsys, pero también resalta la increíble complejidad de ambos procesos. Las principales dificultades que se resolvieron con el trabajo de Jonathan fueron:

- Instalar y configurar inicialmente el sistema operativo CentOS.
- Descargar las aplicaciones desde un servidor FTP con SSL a un servidor CentOS.

- Investigar y documentar el proceso de instalación de cada aplicación de Synopsys.
- Lograr el funcionamiento correcto de un servidor de licencias para que todos los programas funcionen.
- Proveer a la facultad con guías técnicas altamente detalladas para la instalación y uso de los programas de Synopsys.
- Finalmente, se logró por primera vez hacer uso de estos componentes de *software* para el diseño básico de un sumador de 32 bits.

Posteriormente, Rubio entregó el trabajo de diseñar un flujo de fabricación de un chip que sería capaz de enviar caracteres ASCII, diseñado como una máquina de estados finitos utilizando las herramientas con *software* VLSI de Synopsys [2], y Luis Nágera se especializó en la implementación de circuitos a nivel de *netlist* como primer paso en el proceso de fabricación [3]. El trabajo de Luis resalta nuevamente lo extenso que es el proceso, al dedicarse únicamente a implementar la generación de un circuito a partir de *netlists*.

A partir de esta fecha, los siguientes años lograron avanzar mucho más rápido en el proyecto gracias a varias contribuciones continuas. El 2020 fue un año especialmente bueno para el proyecto, ya que se logró determinar la lista de tareas necesaria para pasar de un circuito simulado a uno más cercano a fabricarse. Parte de lo que se logró definir ese año fue lo siguiente:

- Definición del flujo en la herramienta VCS para la simulación de HDLs en la fabricación de un chip con tecnología nanométrica CMOS, por Jefferson Ruano [4].
- Etapa de verificación física de diseño en silicio vs. esquemático (LVS) en el flujo de diseño para un chip a nanoescala, por Juan Ricardo Girón Rubio [5].
- Corrección de anillo de entradas/salidas y pruebas de antenna y ERC para la definición del flujo de diseño del primer chip con tecnología nanométrica desarrollado en Guatemala, hecho por Marvin Geovanni Flores Espino [6].

Es solo hasta este último trabajo, realizado por Marvin Geovanni, que podemos ver un diagrama de flujo claro y conciso que describe dónde y qué hacen cada una de las distintas partes pertenecientes a este sistema tan complejo.

Inicialmente se trabajó instalando todos los programas en CentOS por su compatibilidad con Red Hat, pero recientemente se ha dado a conocer que este sistema operativo será descontinuado a partir del 30 de junio de 2024. Afortunadamente, han habido avances en la migración hacia Rocky Linux, un *fork* de CentOS.

En el año 2019, la empresa TSMC (Taiwan Semiconductor Manufacturing Company) [7] le presentó a la Universidad del Valle de Guatemala la oportunidad de fabricar un chip a partir de un diseño propio de la universidad. En este año, el trabajo de Steven Rubio propuso el flujo de diseño (*design flow*) para la fabricación de un chip que sería capaz de enviar caracteres ASCII, diseñado como una máquina de estados finitos utilizando las herramientas de *software* de Synopsys.

Durante el año 2022, Paola Mendizábal trabajó en el proceso de instalación de las aplicaciones de Synopsys en contenedores utilizando Singularity [8], ahora conocido como Apptainer, logrando importantes avances. Este enfoque ha permitido encapsular el *software* y sus dependencias en una unidad autónoma, facilitando su instalación y actualización. Sin embargo, los pasos a seguir para este proceso aún son varios y requieren de conocimientos técnicos avanzados en Linux. A partir de este año, casi todos los trabajos, como el de Carlos Letona [9], Diego Equité [10], Luis Gómez [11], [12], entre otros, todos utilizan *scripts* los cuales apoyan, pero respaldan más aún la necesidad de esta automatización.

Finalmente a principios de junio de 2024, se recibió información de el ingeniero Carlos Esquit detallando que estarían disponibles librerías de TSMC con tecnología de 65 nm.

Hasta el momento, los pasos definidos son los siguientes:

- Descargar el *software* de Synopsys utilizando el protocolo SFTP.
- Instalar Singularity/Apptainer.
- Crear un contenedor Singularity/Apptainer.
- Configurar el contenedor con el archivo de configuración.
- Descargar e instalar las aplicaciones de Synopsys en el contenedor.
- Verificar que los directorios *home* han sido montados correctamente.
- Verificar que todos los puntos de montaje están disponibles.
- Desplegar la configuración del contenedor en las aplicaciones de Synopsys instaladas.
- Cambiar la variable de entorno SNPS_CONTAINER para acceder automáticamente al contenedor.
- Iniciar las aplicaciones accediendo a los contenedores.

Actualmente se tiene un *script* que descarga gran parte de los programas requeridos, siguiendo parcialmente el proceso de instalación, omitiendo y fallando en ciertos aspectos según las configuraciones que difieran de la computadora tanto como la autenticación y descarga de Synopsys. Un total de 24 aplicaciones de Synopsys se descargaron e instalaron de manera mayormente exitosa utilizando estos *scripts*, lo que demuestra su eficacia pero también el trabajo pendiente por implementar en el *script* para mejorarlo y documentarlo.

Pasos de creación de un chip desde lógica hasta fabricación

El proceso que descubierto a través de los esfuerzos y trabajos previos se ha definido como lo siguiente:

Generación de archivo Verilog: este archivo se genera a partir de cualquier herramienta Verilog que genere una descripción de *hardware* (HDR por sus siglas en inglés) que contenga todas las características lógicas del chip o circuito a fabricar.

Síntesis lógica: utilizando Synopsys Design Compiler convertimos el HDL a su equivalente compuesto por compuertas lógicas. En un segundo paso se conecta la primer fase al *footprint* del chip con entradas y salidas.

Síntesis física: esta etapa de diseño nos genera un resultado o esquemático físico que equivale a la síntesis lógica generada anteriormente, compuesto por las librerías, compuertas y componentes deseados.

Comprobaciones físicas: esta etapa se compone de muchas otras, las cuales son las más complejas y donde actualmente se tienen problemas pendientes de resolver. El componente físico debe pasar distintas simulaciones listadas a continuación.

- *Design Rule Check*
- *Layout vs Schematic*
- *Antenna Rule Check*
- Extracción de parásitos.

Cabe mencionar los *scripts* de automatización detallados en los trabajos del 2022 creados por Alison Aguilar quien logró automatizar parcialmente, por medio de *scripts* de Python y Bash la etapa de síntesis lógica y creación de archivos Verilog [13], junto con la automatización del proceso de extracción de parásitos. Stefan Schwendener, específicamente hizo un *script* en Bash que simplificará el proceso de creación de contenedores automatizados.

CAPÍTULO 3

Justificación

Como se menciona consistentemente a través de todos los trabajos relacionados con el uso e instalación de los programas de Synopsys, el proceso es tedioso, complicado, lento y requiere de un amplio conocimiento de Linux en una distribución no muy conocida, como es Rocky Linux. Además, se debe lidiar con las constantes actualizaciones por parte de Synopsys, que incluyen cambios, nueva documentación y posibles modificaciones en la instalación de los programas.

El objetivo prioritario es delegar el conocimiento de las plataformas fundamentales para ejecutar los programas exclusivamente a sistemas automatizados y a las personas que deseen entender más a profundidad el proceso de creación de un nanochip, con el fin de actualizarlos o mejorarlos. Claramente, esto justifica la creación de un sistema automatizado de fabricación de imágenes de Apptainer (previamente llamado Singularity), con su respectiva documentación de la instalación y uso de dichas imágenes para proveer a los usuarios un ambiente estandarizado y fácil de obtener, con todas las aplicaciones de Synopsys de una forma mucho más accesible y abstracta, permitiéndoles dedicarse únicamente al trabajo de ingeniería electrónica y no al trabajo de administración de sistemas.

Asimismo, una vez creadas estas imágenes, se podrán utilizar para crear un repositorio privado de Docker o Apptainer que permita desarrollar un *pipeline* completamente basado en contenedores autoescalables. Esta forma de trabajo es altamente efectiva, tanto que se ha propagado entre diversas aristas del trabajo moderno; incluso, existe soporte parcial nativo de Synopsys para implementar esta metodología [14].

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Simplificar y facilitar el proceso de diseño de un nanochip con herramientas de Synopsys utilizando herramientas de automatización de procesos tales como la creación de contenedores, aplicaciones de integración continua y trabajos de compilado automático.

4.2. Objetivos específicos

- Instalar las herramientas de Synopsys en contenedores.
- Ejecutar las aplicaciones desde la línea de comando con los parámetros necesarios para que lean y escriban sus resultados de forma predecible, invocando las instalaciones dentro de los contenedores.
- Documentar exhaustivamente la instalación de la plataforma de código abierto que se elija para el funcionamiento de la cadena de montaje a construir.
- Crear una instalación demostrativa de integración continua reproducible para la automatización del proceso de compilado de archivos Verilog.
- Ejemplificar una alerta descriptiva que comparta los errores al momento que falle cualquier paso de la cadena de montaje.
- Crear configuraciones de procesos de integración continua que puedan servir de base para futuros proyectos de creación de chips utilizando las herramientas de Synopsys.

CAPÍTULO 5

Alcance

El alcance de este proyecto es optimizar el uso de las herramientas de Synopsys mediante la creación de un proceso estandarizado de automatización en forma de un *pipeline* en Jenkins. Este *pipeline* permitirá la transición de un flujo de trabajo manual, basado en la interfaz gráfica, a una ejecución automatizada utilizando la versión CLI (interfaz de línea de comandos) de las herramientas de Synopsys. Esta transición busca mejorar la eficiencia en procesos como la síntesis lógica y física, sentando una base sólida para la futura escalabilidad y automatización completa.

Además, se generará la documentación adecuada para el lanzamiento de esta plataforma, incluyendo los pasos para crear procesos y tareas de automatización en la plataforma de *continuous integration* seleccionada. La documentación abarcará métodos de actualización, depuración y limitaciones del sistema. Para ello, se emplearán herramientas de *software* como Jenkins para la automatización, Docker y Apptainer para la contenedorización, y herramientas específicas de Synopsys, como *Synopsys Container*. Como prueba de concepto y paso fundamental hacia la automatización completa del diseño de chips, se integrará el proceso de síntesis lógica utilizando la herramienta VCS y síntesis física utilizando *icc2_shell* junto con ICV.

Finalmente, se proporcionará una documentación ampliada, basada en los avances del proyecto previo, «El Gran Jaguar» con las modernizaciones y recomendaciones necesarias para optimizar el proceso detallado previo a este documento.

Debido a que se busca entregar procesos automatizados, funcionales y listos para usarse, el alcance estará limitado por el conocimiento adquirido sobre el uso de las herramientas CLI, ya que se debe saber cómo usar la herramienta antes de poder automatizarla. En todo caso, este documento presenta como objetivo la documentación de cualquier automatización generada, por lo que deberá servir para la creación de nuevas y futuras automatizaciones.

CAPÍTULO 6

Marco teórico

Los contenedores son una rama de la virtualización de sistemas operativos, que han ganado fama y participación en el mercado en años recientes por su gran capacidad de adaptabilidad, portabilidad y reproducción de sistemas o ambientes de desarrollo consistentes. Es importante distinguir entre un contenedor y una máquina virtual, ya que esta distinción conlleva ventajas y desventajas que se deben tomar en cuenta al momento de decidir cuál utilizar. Por esta razón se explicarán a continuación, comenzando por entender qué es la virtualización, un *hypervisor* y una máquina virtual.

6.1. Virtualización e *hipervisor*

La virtualización se refiere a la capacidad de ejecutar varios sistemas operativos simultáneamente, a través de un proceso o aplicación ejecutada desde el sistema operativo principal. La teoría de la virtualización ha existido por muchos años, pero es hasta tiempos recientes, gracias a la disponibilidad de CPUs potentes con múltiples núcleos, que la virtualización ha dejado de ser exclusiva para el ámbito *enterprise* [15]. El proceso de virtualizar un sistema operativo se reduce a dos aspectos principales:

6.1.1. Aislamiento de memoria

El CPU, o sus instrucciones de microcódigo, deben manejar el espacio de memoria del sistema en modo virtualizado, traduciendo eficientemente las direcciones de memoria y evitando que el sistema huésped acceda a la memoria del anfitrión. Explicando en un nivel técnico, en Linux, cada proceso en el sistema operativo tiene un número de identificador o *process id* o PID, el cual va desde el uno ascendentemente hasta el infinito teórico. De esta misma forma el sistema asigna la memoria RAM, identificando cada celda en una «dirección» que, nuevamente va desde cero hasta 2^{32} o hasta 2^{64} según el número de bits que tenga el sistema operativo y la capacidad de memoria física.

Al virtualizar un sistema, en términos simples, se utiliza un desface artificial en la numeración de procesos y en la dirección de la RAM. Esto significa que el sistema virtualizado obtiene un nuevo proceso *root* el cual se convierte en un cero virtual, en caso el PID fuese 14231, este ahora se indica virtualmente como 14231+0, el sistema operativo anfitrión debe traducir la memoria y los PIDs de tal forma que el PID 28 del sistema virtualizado se redirecciona al número 14231+28.

6.1.2. Aislamiento de *hardware* e instrucciones

Para el sistema huésped, las instrucciones del CPU deben ejecutarse exactamente como lo harían en un sistema físico.

Actualmente, los CPUs modernos cuentan con extensiones o instrucciones específicas de virtualización, como Intel VT-x y AMD-V, diseñadas para facilitar el proceso de virtualización haciendo que el proceso de traducción entre instrucciones virtuales y del sistema anfitrión sean casi instantáneas, para el usuario final esto significa que el sistema virtualizado funciona, prácticamente a la misma velocidad y responsividad que tendría al instalarlo como sistema único. El programa que se encarga de manejar todos estos aspectos del sistema se llama *hipervisor*. Es importante aclarar que un sistema de virtualización no es lo mismo que un sistema de emulación, por lo que el tipo de instrucciones en el CPU virtual debe coincidir con el tipo de instrucciones del CPU real.

6.2. Máquina virtual

Una máquina virtual es la forma más completa de virtualizar un sistema operativo. Esto se logra haciendo que el hipervisor exponga una serie completa de dispositivos virtuales al sistema huésped. El sistema huésped, por su parte, está instalado completamente, es decir, todos los componentes necesarios para una instalación física son requeridos para que la instalación sea funcional, dado que se instalarán los programas y *drivers* necesarios, un sistema ya instalado físicamente, se puede virtualizar y vice versa. Esto tiene la ventaja de funcionar como un sistema completamente independiente que se comporta prácticamente igual al equivalente con *hardware* físico, a costa de ocupar más espacio en disco y ser «pesado» para el sistema anfitrión. Como se menciona anteriormente, hoy en día se tiene la grán ventaja de tener sistemas con múltiples núcleos por lo que pueden destinarse recursos y límites específicos a una máquina virtual.

6.3. Contenedores

En este contexto, los contenedores presentan una alternativa que resuelve directamente las desventajas de una máquina virtual al no tener que instalarse un sistema operativo entero encima de otro. La premisa tiene lógica, si mi sistema ya tiene el Linux *kernel* versión cinco, podría simplemente «exponerlo» de alguna forma al sistema operativo virtualizado para evitar tener dos copias exactas del mismo. Un contenedor comparte muchas de las ventajas

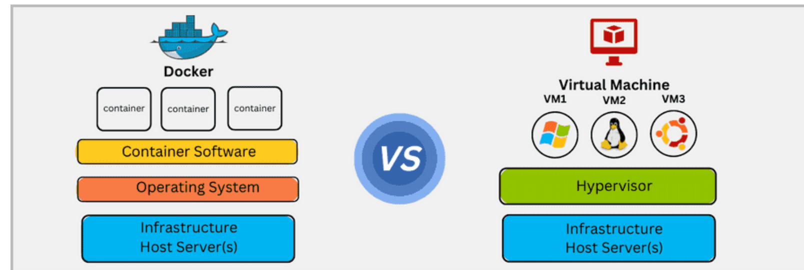

Figura 1: Máquina virtual comparada con contenedores [16]

de la virtualización y de las máquinas virtuales, pero con una diferencia principal mencionada antes: generalmente un contenedor es mucho más ligero (*thin* en inglés). A diferencia de una máquina virtual, un contenedor comparte la mayoría de los componentes básicos, como el *kernel* del anfitrión. Esto permite que el espacio requerido por un contenedor sea únicamente el necesario para las aplicaciones o el *software* especializado que incluya dicho contenedor.

Otra ventaja de los contenedores es la distinción entre *imágenes* y *volúmenes*. Un contenedor se crea a partir de múltiples capas, cada una de las cuales modifica algo en el sistema virtual, ya sea la instalación de algún componente o la creación o eliminación de archivos. Cada acción se convierte en una capa y el conjunto se conoce como una imagen. El equivalente de esto en una máquina virtual se conoce como un *snapshot*, con la diferencia de que los contenedores siempre se inician a partir de esta imagen, que siempre existe en modo de solo lectura. Cualquier cambio que ocurra en los archivos del contenedor durante su ejecución solo puede existir en un sistema de archivos temporal o en un *volumen*. En la Figura 1 se muestra de forma gráfica y resumida la diferencia entre contenedores y máquinas virtuales.

6.4. *Pipelines* y CI/CD en el desarrollo de *software*

Los *pipelines* y la integración continua/despliegue continuo (CI/CD) representan una transformación moderna en el desarrollo de *software*, automatizando tareas repetitivas y mejorando la calidad y consistencia de las entregas. En este contexto, los contenedores han jugado un papel crucial, ya que encapsulan el ambiente necesario para ejecutar procesos automatizados en entornos consistentes y portables.

6.4.1. Definición y propósito de CI/CD

CI/CD engloba un conjunto de prácticas para entregar *software* rápidamente y de manera fiable. Estas prácticas combinan tres elementos clave:

- **Integración continua (CI):** combina automáticamente los cambios realizados por varios desarrolladores en un solo repositorio, ejecutando pruebas para garantizar que los cambios sean compatibles con el código existente.
- **Entrega continua (CD):** automatiza el despliegue del *software* en entornos predefinidos (por ejemplo, pruebas o *staging*).

- **Despliegue continuo:** una extensión de la entrega continua que incluye la implementación automatizada en producción después de pasar todas las pruebas.

Los contenedores contribuyen significativamente a este flujo al garantizar que los entornos de desarrollo, pruebas y producción sean idénticos, minimizando errores relacionados con diferencias de configuración.

6.4.2. Impacto de los contenedores en CI/CD

Los contenedores, como Docker, resuelven problemas comunes en CI/CD mediante:

- **Consistencia:** eliminan la «discrepancia ambiental» al garantizar que las aplicaciones se ejecuten en entornos idénticos, desde el desarrollo hasta la producción.
- **Velocidad:** la portabilidad y ligereza de los contenedores permiten desplegar aplicaciones rápidamente en múltiples entornos.
- **Escalabilidad:** facilitan la elasticidad en la nube, permitiendo ejecutar múltiples instancias del mismo contenedor para satisfacer demandas crecientes.

6.4.3. Automatización con Jenkins

Jenkins es una herramienta central en CI/CD, diseñada para gestionar *pipelines* y automatizar tareas repetitivas. Con Jenkins, se pueden definir cadenas de montaje (*pipelines*) que integren pruebas, compilaciones y despliegues en una única secuencia automatizada. Esto no solo mejora la eficiencia, sino que también asegura que cada etapa sea auditable y repetible.

Los elementos clave de un *pipeline* en Jenkins incluyen:

- **Agentes:** los nodos donde se ejecutan las tareas. Estos pueden ser físicos o virtuales.
- **Etapas:** divisiones del *pipeline* que agrupan tareas relacionadas, como compilación, pruebas o despliegue.
- **Resultados:** archivos mejor conocidos como *artifacts* generados en una etapa que se reutilizan en otras, como binarios o reportes de prueba.

Un ejemplo típico de *pipeline* podría incluir:

- **Compilación:** construir el código fuente.
- **Pruebas:** ejecutar pruebas unitarias y de integración.
- **Empaquetado:** crear un contenedor con Docker.
- **Despliegue:** subir la imagen a un registro de contenedores y lanzar el contenedor en producción.

6.4.4. Aplicación práctica de CI/CD y Jenkins en el proyecto

El proyecto requiere una integración eficiente de CI/CD con Jenkins para gestionar las herramientas de Synopsys en un entorno automatizado. La implementación utiliza contenedores para encapsular herramientas como VCS e ICC2, permitiendo que los *pipelines* gestionen tareas de síntesis lógica y física.

6.4.5. *Pipeline* para síntesis lógica

Este *pipeline* define etapas específicas para:

1. Configurar el ambiente utilizando contenedores Synopsys.
2. Compilar y verificar diseños en VHDL o Verilog.
3. Archivar resultados como *artifacts* para su consulta futura.

6.4.6. *Pipeline* para síntesis física

Incluye tareas adicionales, como:

1. Ejecutar *scripts* TCL en ICC2 para generar diseños GDS.
2. Comprimir y almacenar resultados en un servidor centralizado.
3. Validar la calidad del diseño con herramientas como IC *Validator*.

Ambos *pipelines* están diseñados para maximizar la reutilización de imágenes Docker predefinidas, asegurando consistencia entre equipos y entornos.

6.4.7. Comparación con métodos tradicionales

El enfoque basado en CI/CD con Jenkins y contenedores ofrece varias ventajas frente a métodos tradicionales:

- **Eficiencia operativa:** las tareas automatizadas reducen la carga manual y los errores humanos.
- **Menor tiempo de entrega:** los procesos automatizados disminuyen el tiempo necesario para pasar de desarrollo a producción.
- **Calidad consistente:** la integración de pruebas automatizadas asegura que los problemas se detecten temprano.

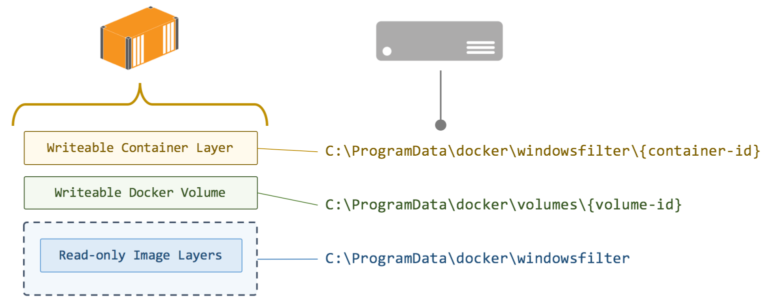

Figura 2: Anatomía de un contenedor que utiliza volúmenes [17]

6.5. Uso de imágenes y volúmenes

La distinción entre imagen y volumen en un contenedor puede explicarse como la diferencia entre un taller y el área de trabajo. La imagen mantiene un estado constante y predecible, mientras que un volumen se considera espacio volátil. Un volumen puede ser compartido entre varios contenedores y es persistente, separado de la imagen y del contenedor.

La gran ventaja de este desacoplamiento es que se puede asegurar la estabilidad del ambiente de trabajo, tanto para dependencias como para actualizaciones, ya que solo basta con publicar una nueva imagen que cualquier usuario pueda descargar y utilizar sin tener que cambiar nada más en la configuración de su ambiente de trabajo, como está ejemplificado en la Figura 2.

6.6. Entrega e integración continua

Este se refiere a un concepto moderno de *software* que se basa principalmente en la automatización del proceso de entrega del *software* integrado al entorno de producción o a los usuarios finales. Esto puede incluir la implementación automática en servidores, la realización de pruebas de integración y aceptación, y la notificación a los usuarios sobre la disponibilidad de nuevas versiones a través de sistemas automáticos que siempre se mantienen en línea.

Los contenedores han tenido un impacto significativo en las prácticas de entrega e integración continua (CI/CD), ya que ofrecen varias ventajas que facilitan y agilizan el proceso de desarrollo de *software*.

Los sistemas de CI/CD ofrecen excelente consistencia y repetibilidad ya que los contenedores encapsulan todo el entorno de *software*, incluyendo el código, las dependencias y las configuraciones, lo que garantiza que las aplicaciones se ejecuten de manera consistente en cualquier entorno, desde el desarrollo local hasta la producción. Esto elimina la variabilidad causada por las diferencias en los sistemas operativos, las configuraciones de *hardware* o las instalaciones de *software*. Además, la naturaleza inmutable de los contenedores garantiza que las implementaciones sean confiables y predecibles.

Otra gran ventaja de un sistema de CI/CD basado en contenedores es la increíble escalabilidad y elasticidad ya que los contenedores se pueden escalar fácilmente para satisfacer las demandas cambiantes de carga, lo que permite a las aplicaciones adaptarse a picos de tráfico o a un mayor número de usuarios. Esta elasticidad es crucial para las aplicaciones modernas que se ejecutan en la nube.

6.7. Cadena de montaje (o *pipeline*)

Una cadena de montaje, o *pipeline* en inglés, es una representación del proceso automatizado que sigue el código desde su desarrollo hasta su despliegue en producción siguiendo las instrucciones descritas en un archivo. Este proceso se puede dividir en varias etapas según sea necesario o convenientes para llegar al resultado final deseado. Cada una de las etapas realiza una tarea específica para garantizar la calidad y la consistencia del *software*. A continuación, se detallan algunas de las etapas comunes en una cadena de montaje:

1. **Compilación y pruebas unitarias:** en esta etapa, el código fuente se compila y se ejecutan pruebas unitarias para verificar su funcionalidad básica. Se asegura que el código cumpla con los estándares de calidad establecidos por el equipo de desarrollo.
2. **Pruebas de integración:** una vez que el código pasa las pruebas unitarias, se integra con el resto del sistema y se realizan pruebas de integración para verificar que las diferentes partes del *software* funcionen correctamente juntas.
3. **Pruebas de aceptación:** después de las pruebas de integración, el *software* se somete a pruebas de aceptación para validar que cumple con los requisitos del cliente y que se comporta según lo esperado.
4. **Despliegue en entornos de prueba y producción:** una vez que el *software* pasa todas las pruebas, se despliega en entornos de prueba y producción para su evaluación final y su puesta en funcionamiento.
5. **Monitoreo y retroalimentación:** una vez que el *software* está en producción, se monitorea su rendimiento y su comportamiento para identificar posibles problemas o áreas de mejora. Esta retroalimentación se utiliza para iterar y mejorar el proceso de desarrollo en futuras versiones.

Una cadena de montaje bien definida y automatizada facilita la colaboración entre equipos de desarrollo, ya que proporciona una estructura clara y procesos estandarizados para el desarrollo, pruebas y despliegue de *software*, o en este caso en la fabricación de un nanochip.

6.8. Git

Git es un sistema de control de versiones distribuido que permite a los desarrolladores rastrear y administrar cambios en el código fuente a lo largo del tiempo. A continuación se presentan algunos conceptos fundamentales de Git y su uso en un flujo de trabajo de desarrollo.

6.8.1. Conceptos básicos de Git

- ***Repositorio*:** un repositorio es un contenedor donde se almacena el código fuente y el historial de cambios.
- ***Commit*:** un *commit* representa un conjunto de cambios guardados en el repositorio.
- ***Branch*:** una *branch* (rama en inglés) es una línea de desarrollo independiente que permite trabajar en diferentes características o correcciones sin afectar la rama principal.
- ***Merge*:** el proceso de fusionar cambios de una rama a otra.
- ***Pull request*:** una solicitud para revisar y fusionar cambios de una *branch* a otra.

6.8.2. Flujo de trabajo con Git

Un flujo de trabajo típico con Git incluye los siguientes pasos:

1. Clonar el repositorio, esto da una copia exacta local de lo que exista en el repositorio remoto.
2. Crear una nueva *branch* para la característica o corrección.
3. Realizar cambios y hacer *commits*.
4. Enviar (*push*) los cambios al repositorio remoto.
5. Crear un *pull request* para revisión y fusión.

6.9. Docker

6.9.1. Creación de una imagen de Docker

Para crear una imagen de Docker, se utiliza un archivo llamado *Dockerfile*, el cual contiene una serie de instrucciones que Docker seguirá para construir la imagen. Este archivo describe la configuración del contenedor, incluyendo el sistema operativo base, las dependencias necesarias, las configuraciones de red, los comandos a ejecutar, y más. A continuación, se presenta un ejemplo básico de un *Dockerfile* para una aplicación *web* basada en Node.js:

```
1  # Usa una imagen base de Node.js
2  FROM node:18
3
4  # Establece el directorio de trabajo dentro del contenedor
5  WORKDIR /app
6
7  # Copia el archivo package.json y package-lock.json
8  COPY package*.json ./
```

```
9
10 # Instala las dependencias del proyecto
11 RUN npm install
12
13 # Copia el resto de los archivos de la aplicacion
14 COPY . .
15
16 # Expone el puerto en el que correra la aplicacion
17 EXPOSE 3000
18
19 # Define el comando para ejecutar la aplicacion
20 CMD ["node", "app.js"]
```

Cuadro 6.1: Ejemplo de *Dockerfile* básico para Node.js

Debo hacerle notar al lector que la imagen base, en este caso es una version de Linux, probablemente Alpine, la cual tiene instalado todo lo necesario para hacer funcionar Node.js. Señalo esto, ya que es importante saber que imagen derivada se hizo de la misma forma, con un *Dockerfile* [18] el cual probablemente inicia con la linea `FROM alpine:3`, o `FROM ubuntu:22`.

- Usa una imagen base de Node.js.
- Establece el directorio de trabajo dentro del contenedor.
- Copia los archivos de dependencias.
- Instala las dependencias usando npm.
- Copia los archivos restantes de la aplicación.
- Expone el puerto 3000.
- Define el comando para ejecutar la aplicación.

Para construir la imagen, se utiliza el siguiente comando:

```
1 docker build -t nombre-de-la-imagen .
```

Cuadro 6.2: Construcción de imagen a partir de Dockerfile

6.9.2. Inicializando una imagen de Docker para hacer un contenedor

Una vez que la imagen de Docker ha sido creada, se puede utilizar para iniciar un contenedor. Para esto, se utiliza el comando `docker run`. Aquí hay un ejemplo de cómo iniciar un contenedor a partir de la imagen creada anteriormente:

```
1 docker run -d -p 3000:3000 --name nombre-del-contenedor nombre-de-la-
    imagen
```

Cuadro 6.3: Lanzamiento de contenedor a partir de una imagen

Este comando hace lo siguiente:

- `-d` ejecuta el contenedor en segundo plano.
- `-p 3000:3000` mapea el puerto 3000 del contenedor al puerto 3000 del *host*.
- `-name nombre-del-contenedor` asigna un nombre al contenedor.
- `nombre-de-la-imagen` especifica la imagen de Docker a usar.

6.10. Creación de una cadena de montaje con Jenkins

Jenkins es una herramienta de automatización de código abierto que permite implementar una cadena de montaje (*pipeline*) para la integración y entrega continua de *software*. A continuación, se detallan los pasos para la instalación de Java en Linux, un requisito previo para la instalación de Jenkins.

Para instalar Java en Linux, sigue estos pasos:

```
1  # Update current packages
2  sudo dnf update
3
4  # Install Java 17 or later
5  sudo dnf install fontconfig java-17-openjdk
6
7  # Verify Java is working
8  java -version
```

Cuadro 6.4: Instalación de Java en Rocky Linux

Después de instalar Java, puedes proceder a instalar Jenkins:

```
1  # Add Jenkins repository key
2  wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key
       add -
3
4  # Add Jenkins repository
5  sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/
       apt/sources.list.d/jenkins.list'
6
7  # Update packages again
8  sudo dnf update
9
10 # Instal Jenkins
11 sudo dnf install jenkins
12
13 # Reload installed daemons
14 sudo systemctl daemon-reload
15
16 # Inicia el servicio de Jenkins
17 sudo systemctl start jenkins
18
19 # Habilita el servicio de Jenkins para que inicie automaticamente al
       arrancar el sistema
```

```
20  sudo systemctl enable jenkins
```

Cuadro 6.5: Instalación de Jenkins en Rocky

Una vez que Jenkins está instalado y en funcionamiento, puedes acceder a la interfaz *web* de Jenkins en `http://localhost:8080`.

6.10.1. Creación de las instrucciones una cadena de montaje con Jenkins

El archivo `Jenkinsfile` es donde se definen las instrucciones para la cadena de montaje. Aquí hay un ejemplo básico de un `Jenkinsfile` para una aplicación Node.js:

```
pipeline {
  agent any

  stages {
    stage('Compilacion') {
      steps {
        sh 'npm install'
      }
    }
    stage('Pruebas') {
      steps {
        sh 'npm test'
      }
    }
    stage('Empaquetado') {
      steps {
        sh 'npm run build'
      }
    }
    stage('Despliegue') {
      steps {
        sh 'docker build -t nombre-de-la-imagen .'
        sh 'docker run -d -p 3000:3000 --name nombre-del-
           contenedor nombre-de-la-imagen'
      }
    }
  }
}
```

Cuadro 6.6: Ejemplo de Jenkinsfile básico

Este `Jenkinsfile` define una pipeline con las siguientes etapas (asumiendo un proyecto de Node.js):

- **Compilación:** Instala las dependencias usando `npm install`.
- **Pruebas:** Ejecuta las pruebas unitarias con `npm test`.

- **Empaquetado:** Empaqueta la aplicación con `npm run build`.
- **Despliegue:** Construye y ejecuta el contenedor de Docker.

6.11. Distro

En el contexto de Linux, una «distro» (abreviación de «distribución») es una versión específica del sistema operativo Linux que incluye el núcleo (o *kernel*) de Linux junto con un conjunto de herramientas, aplicaciones, y una interfaz de usuario diseñada para distintos usos o tipos de usuarios. Las distribuciones son adaptaciones de Linux que vienen empaquetadas con diferentes configuraciones, aplicaciones y entornos gráficos, facilitando la elección según las necesidades del usuario, desde servidores hasta computadoras personales.

Cada distribución se diferencia por el *software* que incluye, el tipo de administración de paquetes que utiliza (como apt para Debian/Ubuntu, dnf para Fedora/Red Hat, o pacman para Arch Linux) y su enfoque en seguridad, facilidad de uso o rendimiento. Ejemplos de distribuciones populares incluyen Ubuntu, CentOS, Fedora, Arch Linux, Debian y por supuesto la que utilizamos, Rocky Linux, entre otras.

6.12. SSH

SSH (*Secure Shell*) es un protocolo de red criptográfico utilizado para establecer conexiones seguras en redes que se consideran no seguras, permitiendo la administración remota de sistemas, transferencia de archivos y ejecución de comandos en servidores de forma segura. SSH cifra la información transmitida para proteger la privacidad y la integridad de los datos, haciéndolo esencial en la administración de sistemas y en la comunicación segura entre equipos.

6.12.1. Configuración de SSH

La configuración de SSH se gestiona mediante un archivo en el directorio local de cada usuario, generalmente ubicado en `~/.ssh/config`. Este archivo permite personalizar la forma en que SSH se conecta a servidores específicos, simplificando comandos de conexión y ahorrando tiempo. En este archivo, se pueden especificar configuraciones personalizadas para servidores específicos. Por ejemplo, se pueden establecer parámetros básicos como el alias de **Host** o el **User** con el que se desea conectar a un servidor.

Sin embargo, también permite ajustes avanzados, como la selección de algoritmos de clave pública aceptados. En una configuración local, por ejemplo, se podría especificar un algoritmo de autenticación específico para un servidor en particular con `PubkeyAcceptedAlgorithms +ssh-rsa`.

6.12.2. Configuración global de SSH

A nivel del sistema, SSH también tiene una configuración global en el archivo `/etc/ssh/ssh_config`. Este archivo afecta a todas las conexiones SSH en el servidor. A continuación se detallan algunos parámetros avanzados relevantes para configuraciones de trabajo específicas, pertinentes para este trabajo:

- **MACs +hmac-sha1:** permite agregar el algoritmo HMAC-SHA1 como un algoritmo aceptado para la autenticación de mensajes. Este ajuste es importante en casos donde se requiere compatibilidad con sistemas antiguos que solo admiten este tipo de autenticación.
- **HostKeyAlgorithms +ssh-rsa:** habilita el uso del algoritmo de clave pública `ssh-rsa` como método de autenticación. Esto es relevante en entornos donde el servidor aún depende de claves RSA para la autenticación.
- **PubkeyAcceptedKeyTypes +ssh-rsa y PubkeyAcceptedAlgorithms +ssh-rsa:** ambos parámetros permiten aceptar claves públicas del tipo RSA como algoritmos válidos para autenticación. Son útiles para habilitar el acceso en configuraciones que dependen de la compatibilidad con RSA.

Mientras que configuraciones locales en `ssh_config` se aplican solo a sesiones específicas del usuario, las configuraciones globales en `ssh_config` aseguran que las políticas de acceso y autenticación a nivel de servidor cumplan con los requisitos de seguridad y compatibilidad para todos los usuarios. Para más información relacionada con el uso de OpenSSH, referirse a la documentación oficial [19].

CAPÍTULO 7

Análisis y mejora del *script* de instalación para entornos Linux: evaluación de herramientas y mejores prácticas

El *script* de instalación actual, desarrollado por Paola Mendizábal, ha demostrado ser funcional en los usos a los que se le ha limitado hasta el momento. Sin embargo, su diseño asume varias características del sistema en el que se ejecuta, lo que lo hace susceptible a errores. Estas suposiciones incluyen dependencias no documentadas, como *scripts* y configuraciones del entorno de trabajo que no suelen estar presentes en todos los sistemas. Además, el *script* no verifica los resultados de los comandos ejecutados ni cuenta con un mecanismo para detectar y manejar errores que podrían detener su ejecución automáticamente, lo que aumenta el riesgo de fallos inesperados sin proporcionar al usuario ninguna explicación sobre la causa del fallo.

En cuanto a la selección de herramientas utilizadas dentro del script, se ha observado que algunas, como `lftp`, aunque efectivas, pueden no ser las más adecuadas para un entorno generalizado. Existen herramientas más comunes, incluidas en las instalaciones estándar de la mayoría de las distribuciones de Linux, que son más accesibles y tienen un mayor soporte en términos de documentación. La adopción de estas herramientas podría reducir la complejidad del *script* al eliminar la dependencia de soluciones específicas y facilitar la transición a otros entornos si esto fuera necesario.

Aunque el *script* actual intenta mantener un orden lógico mediante el uso de variables, directorios y funciones, la falta de documentación detallada es un problema significativo. Sin una guía clara, cualquier usuario que intente simplemente ejecutar el *script* enfrentará dificultades. Se considera que en el proceso de instalación ya existe suficiente complejidad y abstracción como para que se añada una capa más mediante el uso del script.

7.1. Cambios en los estándares de seguridad y actualización del proceso de instalación

A partir de la versión RHEL 8.2, se desactivaron ciertos estándares criptográficos que se consideran obsoletos en la configuración predeterminada de Rocky Linux, una política que también se aplica a las versiones de Rocky Linux desde la versión 8.6. Esta actualización es crucial para mantener la seguridad, pero presenta un desafío al interactuar con servidores que aún utilizan criptografía *legacy*, como es el caso del servidor SFTP de Synopsys. Para establecer una conexión exitosa con este servidor, es necesario configurar manualmente herramientas como `SSH` para aceptar estos estándares criptográficos antiguos, lo que añade una capa de complejidad al proceso. Para resolver este problema, se han desarrollado archivos de configuración específicos que pueden ser fácilmente incorporados en cualquier *distro* de Linux. Estos archivos permiten habilitar las configuraciones de seguridad *legacy* necesarias sin comprometer la seguridad general del sistema. Esta solución asegura que las conexiones con servidores antiguos puedan mantenerse sin debilitar otros aspectos críticos de la seguridad del sistema.

7.2. Configuración de seguridad de SSH para la conexión con SFTP de Synopsys

De manera simple, lo único que se debe hacer para rehabilitar los estándares de encriptación `ssh-rsa` es agregar las siguientes directivas a la instalación de Linux. Estas son independientes del *distro* utilizado y se puede escoger entre un cambio global, un cambio individual para cada usuario o incluso para cada acceso remoto. Para más información sobre estas configuraciones, se recomienda consultar la sección relevante en el marco teórico.

7.2.1. Configuración global

Es necesario crear un archivo en la carpeta `/etc/ssh/ssh_config.d/`, nombrándolo de forma descriptiva y comenzando con un número. En este caso, el archivo fue nombrado como `10_synopsys_enable_rsa.conf`. El archivo debe contener lo siguiente:

```
1  MACs +hmac-sha1
2  HostKeyAlgorithms +ssh-rsa
3  PubkeyAcceptedKeyTypes +ssh-rsa
4  PubkeyAcceptedAlgorithms +ssh-rsa
```

Cuadro 7.1: Parámetros de configuración SSH globales

7.2.2. Configuración individual

De manera similar, estos parámetros de configuración pueden definirse para un único *host*. Esto debe realizarse en el archivo `~/.ssh/config`. Si dicho archivo no existe, es completamente seguro crearlo con el editor de texto de preferencia. En caso de que el archivo

ya exista, este bloque puede agregarse en cualquier parte del archivo, asegurándose de no dividir configuraciones previamente escritas. Por convención, se sugiere colocarlo al principio o al final del archivo.

```
1 Host eft.synopsys.com
2     MACs +hmac-sha1
3     HostKeyAlgorithms +ssh-rsa
4     PubkeyAcceptedKeyTypes +ssh-rsa
5     PubkeyAcceptedAlgorithms +ssh-rsa
```

Cuadro 7.2: Parámetros de configuración SSH locales

Para confirmar la correcta funcionalidad de estos parámetros, se debería obtener un *login prompt* exitoso al intentar ingresar por SSH al servidor EFT de Synopsys. A continuación, se incluye una sección de la consola que demuestra lo anterior.

```
1  # Previo al cambio de SSH
2  ssh eft.synopsys.com
3  Unable to negotiate with 149.117.73.40 port 22: no matching host key type
       found. Their offer: ssh-rsa
4
5
6  # Intento con RSA habilitado para SSH
7  ssh eft.synopsys.com
8  You have connected to the synopsys EFT server.
9
10 Information about .. cortado .. If the technical data is otherwise
       classified you
11 must provide alternative export control information before submitting
12 John@eft.synopsys.com's password:
```

Cuadro 7.3: Demostración de configuracion de SSH aplicada funcionando correctamente

7.3. Descarga FTP e instalación manual con interfaz gráfica

De acuerdo con la propuesta de mantener el uso de herramientas lo más estándar posible, se presenta una forma mucho más sencilla de instalar los programas de Synopsys. Iniciando con la descarga, a continuación se muestran los pasos a seguir comenzando con la descarga FTP mediante una interfaz gráfica.

Una vez que se haya configurado SSH correctamente, se puede aprovechar el uso del *File Explorer* para conectarse gráficamente al servidor y descargar el *software* requerido con un simple *drag and drop*.

Utilizando el explorador de archivos, es posible conectarse a servidores externos. Véanse las Figuras 3 y 4, que muestran el menú relevante para la conexión a Synopsys.

En esta ventana, con la opción de *Other Locations* seleccionada, se puede ingresar la URL del servidor de Synopsys, y enseguida el servidor presentará un *modal* para ingresar la contraseña, tal como se muestra en la Figura 5. Al ingresar la contraseña correcta, será posible listar archivos y arrastrar las carpetas del servidor directamente al *folder* local de preferencia (véase la Figura 6).

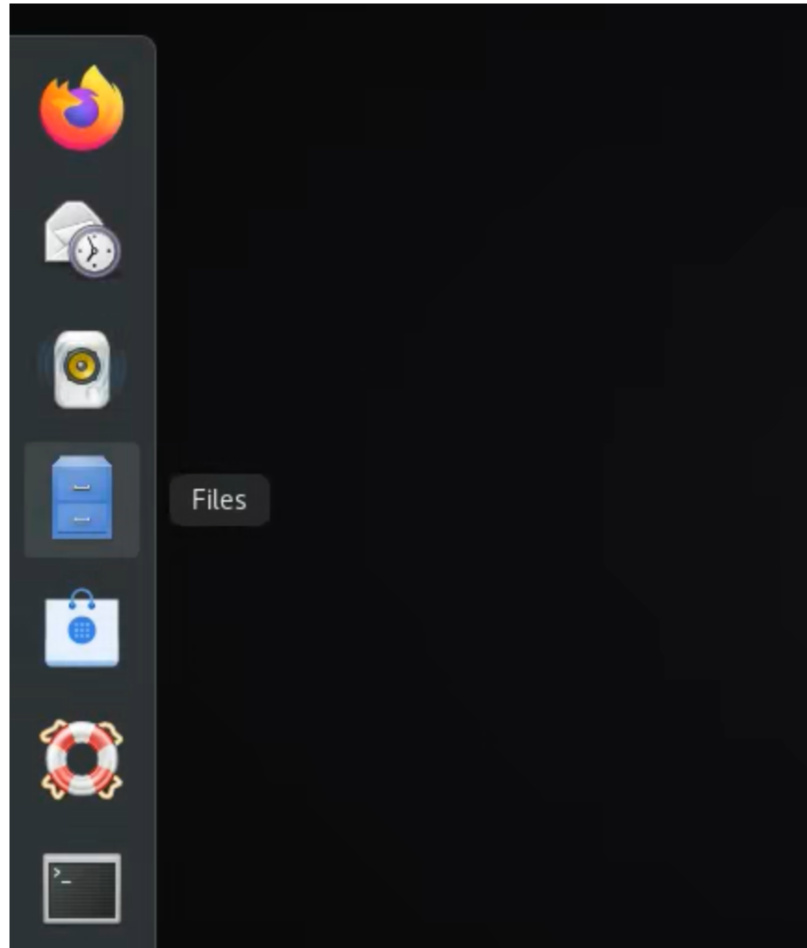

Figura 3: Explorador de archivos por defecto de GNOME

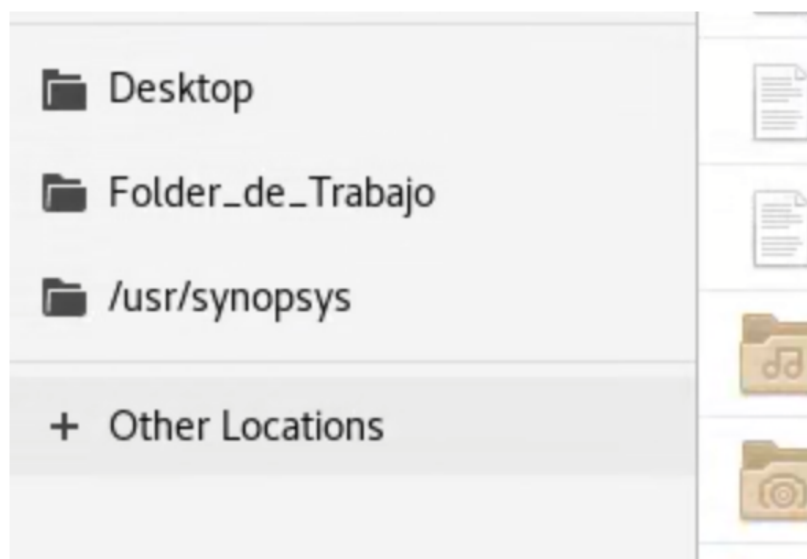

Figura 4: Menú de *Other Locations* seleccionado.

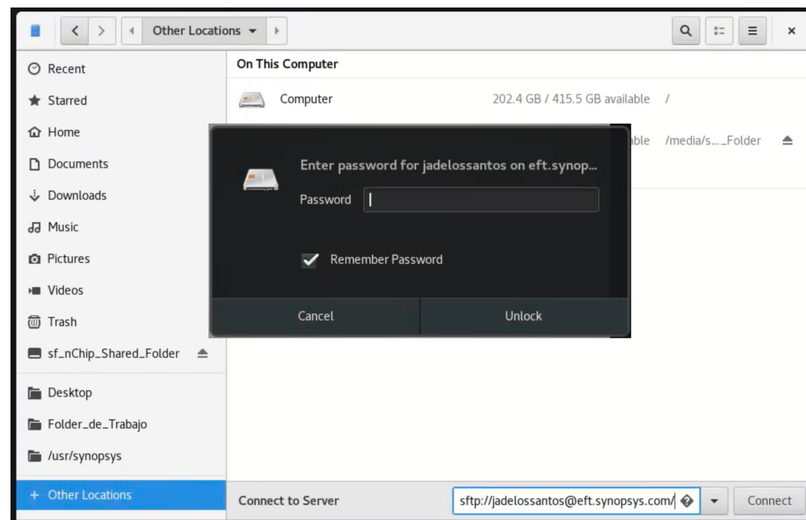

Figura 5: Servidor EFT de Synopsys requiriendo contraseña.

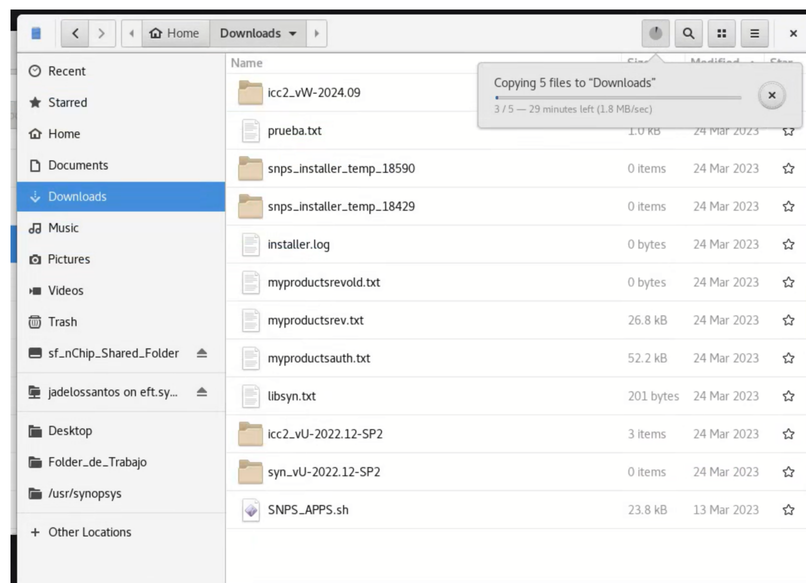

Figura 6: Descarga iniciada con indicador visual directamente en el explorador de archivos

CAPÍTULO 8

Evaluación de Apptainer como alternativa a Docker: instalación, pruebas y comparación de características

Apptainer presenta una notable flexibilidad en cuanto a su compatibilidad con múltiples sistemas operativos, incluyendo Windows, macOS y Linux. Sin embargo, es importante destacar que las versiones precompiladas actualmente solo están disponibles para Linux, lo que puede limitar su accesibilidad inmediata para usuarios de otros sistemas operativos. A pesar de esta limitación, su capacidad de ser instalado en los tres principales sistemas operativos ofrece una ventaja significativa para quienes buscan una solución de contenedorización adaptable a diversos entornos.

Durante las pruebas iniciales, se utilizó un sistema Ubuntu 22.04, lo que permitió realizar varias observaciones relevantes sobre el funcionamiento de Apptainer. Una de las características más útiles es su capacidad para utilizar repositorios públicos de Docker, permitiendo la descarga de imágenes a través del comando `apptainer pull docker://rockylinux:9`. Este proceso genera un archivo con extensión `.sif`, que es el formato de contenedor utilizado por Apptainer. Esta funcionalidad es especialmente valiosa, ya que permite aprovechar el extenso ecosistema de imágenes de Docker para la construcción y gestión de contenedores, integrando de manera eficiente ambas plataformas. Esta funcionalidad será de gran utilidad más adelante.

Una de las ventajas más notables de Apptainer, en comparación con Docker, es la transparencia en su arquitectura y diseño. Mientras que Docker tiende a aislar los sistemas de manera estricta, creando todo un *filesystem* virtual, Apptainer permite un acceso directo a los archivos del usuario actual cuando se inicializa un contenedor, montando como carpeta base de usuario la carpeta donde se inicializa el contenedor, o la carpeta `home` del usuario actual si se utiliza la opción `--fakeroot`. Esta característica facilita la interacción entre el contenedor y el sistema operativo host, ofreciendo una mayor flexibilidad en la gestión de datos, ya que todos los archivos que se deben utilizar se pueden encontrar en los mismos

directorios relativos dentro del contenedor que en la computadora del usuario, lo cual puede ser de gran utilidad para el objetivo de este proyecto. No obstante, Apptainer también proporciona opciones para crear directorios que existen únicamente dentro del contenedor, lo que permite una mayor seguridad y aislamiento cuando es necesario. Estas dos formas de trabajo serán exploradas con mayor detalle en secciones posteriores de este trabajo.

Apptainer también permite escribir archivos de definición, conocidos como *definition files*, que son análogos a los *Dockerfiles* en el ecosistema de Docker. Estos archivos permiten especificar los pasos necesarios para construir una imagen personalizada, lo que representa una gran ventaja para usuarios que requieren un mayor control sobre la configuración de sus entornos. Esta característica está en línea con el diseño general de Apptainer, donde cada contenedor se asemeja más a un *script* de automatización que a una máquina virtual tradicional. Esta filosofía de diseño simplifica la creación y gestión de contenedores, los cuales pueden ejecutar procesos predefinidos *one off* repetidamente, de manera similar a un brazo robótico en una línea de ensamblaje.

Una clara desventaja que se ha identificado al crear los archivos de definición es que, a diferencia de Docker, Apptainer no guarda capas intermedias entre pasos de construcción, lo que alarga el proceso de agregar nuevas dependencias a la imagen, ya que cualquier cambio implica recrear la imagen desde cero, reinstalando todas las dependencias. Existe una forma sencilla de superar esta limitación: se debe crear un *Dockerfile* con las mismas dependencias, separadas en múltiples directivas *RUN*, para que Docker guarde cada capa por separado. Posteriormente, se crea la imagen del contenedor con un nombre personalizado y se utiliza esta imagen como referencia en el *definition file* de Apptainer.

8.1. Creación de contenedores en Apptainer

8.1.1. Introducción a Apptainer

Al momento de escribir este documento, se utiliza la versión 1.3.5. Se recomienda revisar la documentación directamente en su sitio [20], en caso de utilizar una versión más moderna o si se presenta algún problema no especificado en este documento.

En este capítulo, se presentan los pasos fundamentales para inicializar y gestionar un contenedor en Apptainer. Estos pasos guiarán al usuario desde la creación básica hasta la ejecución de procesos dentro del contenedor, permitiendo una integración fluida en el flujo de trabajo de automatización.

Como ejemplo demostrativo, se indican los comandos para descargar la imagen de Ubuntu versión 20.04 desde los repositorios de Docker y luego ejecutar *bash* dentro de esta.

```
1  # Paso 1: Hacer pull de una imagen desde Docker Hub
2
3  # Este comando descarga una imagen desde Docker Hub y la convierte en
       formato SIF.
4  sudo apptainer pull docker://ubuntu:20.04
5
6  # Paso 2: Ejecutar la imagen descargada
7  # Una vez descargada, se puede ejecutar directamente la imagen en
```

```
    Apptainer.
apptainer shell ubuntu_20.04.sif

# Alternativamente, indicar el programa a ejecutar
apptainer exec ubuntu_20.04.sif /bin/bash
```

Cuadro 8.1: Creación de una imagen básica de Apptainer basado en Docker Hub

Para cumplir los objetivos de este proyecto, únicamente es necesario utilizar las imágenes de `rocky:8` y/o `rocky:9`. En caso de requerir otra imagen, se puede buscar con gran facilidad utilizando el repositorio de imágenes público de Docker, el *Docker Hub*, el cual es extremadamente extenso y se puede acceder en [21].

Una vez inicializado un contenedor de la forma mencionada, la consola cambia, indicando que se está utilizando el ambiente virtualizado al inicializar el *prompt* con el texto `Apptainer>`. Dentro de esta consola, se dispone de todos los programas, librerías y demás elementos que se especificaron al crear la imagen. Esto crea un contenedor que ejecuta el comando descrito y luego desaparece.

La siguiente progresión en el ecosistema de Apptainer es el uso de *runscripts*, los cuales son comandos predefinidos como parte del contenedor que se ejecutan si no se especifica ningún comando al inicializar el contenedor. Una forma sencilla de comprobar esto es con la imagen demostrativa de *lolcow*, la cual únicamente empaqueta el comando *cowsay* dentro de la imagen.

```
  # Step 1: Pull an image from Docker Hub

# This command downloads an image from Docker Hub and converts it to SIF
    format.
sudo apptainer pull docker://ghcr.io/apptainer/lolcow

# Step 2: Execute the cowsay command with random text
apptainer exec lolcow_latest.sif cowsay "MechEng UVG"
_____
< MechEng UVG >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

  # If the container is run without specified text, it does not fail.
# Instead, it displays today's date.

_____
< Sun Nov 11 18:20:55 CST 2024 >
 -----
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Cuadro 8.2: Demostración de configuración por defecto de un contenedor en Apptainer

8.1.2. Definiendo un contenedor personalizado

Como se menciona anteriormente, para crear un contenedor personalizado se debe escribir un archivo de definición, similar a un *Dockerfile*. Este archivo define los pasos determinísticos que se deben ejecutar en el sistema virtual para satisfacer las necesidades del uso. A continuación, se sigue un enfoque incremental, construyendo sobre los pasos y conceptos previamente mencionados.

Anteriormente se observó que un contenedor parte de una imagen, ya sea directamente de Docker o de un archivo *.sif*. De igual manera, para construir un archivo *.sif* propio, se puede definir una base conveniente sobre la que se ejecutarán los demás pasos de instalación. Para ejemplificar de forma sencilla, se puede considerar la imagen de Debian [22] y la imagen de Ubuntu [23]. Si bien Debian es más ligero, Ubuntu incluye más programas por defecto. Por otro lado, si las necesidades son mínimas, se puede considerar el uso de una imagen ligera, especificando los programas específicos a instalar. Este es el caso dentro del ecosistema de Docker con el extenso uso de las imágenes extremadamente ligeras de *Alpine Linux*, las cuales llegan a tener tamaños finales de 50 megabytes o menos.

Construyendo hacia pasos posteriores, se utiliza Rocky Linux versión 8 como base para el ejemplo.

```
1   $ cat sample-definition.def
2
3   # Sample definition file for UVG by Juan Pablo Mora
4   Bootstrap: docker
5   From: rocky:8
6   Stage: build
7
8   %setup
9     # This section will run the commands in the HOST machine
10  echo "Starting the image build process..."
11
12  %environment
13  export MY_MESSAGE=${MY_MESSAGE:-'Hello World from Guatemala'}
14
15  %post
16  # This section will run within the container
17  dnf update -y
18  dnf install -y lftp wget
19
20  %startscript
21  # This will be run when the container starts,
22  echo $MY_MESSAGE
23
24  %labels
25    Author mor15799@uvg.edu.gt
26    Version v0.0.1
27
28  %help
29    Base container, created from Rocky 8 to run \gls{synopsys} tools.
```

Cuadro 8.3: Definición demostrativa de una imagen de Apptainer.

8.1.3. Bootstrap y from

Las primeras líneas del archivo de definición establecen el tipo de sistema base del contenedor. La palabra clave `Bootstrap` indica el origen del sistema base que se utilizará, en este caso, una imagen de Docker. La línea `From: rocky:8` especifica que la imagen base será Rocky Linux en su versión 8.

8.1.4. Sección %setup

La sección `%setup` permite ejecutar comandos en el sistema anfitrión (**tomar nota**, fuera del contenedor) durante el proceso de construcción. En este ejemplo, se utiliza un comando `echo` para mostrar un mensaje inicial que indica el inicio del proceso de construcción de la imagen. Esta sección puede ser útil para realizar configuraciones previas o montar directorios desde el sistema anfitrión que luego serán accesibles desde dentro del contenedor.

8.1.5. Sección %environment

La sección *%environment* define las variables de entorno que estarán disponibles en el contenedor una vez iniciado. Estas variables funcionan de manera similar a las definidas en archivos como `.bashrc` o `.profile` en sistemas basados en Linux. En este ejemplo, `MY_MESSAGE` es una variable de entorno que se inicializa con el valor por defecto `'Hello World from Guatemala'`, a menos que se especifique otro valor al iniciar el contenedor, exportándolo en la consola donde se invoca. Este enfoque permite una personalización flexible de los contenedores sin tener que modificar el archivo de definición.

8.1.6. Sección %post

La sección `%post` contiene comandos que se ejecutarán dentro del contenedor durante su construcción. En este ejemplo, se utiliza el administrador de paquetes `dnf` para actualizar los repositorios de Rocky Linux e instalar los paquetes `lftp` y `wget`. Estos comandos son esenciales para preparar el entorno dentro del contenedor con las dependencias necesarias para futuras tareas.

8.1.7. Sección %startscript

Finalmente, la sección `%startscript` define un *script* que se ejecuta cada vez que se inicia el contenedor. En este caso, el comando `echo $MY_MESSAGE` imprime el valor de la variable de entorno `MY_MESSAGE` al inicio. Esta funcionalidad permite personalizar el comportamiento inicial del contenedor, siendo útil para mostrar mensajes de bienvenida, iniciar servicios o cargar configuraciones específicas al iniciar el contenedor.

CAPÍTULO 9

Uso de programas de Synopsys dentro de Apptainer

Synopsys y varias de sus herramientas incluyen soporte nativo para utilizarse dentro de un contenedor de Apptainer. Afortunadamente, todas las herramientas utilizadas en el proyecto Gran Jaguar están en esta lista de programas con soporte, por lo que se puede aprovechar en gran medida esta funcionalidad, confiando en el soporte que brinda el proveedor.

Es de gran importancia notar que Synopsys define una forma muy específica de trabajar con sus programas dentro de contenedores. Para el usuario final, basta con agregar la bandera `-container` a cualquier comando, lo que hará que este, de manera inmediata y transparente, ejecute el proceso indicado dentro de un contenedor.

Para lograr esta configuración transparente del uso de contenedores, se deben seguir pasos específicos indicados por Synopsys, ya que no es el usuario, sino el programa, quien inicializa el contenedor donde se ejecuta el proceso. A continuación, se describen los pasos para configurar Apptainer de forma funcional en Rocky 8.9 con las herramientas de Synopsys.

9.1. Instalación de Synopsys Container

Utilizando el explorador de archivos, se descarga la versión 2 de Synopsys Container, que es compatible con la versión más reciente de *Synopsys Installer*, que al momento de la escritura es la v5.8.1 (véase la Figura 7). Colocar el directorio en una ruta conocida y lanzar el *Synopsys Installer*, siguiendo todos los pasos y seleccionando las rutas y opciones por defecto, o aquellas que reflejen la instalación actual del sistema (véase la Figura 9).

El *Synopsys Installer* busca automáticamente ciertos directorios para determinar si existe *software* listo para ser instalado. En caso de que el instalador haya encontrado el folder,

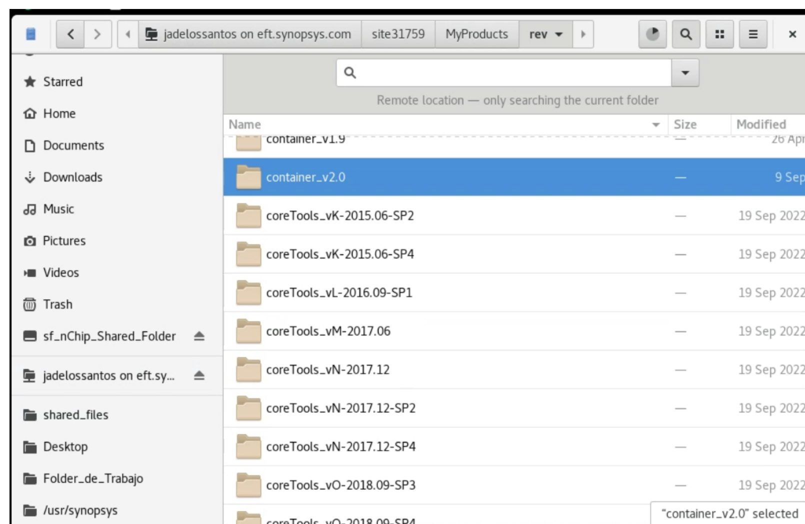

Figura 7: Explorador de archivos mostrando la carpeta de Container V2

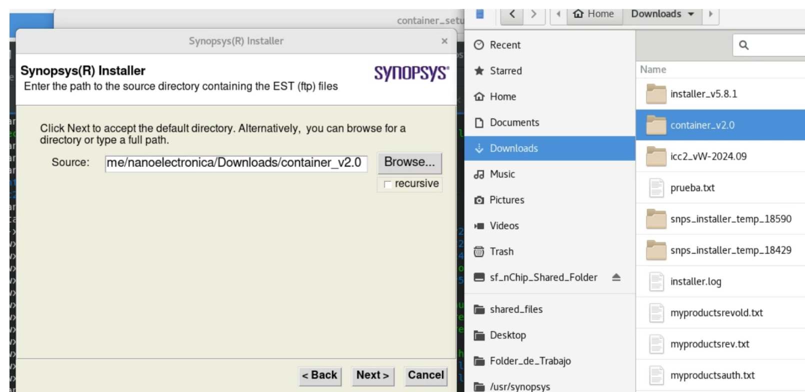

Figura 8: Instalador de Synopsys mostrando la carpeta seleccionada.

verificar que sea la ruta correcta (véase la Figura 8) y continuar con la instalación.

Nota: todos los programas propuestos para usarse con Synopsys Container deben estar instalados previamente. En el caso que se deba instalar un programa en el futuro, se deben repetir los pasos de instalación de Synopsys Container.

El proceso es bastante directo; sin embargo, vale la pena revisar dos pasos en específico:

- El directorio desde el que se instalará el *software*, en este caso Container V2, y
- El directorio destino o *root* de instalación de Synopsys. En este caso, el directorio es `/usr/synopsys`.

Si los pasos se han seguido correctamente, *Synopsys Container* estará instalado en el directorio raíz (véase la Figura 10). Finalmente, Synopsys provee un *script* (véase la Figura 11) que creará un archivo de configuración y lo copiará a cada programa instalado, de tal forma que los programas sepan qué hacer y qué configuración seguir al momento de utilizar la bandera `-container`.

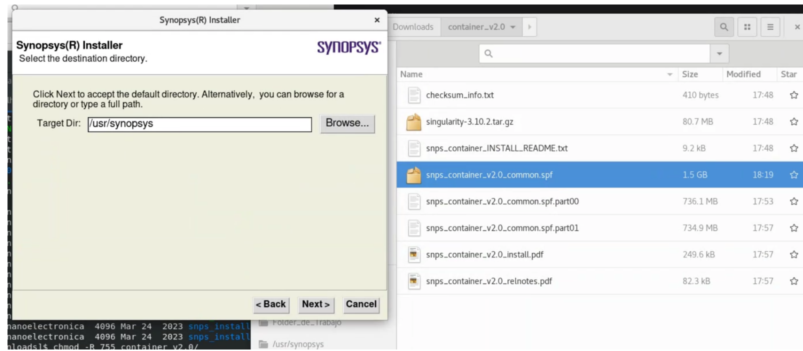

Figura 9: Instalador de Synopsys mostrando la carpeta base de instalación.

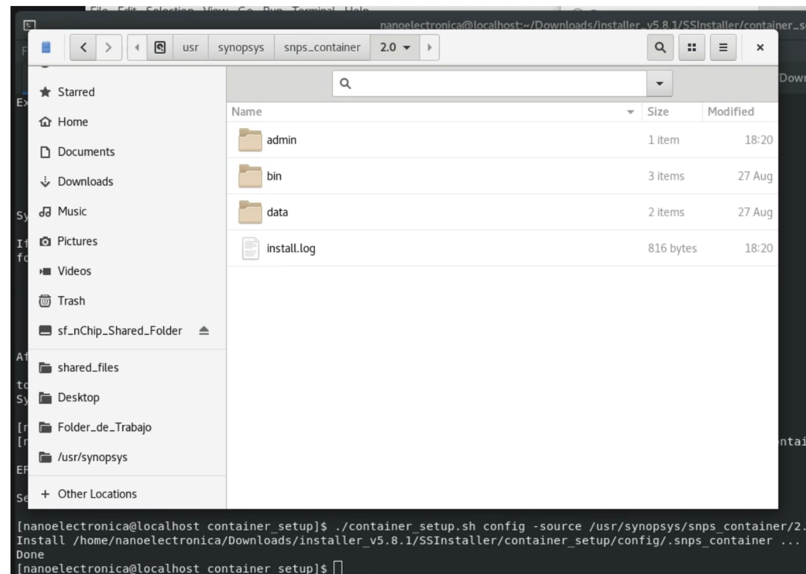

Figura 10: Instalador de Synopsys mostrando la carpeta base de instalación.

La ejecución del *script* es bastante sencilla. Las banderas y opciones disponibles están explicadas claramente en la documentación de Synopsys, provista en formato PDF dentro del directorio de instalación. El comando utilizado en este caso es el siguiente:

```
1  # Generating a \gls{synopsys} Container config with
2  # - A specified image
3  # - Folder bind mounts, these get mounted on the same relative
4  #   location within the container. eg /mnt/ becomes /mnt/ within the
     container.
5  $ ./container_setup.sh config -image centos7.simg -bind /mnt,/home/
     nanoelectronica\
6    /Downloads/Installer_v5.8.1/SSInstaller/container_setup/testmount
```

Cuadro 9.1: Uso de *script* para generar configuración de Synopsys Container.

Es importante notar ciertos aspectos al momento de ejecutar el comando anteriormente detallado. Si bien es sencillo de usar, la imagen debe especificarse únicamente por nombre. En el caso anterior, se realizó una prueba con CentOS 7. Esta definición de imagen debe estar en el directorio `/usr/synopsys/snps_container/2.0/data/`. Si se utilizan las

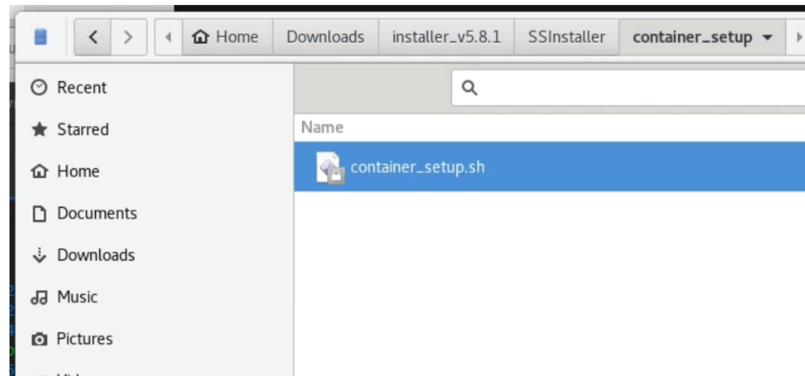

Figura 11: Script de configuración de Synopsys Container.

banderas correctamente y el comando se ejecuta sin problemas, el *script* genera un archivo de configuración bajo `./config/.snps_container` (véase la Figura 12). Este archivo debe moverse a cada instalación de los programas de Synopsys. El *script* provisto se encarga de este proceso, ejecutándose como se muestra a continuación.

La configuración a instalar fue generada con los siguientes parámetros:

```
1  $ ./container_setup.sh config -image rocky89_uvg.sif -bind /lib64,/mnt,/
     usr/synopsys
```

Cuadro 9.2: Uso de *script* para generar configuración de Synopsys Container con datos reales

Y su despliegue o instalación se ejecuta de esta forma:

```
1  # Install the generated config to all the synopsys apps
2  $ ./container_setup.sh deploy -target /usr/synopsys/
```

Cuadro 9.3: Uso de *script* para instalar configuración de Synopsys Container

El *script* detalla el proceso de instalación (véase la Figura 13). Este proceso debe repetirse siempre que sea necesario cambiar la configuración de Apptainer. El cambio más común será reemplazar la imagen, ya que, en lugar de CentOS 7, se utilizará la imagen creada por el usuario.

Se sugiere establecer un nombre de imagen que sea lógico, sencillo y corto. Además, cualquier prueba o actualización requerida debe realizarse guardando la imagen con el mismo nombre. Esto garantiza que no sea necesario repetir la configuración de Synopsys Container.

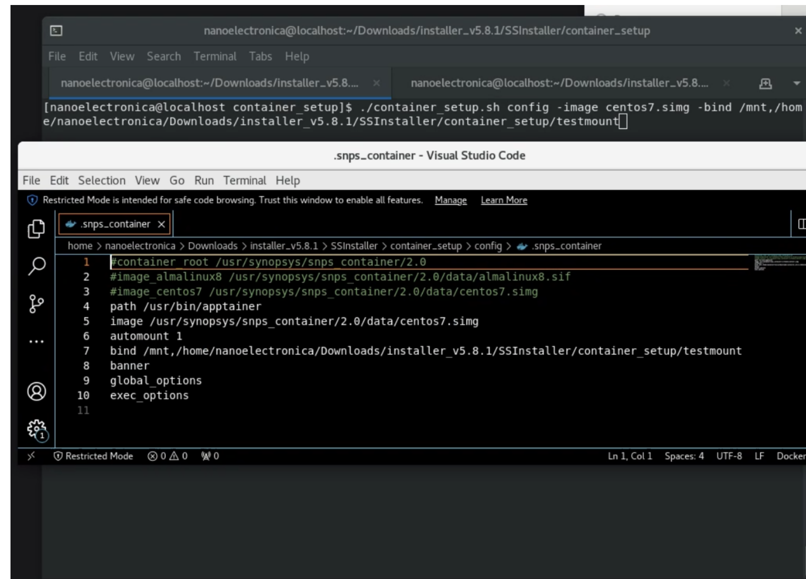

Figura 12: Configuración generada por el *script* de Synopsys Container.

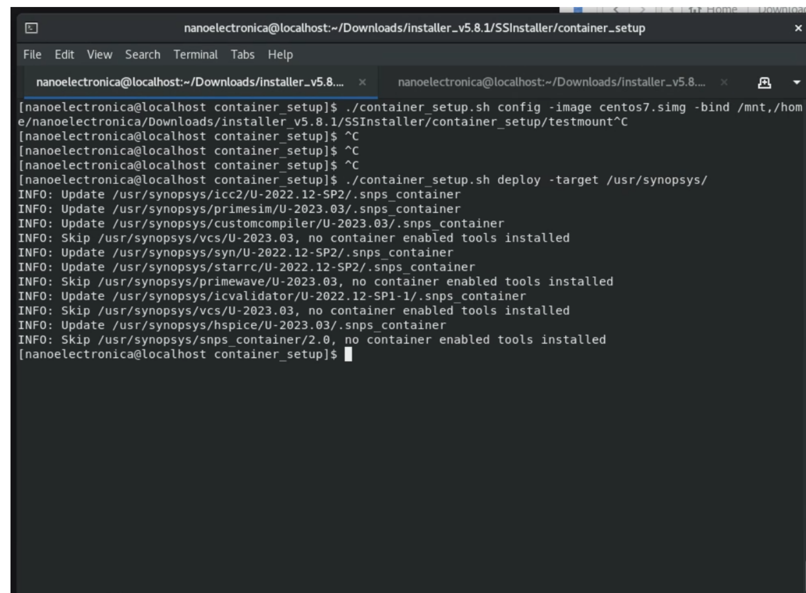

Figura 13: *Output* del proceso de instalación de la configuración creada por Synopsys Container.

9.2. creación de una imagen apta para ejecutar los programas de Synopsys

Una vez que se han instalado y configurado las partes y programas mencionados anteriormente, solo resta dedicarse a crear una imagen y configuración apta para que los programas de Synopsys se ejecuten correctamente. Gran parte de este proceso fue basado únicamente en hallazgos por prueba y error. Sencillamente, se inicia con una definición de contenedor

como la propuesta anteriormente y se instala en el directorio correspondiente de imágenes.

Después de esto, se busca lanzar cualquier programa de Synopsys utilizando la bandera `-container`. Simplemente llamando al comando es suficiente para que, si el programa se ejecuta exitosamente, despliegue los términos y condiciones de Synopsys. En caso contrario, se mostrará algún error, seguramente relacionado con alguna librería faltante. En este caso, una de ellas fue `libXft.so.2`. Al notar esto, se realiza una búsqueda sencilla en Internet para localizar el nombre del paquete que proveerá dicha librería.

El comando de instalación debe agregarse a la definición de la imagen antes de recrearla. Siguiendo el ejemplo de las secciones anteriores, se deberá cambiar una línea de la siguiente forma.

```
1  # ... truncated
2
3  %post
4  # This section will run within the container
5  dnf update -y
6
7  # We add our newly discovered dependency libXft to the list
8  dnf install -y lftp wget libXft
9
10 # ... truncated
```

Cuadro 9.4: Actualización de archivo de definición para instalar nuevas librerias o dependencias.

La definición que finalmente me permitió inicializar los programas de Synopsys se detalla a continuación.

```
1    $ cat snps_uvg_rocky89.def
2  Bootstrap: docker
3  From: rockylinux:8.9
4
5  %post
6    # This section runs during the build process
7    # Install basic packages and Perl
8    dnf update -y
9    dnf -y install perl lftp openssh-clients rsync glibc-locale-source \
10   langpacks-en langpacks-es nano xz wget
11   dnf -y install libX11 libXft libnsl
12   dnf clean all
13
14 %runscript
15   # This script runs by default when you execute the container
16   exec /bin/bash "$@"
17
18 %labels
19   Author mor15799@uvg.edu.gt
20   Version v0.0.1
21
22 %help
23   Base container, created from Rocky 8 to run synopsys tools.
```

Cuadro 9.5: Definición de Rocky 8.9 utilizada para generar la imagen compatible con Synopsys Container.

CAPÍTULO 10

Depuración de imágenes y contenedores de Apptainer

Este documento tiene como objetivo proporcionar información que esté completamente lista para ser utilizada, incluyendo comandos, resultados y configuraciones, con el propósito de optimizar el tiempo empleado en el diseño de nanoelectrónica. Por esta razón, se incluye un capítulo dedicado a la depuración del proceso, considerando la importancia de estar preparados para resolver cualquier imprevisto.

La creación de contenedores e imágenes es un proceso relativamente sencillo para quienes poseen conocimientos básicos del uso de la terminal en Linux. Sin embargo, pueden surgir casos en los que la resolución de problemas de dependencias se torne tediosa, al requerir intentos repetitivos de ejecutar un programa, identificar errores, recrear la imagen y repetir el proceso. Por ello, se presentan diversas alternativas para construir imágenes y contenedores temporales, con el fin de acelerar significativamente la resolución de estos problemas.

10.1. Creación de una imagen de Apptainer con permisos de *root*

Normalmente, un contenedor inicializará el proceso o consola que se le indique bajo los permisos del usuario actual. Por lo tanto, si se deben instalar dependencias adicionales en el contenedor, no será posible ejecutar `dnf install ...`. Además, utilizar `sudo` o la bandera `--fakeroot` únicamente hará notar que el sistema es de solo lectura, lo cual no se puede resolver sin agregar complejidad y emplear *overlays* en los cuales el sistema pueda escribir [24].

La solución propuesta es crear un contenedor con sistema de archivos en modo *sandbox*. En esta configuración, Apptainer creará el contenedor en un folder, en lugar de generar el archivo `.sif`. Para Apptainer, la imagen funciona exactamente igual, y los comandos vistos

anteriormente operan de manera transparente. Además, se gana la ventaja de poder reiniciar el contenedor utilizando las banderas `--fakeroot` junto con `--writable` para instalar nuevas dependencias sin necesidad de reconstruir la imagen.

El *workflow* óptimo con el *sandbox* consiste en tener dos consolas abiertas: una normal y otra con permisos de *root*. De esta forma, se pueden instalar programas en una y comprobar su funcionamiento en la otra.

Nota: Desafortunadamente, Synopsys Container verifica que el archivo `.sif` seleccionado sea un archivo, por lo que no es posible usar una imagen en modo *sandbox* directamente como contenedor de Synopsys.

```
# First we build the image based on our definition file
apptainer build rocky89_uvg.sif snps_uvg_rocky89.def

# Then we extract the filesystem into our sandbox
apptainer build --sandbox my-rocky-sandbox.sif/ rocky89_uvg.sif

# Run the new container transparently
apptainer exec --fakeroot --writable my-rocky-sandbox.sif/ /bin/bash

# (Optional) Convert the sandbox back to image
apptainer build new_rocky89_uvg.sif my-rocky-sandbox.sif/
```

Cuadro 10.1: Creación de imagen de Apptainer basado en su archivo de definición.

Finalmente, después de terminar las pruebas deseadas, se puede tomar nota de los procesos, archivos, configuraciones y dependencias que hicieron exitosa la imagen en modo *sandbox* para agregarlas a la definición del contenedor o bien, convertir la imagen *sandbox* nuevamente a un archivo `.sif`. Se recomienda al lector mantener un *definition file* actualizado en todo momento para garantizar que el proceso, incluso durante las pruebas, sea determinístico y con la ventaja de poder utilizar sistemas de control de versiones como `git`.

CAPÍTULO 11

Depuración de *scripts* de Synopsys

Basado en el uso constante que se le ha dado a la mayoría de programas de Synopsys, se observa que todos contienen varias dependencias, no solo de *software* y librerías, sino también de configuraciones y variables de entorno específicas. A pesar de esto, estos programas tienden a evitar en gran medida generar *logs* o reportes que permitan entender los errores para poder solucionarlos.

Durante la contenedorización de las aplicaciones, surgieron muchos problemas y, por ende, hubo muchos errores que solo podrían definirse como crípticos. Afortunadamente, se encontró una forma bastante eficiente de identificar y comprender los procesos, binarios y banderas que cada programa utiliza al ejecutar los comandos.

11.1. Variables de entorno para depuración

SNPS_CONTAINER_TRACE=1

Esta variable activa un modo de traza detallado en contenedores Synopsys. Al habilitarla, es posible observar cada comando ejecutado por el contenedor, incluyendo las rutas, argumentos y dependencias que el programa requiere. Estos datos son útiles al momento de depurar cualquier *script* que se desee crear.

SNPS_CONTAINER_VERBOSE=1

Esta variable amplifica la salida estándar de los programas Synopsys al ejecutarse dentro de contenedores. En combinación con `SNPS_CONTAINER_TRACE=1`, proporciona un nivel

adicional de detalle sobre los procesos internos.

Depuración en *vcs* con `VCS_SCRIPT_DEBUG`

La variable `VCS_SCRIPT_DEBUG` permite depurar la ejecución de *scripts* dentro de `vcs`, mostrando las líneas ejecutadas y los argumentos utilizados. El comportamiento de esta bandera depende de su valor:

- `show_args`: si el valor es `show_args`, el *script* imprime los argumentos que se pasan al comando `vcs`, facilitando el análisis de parámetros específicos y banderas utilizadas.
- **Otros valores**: si `VCS_SCRIPT_DEBUG` toma cualquier otro valor, se habilita el modo de depuración de bash con `set -x`, mostrando cada línea ejecutada junto con información adicional como el ID de proceso (`$$`), número de línea (`LINENO`) y nombre de la función (`FUNCNAME[0]`).

Este nivel de depuración es crucial para diagnosticar problemas en *scripts* complejos, permitiendo al desarrollador localizar rápidamente errores en lógica, rutas o argumentos.

CAPÍTULO 12

Instalación de servidor de Jenkins

La instalación de Jenkins es el primer paso para configurar un servidor de integración continua. A continuación, se detallan los pasos para instalar Jenkins en un sistema Linux, incluyendo la configuración inicial y la personalización del servidor para adaptarlo a las necesidades del proyecto.

12.1. Instalación de Jenkins

1. **Actualizar los paquetes del sistema e instalar Java.** Antes de instalar Jenkins, es importante asegurarse de que los paquetes del sistema estén actualizados:

```
1    sudo dnf update && sudo dnf upgrade -y
2    sudo dnf install fontconfig java-17-openjdk wget
```

Cuadro 12.1: Actualizar los paquetes del sistema e instalar Java.

2. **Agregar el repositorio de Jenkins.** Agregar la clave y el repositorio oficial de Jenkins:

```
1    sudo wget -O /etc/dnf.repos.d/jenkins.repo \
2    https://pkg.jenkins.io/redhat-stable/jenkins.repo
3    sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io
         -2023.key
```

Cuadro 12.2: Agregar el repositorio de Jenkins.

3. **Instalar Jenkins y registrar el servicio.** Instalar Jenkins y configurarlo para que se inicie automáticamente al arrancar el sistema (Figura 14):

```
1    sudo dnf install jenkins
2    sudo systemctl daemon-reload
```

Figura 14: *Output* del servicio de Jenkins mostrando los pasos de inicialización.

```
3   sudo systemctl enable jenkins
```

Cuadro 12.3: Instalar Jenkins y registrar el servicio.

4. **Inicializar Jenkins.** Iniciar el servicio de Jenkins y mostrar el estado para encontrar la contraseña de configuración:

```
1   sudo systemctl start jenkins
2   sudo systemctl status jenkins
```

Cuadro 12.4: Inicializar Jenkins.

5. **Acceso a la interfaz *web* de Jenkins.** Acceder a Jenkins desde un navegador en la dirección `http://localhost:8080` para completar la configuración inicial. Al visitar esta página por primera vez, se solicitará la contraseña de configuración mostrada en el paso anterior (ver Figura 15).

Al terminar estos pasos, se deberá contar con una instalación funcional de la última versión LTS (*Long Term Support*) de Jenkins. Siempre se recomienda leer la documentación más reciente en [25].

12.2. Configuración inicial del servidor de Jenkins

1. **Instalar plugins recomendados.** Seleccionar la opción para instalar los plugins recomendados en la página de configuración inicial (véase la Figura 16).

2. **Crear un usuario administrador.** Crear una cuenta de administrador con nombre de usuario, contraseña y correo electrónico (véase la Figura 17).

3. **Configurar la URL del servidor (opcional).** Configurar la URL principal para el servidor Jenkins, lo cual facilita el acceso remoto y la integración con otros servicios.

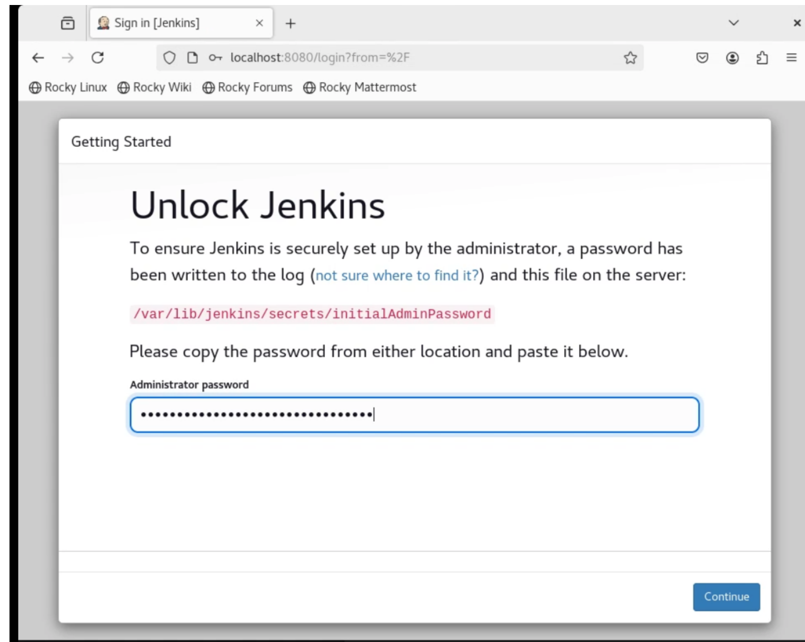

Figura 15: Interfaz *web* de Jenkins pidiendo la contraseña de configuración.

Una instalación funcional y completamente configurada deberá mostrar la página inicial, la cual se usará en la siguiente sección para crear la primera automatización (véase la Figura 18).

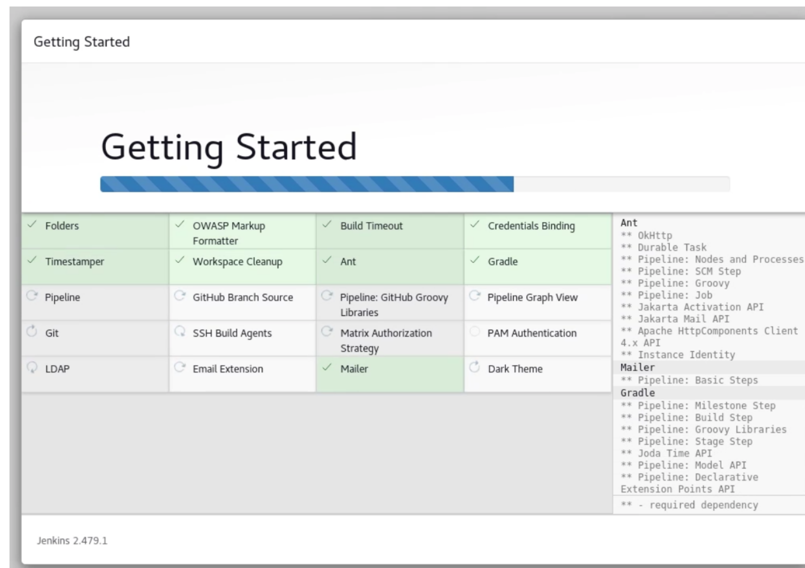

Figura 16: Interfaz *web* de Jenkins instalando los plugins recomendados.

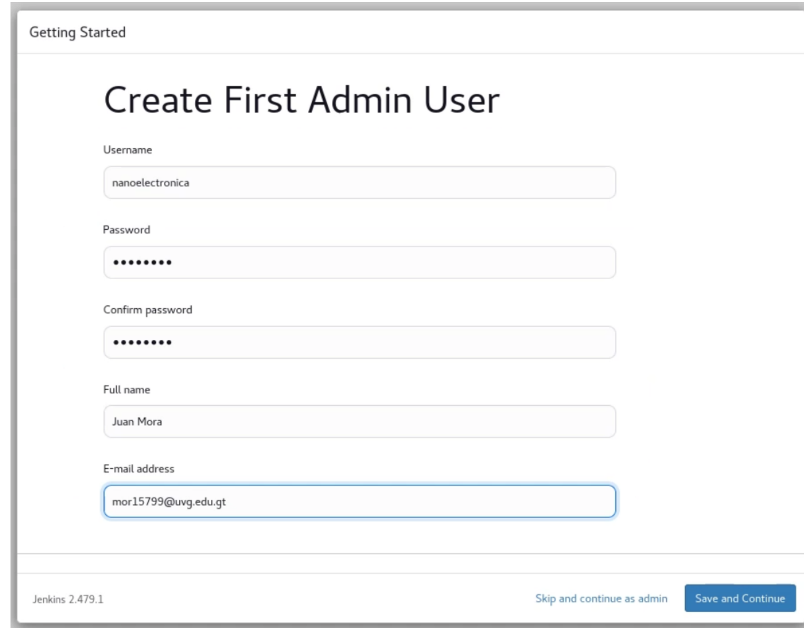

Figura 17: Interfaz de Jenkins creando un usuario administrador nuevo

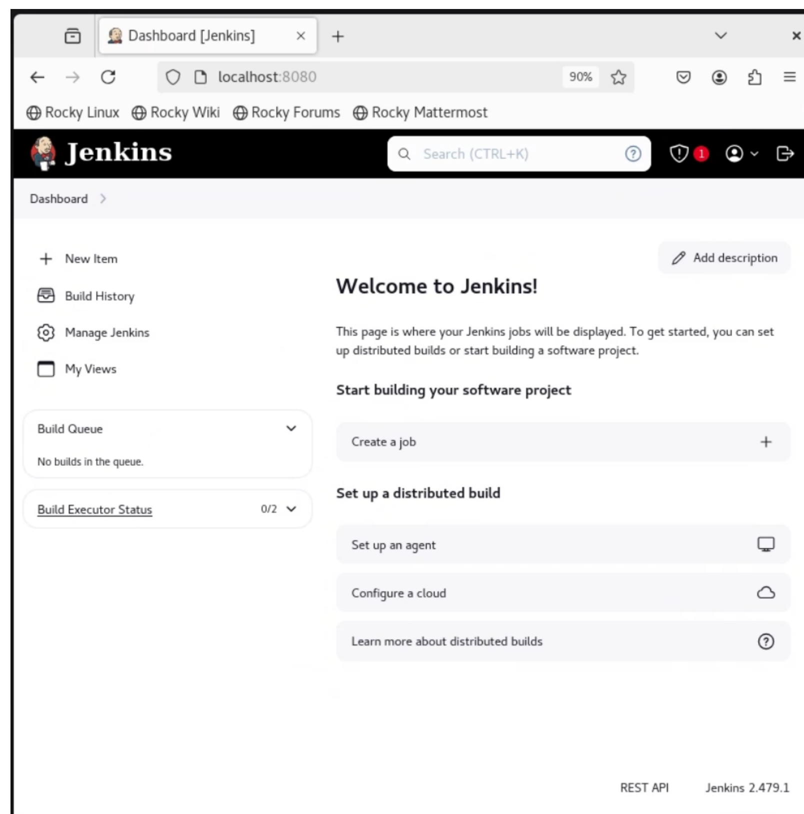

Figura 18: Interfaz de Jenkins mostrando la página inicial

12.2.1. Personalización de instalación y creación de nodos adicionales

Se sugiere fuertemente que el lector explore la interfaz de Jenkins y sus configuraciones, modificando según sus preferencias los distintos parámetros con el objetivo de maximizar el uso de esta excelente plataforma (véase la Figura 19).

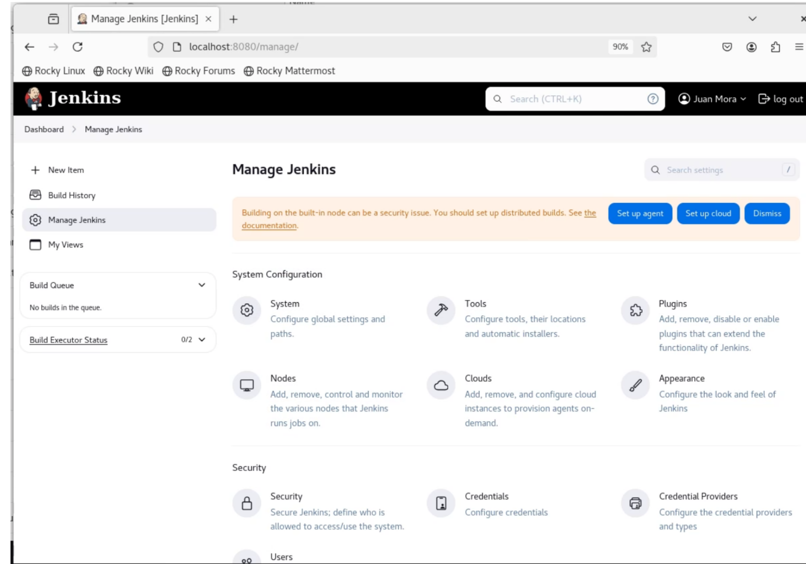

Figura 19: Interfaz de Jenkins mostrando la página de configuraciones

12.3. Creación de un *pipeline* desde la interfaz de Jenkins

1. **Acceder a la sección de creación de un nuevo trabajo.** Desde el panel principal, seleccionar *New Item*.

2. **Seleccionar *Pipeline* como tipo de proyecto.** Introducir un nombre para el *pipeline* y seleccionar *ipeline* como el tipo de proyecto a crear.

3. **Definir el *script* de *pipeline*.** En la sección de configuración, ingresar el *pipeline script* deseado (véase la Figura 20).

4. **Guardar y ejecutar el *pipeline*.** Guardar la configuración del *pipeline* y ejecutar el trabajo para verificar que las tareas se ejecuten correctamente.

El *Pipeline Script* que utilizaremos para ejecutar el proceso de simulación de un Verilog es el siguiente:

```
1  def vcdFile = ""
2
3  pipeline {
4      agent any
5      parameters {
6          string(name: 'REPO_URL', defaultValue: 'https://github.com/JPaulMora
              /sample_verilog_project.git', description: 'Git repository URL')
```

```
7            string(name: 'VERILOG_BASE', defaultValue: 'main', description: 'Base name of the Verilog file (e.g., not, adder)')
8            string(name: 'TECHNOLOGY', defaultValue: 'tsmc/tcb018', description: 'The vendor and libraries to be used for synthesis.')
9        }
10       environment {
11           SNPSLMD_LICENSE_FILE = '27020@192.168.74.1'
12           VCS_HOME = '/usr/synopsys/vcs/U-2023.03/' // Path to VCS installation
13           DC_SHELL_HOME = '/usr/synopsys/syn/U-2022.12-SP2/' // Path to DC Shell installation
14           PATH = "${env.PATH}:${VCS_HOME}/bin:${DC_SHELL_HOME}/bin"
15       }
16       stages {
17           stage('Prepare Workspace') {
18               steps {
19                   echo 'Cleaning workspace...'
20                   deleteDir() // Ensure a clean workspace
21               }
22           }
23           stage('Checkout Code') {
24               steps {
25                   // Pull source files from the repository specified in the REPO_URL parameter
26                   git branch: 'main', url: "${params.REPO_URL}"
27               }
28           }
29           stage('Create Outputs Folder') {
30               steps {
31                   sh 'mkdir -p Outputs' // Create the Outputs folder
32                   echo 'Outputs folder created.'
33               }
34           }
35           stage('Run Initial VCS Simulation') {
36               steps {
37                   script {
38                       // Run the initial VCS simulation
39                       def vcsCmd = "vcs -R -nc -v ${params.VERILOG_BASE}.v ${params.VERILOG_BASE}_tb.v -o output.sim -full64 -debug_access+all"
40                       echo "Executing initial VCS simulation: ${vcsCmd}"
41                       sh vcsCmd
42                   }
43               }
44           }
45           stage('Run Synthesis Step 1') {
46               steps {
47                   script {
48                       echo "Starting Synthesis Step 1..."
49
50                       // Run synthesis with DC Shell
51                       def dcCmd = "dc_shell -f /opt/eda/scripts/synthesis_step_1.tcl -x \"set PARAMETRO \\\"${params.VERILOG_BASE}\\\"; set TECHNOLOGY \\\"${params.TECHNOLOGY}\\\"\""
52                       echo "Executing command: ${dcCmd}"
53                       sh dcCmd
54
55                       // Check if output file is created
```

```
56                     if (!fileExists("./Outputs/${params.VERILOG_BASE}_sintetizado.ddc")) {
57                         error("Synthesis Step 1 failed: Expected output file ${params.VERILOG_BASE}_sintetizado.ddc was not generated.")
58                     }
59                 }
60             }
61         }
62         stage('Combine Synthesized Module with IO File') {
63             steps {
64                 script {
65                     echo "Combining synthesized module with IO file..."
66                     def combinedFile = "Outputs/${params.VERILOG_BASE}_sintetizado_and_IO.v"
67                     def ioFile = "IO_${params.VERILOG_BASE}.v"
68                     def synthesizedFile = "Outputs/${params.VERILOG_BASE}_sintetizado.v"
69 
70                     // Combine files and handle any errors
71                     def catCmd = "cat ${ioFile} ${synthesizedFile} > ${combinedFile}"
72                     if (sh(script: catCmd, returnStatus: true) != 0) {
73                         error("Failed to combine synthesized module with IO file.")
74                     }
75                     echo "Files combined successfully into ${combinedFile}."
76                 }
77             }
78         }
79         stage('Rename VCD File') {
80             steps {
81                 script {
82                     // Use find to locate the VCD file (assumes there's only one .vcd file)
83                     vcdFile = sh(script: "find . -name '*.vcd' -print -quit", returnStdout: true).trim()
84 
85                     // Check if a VCD file was found and rename it
86                     if (vcdFile) {
87                         def targetVcdFile = "sim_1.vcd"
88                         sh "mv ${vcdFile} ${targetVcdFile}"
89                         echo "VCD file ${vcdFile} renamed to ${targetVcdFile}"
90                     } else {
91                         echo "Warning: No VCD file found."
92                     }
93                 }
94             }
95         }
96         stage('Run VCS Simulation with Synthesis Output') {
97             steps {
98                 script {
99                     // Run VCS simulation with the synthesized output from Step 1
100                    def vcsCmd = "vcs -R -nc -v /opt/eda/libs/synopsys/tsmc/tcb018/tcb018gbwp7t.v /opt/eda/libs/synopsys/tsmc/tcb018/tpd018nv.v  Outputs/${params.VERILOG_BASE}_sintetizado.v ${params.VERILOG_BASE}_tb.v -o
```

```
                            output_synth.sim -full64 -debug_access+all"
101                        echo "Executing VCS simulation with synthesized output:
                            ${vcsCmd}"
102                        sh vcsCmd
103                    }
104                }
105            }
106            stage('Rename second VCD File') {
107                steps {
108                    script {
109                        if (vcdFile) {
110                            def targetVcdFile = "sim_2.vcd"
111                            sh "mv ${vcdFile} ${targetVcdFile}"
112                            echo "VCD file ${vcdFile} renamed to ${targetVcdFile
                                }"
113                        } else {
114                            echo "Warning: No VCD file found."
115                        }
116
117                    }
118                }
119            }
120            stage('Run Synthesis Step 2') {
121                steps {
122                    script {
123                        echo "Starting Synthesis Step 2..."
124
125                        // Run synthesis with DC Shell
126                        def dcCmd = "dc_shell -f /opt/eda/scripts/
                            synthesis_step_2.tcl -x \"set PARAMETRO \\\"${params
                            .VERILOG_BASE}\\\"; set TECHNOLOGY \\\"${params.
                            TECHNOLOGY}\\\"\""
127                        echo "Executing command: ${dcCmd}"
128                        sh dcCmd
129
130                        // Check if output file is created
131                        if (!fileExists("./Outputs/${params.VERILOG_BASE}
                            _sintesis_final.ddc")) {
132                            error("Synthesis Step 1 failed: Expected output file
                                 ${params.VERILOG_BASE}_sintesis_final.ddc was
                                not generated.")
133                        }
134                    }
135                }
136            }
137            stage('Run VCS Simulation with final Synthesis Output') {
138                steps {
139                    script {
140                        // Run VCS simulation with the synthesized output from
                            Step 1
141                        def vcsCmd = "vcs -R -nc -v /opt/eda/libs/synopsys/tsmc/
                            tcb018/tcb018gbwp7t.v /opt/eda/libs/synopsys/tsmc/
                            tcb018/tpd018nv.v  Outputs/${params.VERILOG_BASE}
                            _sintesis_final.v ${params.VERILOG_BASE}_tb.v -o
                            output_synth_final.sim -full64 -debug_access+all"
142                        echo "Executing VCS simulation with synthesized output:
                            ${vcsCmd}"
143                        sh vcsCmd
144                    }
```

```
145                 }
146             }
147         stage('Rename third VCD File') {
148             steps {
149                 script {
150                     if (vcdFile) {
151                         def targetVcdFile = "sim_3.vcd"
152                         sh "mv ${vcdFile} ${targetVcdFile}"
153                         echo "VCD file ${vcdFile} renamed to ${targetVcdFile
                                }"
154                     } else {
155                         echo "Warning: No VCD file found."
156                     }
157                     sh "tree ."
158 
159                 }
160             }
161         }
162         stage('Archive Results') {
163             steps {
164                 // Archive all relevant output files
165                 archiveArtifacts artifacts: '*.vcd', allowEmptyArchive: true
166                 archiveArtifacts artifacts: '*.log', allowEmptyArchive: true
167                 archiveArtifacts artifacts: 'Outputs/*sintesis_final*',
                        allowEmptyArchive: true
168                 echo 'Archiving complete.'
169             }
170         }
171     }
172     post {
173         always {
174             echo 'Cleaning up workspace...'
175             deleteDir() // Clean workspace after completion
176         }
177         success {
178             echo 'Pipeline completed successfully!'
179         }
180         failure {
181             echo 'Pipeline failed. Check logs for details.'
182         }
183     }
184 }
```

Cuadro 12.5: Definición del *Pipeline Script* utilizado para hacer síntesis lógica.

12.4. Explicación del *script* de *pipeline*

El siguiente *script* define un *pipeline* en Jenkins para la ejecución de simulaciones utilizando VCS de Synopsys. A continuación, se explican brevemente las secciones principales del *script*.

- `pipeline` **Block:** define el *pipeline* en Jenkins, estableciendo el agente, las variables de entorno y las etapas de trabajo necesarias para completar el proceso de simulación.

- `agent any`: especifica que Jenkins puede ejecutar el *pipeline* en cualquier nodo disponible, proporcionando flexibilidad en la asignación de recursos.
- `environment` **Block:** configura las variables de entorno necesarias para el *pipeline*:
  - `SNPSLMD_LICENSE_FILE`: define el servidor de licencias requerido para ejecutar herramientas de Synopsys, lo que permite que el *software* valide la licencia durante su uso.
  - `VCS_HOME`: define la ruta de instalación de VCS en el servidor Jenkins, asegurando que el *pipeline* pueda acceder al ejecutable de VCS.
  - `PATH`: añade el directorio binario de VCS al `PATH`, facilitando el acceso a los comandos de VCS en cada etapa del *pipeline*.
- **Etapa** `Prepare Workspace`**:** limpia el espacio de trabajo para asegurar que no haya archivos residuales que interfieran con la ejecución del *pipeline*. Utiliza el comando `deleteDir()` para eliminar todos los archivos y carpetas en el *workspace*.
- **Etapa** `Checkout Code`**:** extrae el código fuente desde el repositorio de control de versiones especificado, en este caso de GitHub, y sincroniza el contenido en el espacio de trabajo.
- **Etapa** `Run VCS Simulation`**:** ejecuta el comando de VCS para compilar los archivos de Verilog, generando un binario de simulación.
  - El comando `vcs` utiliza varias opciones, como `-container` para ejecutar en modo contenedor, `-v` para especificar los archivos de entrada, `-o` para definir el nombre del archivo de salida, y `-debug_access+all` para activar el modo de acceso completo de depuración.
- **Etapa** `Run Simulation`**:** ejecuta el binario de simulación generado en la etapa anterior (`./output.sim`) y muestra la salida de la simulación en Jenkins para su revisión.
- **Etapa** `Archive Results`**:** almacena los archivos de salida relevantes, como archivos `.vcd` y `.log`, que contienen los resultados de la simulación y los mensajes de registro. Estos archivos son archivados para su consulta posterior, permitiendo revisar los resultados de la simulación en cualquier momento.
- `post` **Block:** define acciones que Jenkins debe ejecutar una vez que el *pipeline* ha finalizado.
  - `always`: limpia el espacio de trabajo después de cada ejecución para evitar la acumulación de archivos en futuros *builds*.
  - `success`: muestra un mensaje de éxito en la consola cuando el *pipeline* se ejecuta sin errores.
  - `failure`: muestra un mensaje de error en caso de que falle alguna etapa, indicando que los registros deben revisarse para identificar problemas.

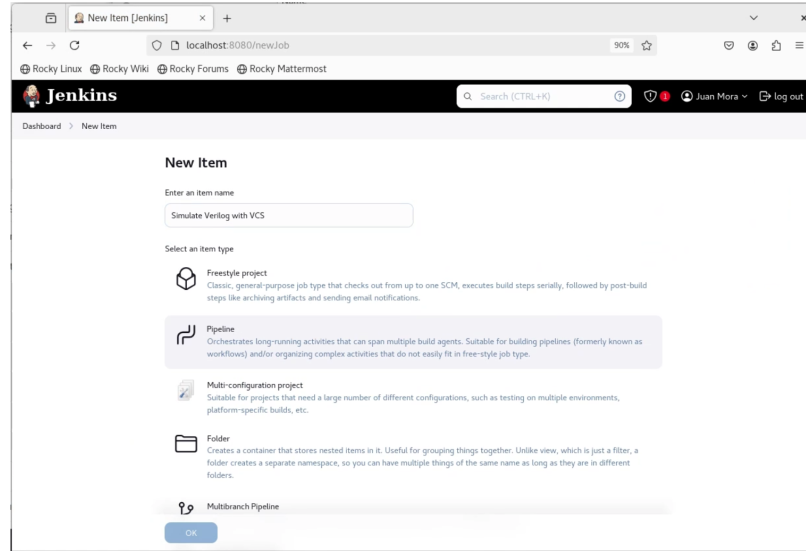

Figura 20: Interfaz de Jenkins creando un usuario administrador nuevo

12.5. *Pipeline* de síntesis física

El siguiente *pipeline* se centra en la síntesis física y complementa el flujo presentado anteriormente, añadiendo nuevos parámetros, configuraciones y herramientas específicas para esta fase del diseño.

```
def gdsFile = ""

pipeline {
    agent {
        node {
            label 'agent'
            customWorkspace '/var/lib/jenkins/workspace/run_physical_synthesis'
        }
    }
    parameters {
        string(name: 'REPO_URL', defaultValue: 'https://github.com/JPaulMora/sample_verilog_project.git', description: 'Git repository URL')
        string(name: 'VERILOG_BASE', defaultValue: 'main', description: 'Base name of the Verilog file')
        string(name: 'TECHNOLOGY', defaultValue: 'tsmc/tcb018', description: 'The technology and libraries used for synthesis')
    }
    environment {
        SNPSLMD_LICENSE_FILE = '27020@192.168.74.1'
        ICC2_HOME = '/usr/synopsys/icc2/W-2024.09/' // Path to ICC2 installation
        ICV_HOME = '/usr/synopsys/icvalidator/U-2022.12-SP1-1/'
        PATH = "${ICC2_HOME}/bin:${ICV_HOME}/bin:${env.PATH}"
    }
    stages {
        stage('Prepare Workspace') {
            steps {
```

```
24                 echo 'Cleaning workspace...'
25                 deleteDir()
26             }
27         }
28         stage('Checkout Code') {
29             steps {
30                 git branch: 'sintesis-fisica', url: "${params.REPO_URL}"
31             }
32         }
33         stage('Create Outputs Folder') {
34             steps {
35                 sh 'mkdir -p Outputs/IO'
36                 echo 'Outputs/IO folder created.'
37             }
38         }
39         stage('Run Physical Synthesis') {
40             steps {
41                 script {
42                     // Set the environment variables for ICC2
43                     env.VERILOG_BASE = params.VERILOG_BASE
44                     env.TECHNOLOGY = params.TECHNOLOGY
45 
46                     // Run ICC2 with the physical synthesis script, using
                           the environment variables directly
47                     def icc2Cmd = "icc2_shell -f /opt/eda/scripts/
                           physical_synthesis_top.tcl"
48                     echo "Executing ICC2 physical synthesis with
                           VERILOG_BASE=${env.VERILOG_BASE} and TECHNOLOGY=${
                           env.TECHNOLOGY}"
49                     sh icc2Cmd
50 
51                 }
52             }
53         }
54         stage('Check for GDS File') {
55             steps {
56                 script {
57                     // Find the generated GDS file
58                     gdsFile = sh(script: "find Outputs/IO -name '*.gds' -
                           print -quit", returnStdout: true).trim()
59                     if (gdsFile) {
60                         echo "GDS file located at ${gdsFile}"
61                     } else {
62                         echo "Warning: No GDS file found."
63                     }
64                 }
65             }
66         }
67         stage('Archive Results') {
68             steps {
69                 // Compress the Outputs folder into a .zip file
70                 sh 'zip -r Outputs.zip Outputs'
71 
72                 // Archive the zipped output file
73                 archiveArtifacts artifacts: 'Outputs.zip', allowEmptyArchive
                       : true
74 
75                 echo 'Archiving complete.'
76             }
```

```
77          }
78      }
79      post {
80          always {
81              echo 'Cleaning up workspace...'
82             // deleteDir()
83          }
84          success {
85              echo 'Physical synthesis pipeline completed successfully!'
86          }
87          failure {
88              echo 'Physical synthesis pipeline failed. Check logs for details
                    .'
89          }
90      }
91  }
```

Cuadro 12.6: Definición del *Pipeline Script* utilizado para hacer síntesis física y comprimir los archivos.

A continuación, se detallan los aspectos más relevantes de este *pipeline*:

12.5.1. Agente con espacio de trabajo personalizado

Durante las pruebas de implementación, se observó que los comandos de *icc2_ shell* no son capaces de utilizar directorios que contengan espacios en el nombre de la ruta. Por convención, Jenkins nombra sus directorios de trabajo utilizando el mismo nombre del *pipeline* que representan. La solución encontrada fue el uso de un directorio o *workspace* específico que no contiene espacios.

Este *pipeline* especifica un agente particular mediante el campo `label: 'agent'` y define un espacio de trabajo personalizado con el parámetro `customWorkspace: '/var/lib/jenkins/workspace/run_physical_synthesis'`. Esto garantiza que los archivos temporales y los resultados generados se almacenen en un directorio dedicado, sin espacios.

12.5.2. Variables de entorno adicionales

Se incorporan nuevas variables de entorno para utilizar herramientas específicas de la síntesis física:

- `ICC2_HOME`: define la ruta de instalación de *ICC2* (*/usr/synopsys/icc2/W-2024.09/*), la herramienta principal para realizar la síntesis física.
- `ICV_HOME`: establece la ubicación de *IC Validator* (*/usr/synopsys/icvalidator/U-2022.12-SP1-1/*), utilizada para la verificación física.

Estas rutas se añaden a la variable `PATH`, ya que *icc2_ shell* invoca el comando *icv* como parte del *script* definido.

12.5.3. Ejecución de síntesis física

En el *stage* `Run Physical Synthesis`, se utiliza la herramienta *ICC2* para realizar la síntesis física mediante la ejecución de un *script* *TCL* llamado `physical_synthesis_top.tcl`. Este *script* emplea las variables de entorno `VERILOG_BASE` y `TECHNOLOGY` para configurar el proceso según los parámetros definidos por el usuario.

12.5.4. Manejo del archivo GDS

El archivo *GDS* (*Graphic Database System*) es el resultado principal de la síntesis física y representa el diseño final en un formato estándar utilizado para la fabricación de circuitos integrados. El *pipeline* incluye un *stage* llamado `Check for GDS File`, donde se busca el archivo GDS en el directorio `Outputs/IO`. Si se encuentra, se almacena en la variable `gdsFile` y se confirma su ubicación.

12.5.5. Compresión y almacenamiento de resultados

Para facilitar la manipulación y almacenamiento de los resultados, el *pipeline* incluye un paso donde la carpeta `Outputs` se comprime en un archivo `Outputs.zip`, el cual se almacena como *artifact* del *pipeline* mediante el comando `archiveArtifacts`. Esto conserva todos los resultados generados y los pone a disposición del usuario de forma más sencilla, permitiendo su descarga directamente desde el navegador.

12.5.6. Rama `sintesis-fisica`

El *pipeline* se asegura de extraer la rama `sintesis-fisica` del repositorio indicado en el parámetro `REPO_URL`. Esta rama probablemente contiene archivos y configuraciones específicas para la fase de síntesis física.

CAPÍTULO 13

Diseño de una estructura de archivos estandarizada para *pipelines*

13.1. Estructura de librerías para diseño de electrónica automatizado (EDA)

La creación de *pipelines* presenta un nuevo reto, ya que, al utilizarse en su máximo potencial, implica que el usuario ya no necesita instalar la *suite* completa de programas de Synopsys, o en el mejor de los casos, el uso de Synopsys se convierte en algo completamente opcional. El reto que esto plantea es la necesidad de definir una estructura estandarizada de directorios para guardar las librerías de distintas tecnologías. Pensando a largo plazo, es lógico considerar que nuevas tecnologías serán puestas a disposición por TSMC, pero también es razonable considerar la posibilidad de incluir otros proveedores.

Asimismo, dado que los usuarios finales no tendrán acceso a las librerías directamente, es esencial que exista este acceso organizado y coherente a los archivos necesarios en cada etapa del proceso de diseño de circuitos, de tal forma que se puedan referenciar en los archivos de configuración.

La nueva estructura de carpetas ha sido organizada de manera que refleje una jerarquía clara basada en el nodo tecnológico y el tipo de librería, lo cual facilita el mantenimiento y la escalabilidad del entorno de trabajo.

En el nivel superior, los directorios se nombran de acuerdo con el proceso específico, como `tcb018`, que representa la tecnología de 0.18µm de TSMC. Este directorio contiene todas las librerías relevantes para dicho nodo, incluyendo subdirectorios para librerías estándar de celdas (`lib`), módulos de descripción de netlists (`ndm`), archivos tecnológicos (`tech`) y *Runsets* (`Runset`).

La organización de las librerías de esta manera proporciona varios beneficios: asegura

claridad al consolidar todos los recursos asociados a un nodo tecnológico en una única ubicación, simplifica el acceso a las librerías necesarias para los procesos de síntesis y simulación, y facilita la gestión modular de librerías provenientes de distintos proveedores o diferentes procesos. Además, esta estructura es escalable, permitiendo una expansión sencilla para incluir múltiples nodos tecnológicos o proveedores de librerías en el futuro, haciéndola adaptable a necesidades de diseño más amplias.

```
1
2   /opt/eda/libs/
3   |-- tsmc/
4   |   `-- tcb018/
5   |       |-- lib/
6   |       |-- ndm/
7   |       |-- Runset/
8   |       |-- tech/
9   |       `-- tlu_and_map/
10  `-- vendor2/
```

Cuadro 13.1: Organización de directorios sugerida para librerías

Una vez estructuradas las librerías, es fácil referenciarlas según la tecnología a utilizar directamente desde el comando o *script* que se emplee.

13.2. Organización de *scripts* para su uso en *pipelines* de Jenkins

Siguiendo la estructura definida con anterioridad, es consistente crear un nuevo folder, `/opt/eda/scripts/`, donde se mantengan los distintos *scripts* necesarios para cada paso del *pipeline*. De esta forma, es posible parametrizar completamente todas las dependencias de un *pipeline* para una tecnología específica.

En este directorio se guardará la lista de comandos requeridos para realizar la síntesis lógica. El *pipeline* funcional que utiliza este estándar de organización se presenta en el Cuadro 12.6.

CAPÍTULO 14

Conclusiones

En este proyecto se abordó el desafío de optimizar el uso de herramientas de Synopsys mediante la creación de un proceso estandarizado de automatización utilizando *pipelines* en Jenkins. A lo largo del desarrollo, se lograron avances significativos en la transición de flujos de trabajo manuales hacia una ejecución completamente automatizada basada en la CLI. Esto representa un paso importante hacia la modernización y escalabilidad de los procesos de diseño de chips electrónicos en la Universidad del Valle de Guatemala.

14.1. Principales logros

- **Automatización eficiente:** se desarrollaron *pipelines* robustos que integran tanto la síntesis lógica como la física, permitiendo ejecutar tareas complejas de manera repetible, rápida y eficiente al punto de simular un Verilog debajo de un minuto, únicamente requiriendo el acceso a la interfaz *web* de Jenkins y un archivo Verilog.
- **Uso de contenedores:** la incorporación de Apptainer como herramienta de contenedores facilitó la creación de entornos portables y reproducibles, asegurando la compatibilidad entre diferentes sistemas y simplificando la gestión de dependencias.
- **Documentación exhaustiva:** se generó una guía completa que documenta no solo los procesos de instalación y configuración, sino también las mejores prácticas para depuración y actualización de los flujos automatizados.
- **Adaptación de herramientas de Synopsys:** la integración de herramientas específicas como VCS, *icc2_shell* y IC *Validator* demostró la capacidad del sistema para manejar procesos complejos de diseño y verificación con resultados predecibles.

14.2. Limitaciones

Es importante notar que el alcance del proyecto estuvo limitado por ciertos factores inherentes al proceso de automatización:

- **Dependencia del conocimiento de las herramientas:** la automatización exitosa depende en gran medida del entendimiento previo del funcionamiento de las herramientas CLI de Synopsys, lo que representó una curva de aprendizaje considerable.
- **Restricciones en *scripts* y en parametrizaciones:** algunas herramientas, como *icc2_shell*, presentaron limitaciones técnicas que requerían configuraciones adicionales, como el uso de un *workspace* sin espacios en los nombres de ruta, así como la inconsistencia en la lectura de variables dentro del *script*, según la sección donde se encuentran.
- **Complejidad en la contenedorización:** la creación de contenedores funcionales para herramientas específicas de Synopsys requirió pruebas exhaustivas para garantizar la compatibilidad, en especial porque Synopsys ofrece una forma no estándar en la industria de utilizar contenedores.

14.3. Reflexión final

El proyecto demuestra que la automatización no solo mejora la eficiencia de los procesos existentes, sino que también abre nuevas oportunidades para explorar y expandir el diseño de nanochips en entornos modernos. La integración de herramientas avanzadas como Jenkins y Synopsys en un flujo de trabajo coherente y documentado establece un estándar que puede ser adoptado y adaptado por futuros proyectos en este campo.

CAPÍTULO 15

Recomendaciones

15.1. Instalación de *software* de Synopsys y uso del *script* de descargas

- **Uso del explorador de archivos de Linux, Nautilus para descargar *software* de Synopsys:** la interfaz gráfica ofrece una forma sencilla y familiar de copiar archivos de la ubicación remota a la PC local.

- **En caso de utilizar un *script* de instalación y descarga:** se recomienda el crear un nuevo *script* utilizando Python, ofrece una plataforma más agnóstica respecto a las versiones de comandos y sistemas operativos, lo que podría mitigar muchos de los problemas asociados con el *script* actual basado en Bash. Además, por ser un lenguaje que abarca el pensum de ingeniería de la universidad, se simplificaría la adaptación del *script* a diferentes entornos, se aprovecharían las capacidades de manejo de errores y modularidad que este lenguaje ofrece, haciendo el *script* más robusto y fácil de mantener.

15.2. Manejo de *artifacts* y resultados de procesos en Jenkins

Para mejorar la gestión y aprovechamiento de los *artifacts* generados en el *pipeline*, se recomiendan las siguientes prácticas:

- **Uso de *artifacts* en procesos posteriores:** los *artifacts* generados pueden ser de gran utilidad como entrada en otros *pipelines* o procesos adicionales. Se recomienda estructurar el flujo de trabajo para que los resultados de una ejecución puedan ser recuperados y reutilizados en etapas futuras o en otros *pipelines*. Esto es especialmente relevante en proyectos donde los datos generados en una simulación o prueba inicial sirven de base para procesos de validación o compilación posteriores.

- **Estrategias de retención de *artifacts*:** para evitar la acumulación excesiva de *artifacts* en el servidor de Jenkins, se recomienda configurar políticas de retención. Jenkins permite definir el tiempo o el número de ejecuciones después del cual los *artifacts* antiguos son eliminados automáticamente. Una práctica común es conservar *artifacts* de las últimas cinco ejecuciones o aquellos generados en el último mes. Estas políticas ayudan a liberar espacio de almacenamiento y mantienen el sistema eficiente, al mismo tiempo que preservan los resultados recientes para análisis o auditorías.

- **Integración de un graficador *web* de archivos VCD:** gracias a las estandarizaciones de procesos de simulación y síntesis lógica, es de esperarse obtener dos a tres archivos de simulación en formato de *waveform* VCD. Es por eso que algún futuro proyecto podría encargarse de adaptar un graficador *web* como https://vc.drom.io/ para comparar fácilmente las simulaciones.

- **Definir los *testbench* de Verilog con macros:** para evitar que los archivos de salida de las simulaciones se sobrescriban, se recomienda parametrizar el nombre del archivo VCD generado en cada simulación utilizando una macro. Esto permite que cada simulación produzca un archivo de salida único sin necesidad de modificar el archivo de *testbench* directamente.

- **Parametrización de *scripts*:** desarrollar un *script* preprocesador utilizando Python y Jinja2 que genere automáticamente un *script* parametrizado para la síntesis física. Este enfoque permite evitar las limitaciones de *icc2_shell*, que no siempre interpreta correctamente las variables definidas en los archivos .tcl, asegurando una ejecución confiable y personalizada.

- **Escalabilidad del sistema:** implementar configuraciones e instalar una mayor cantidad de nodos que permitan ejecutar múltiples *pipelines* en paralelo, optimizando los recursos disponibles.

CAPÍTULO 16

Bibliografía

[1] J. de los Santos, “Diseño de un sumador/restador completo de 32 bits con tecnología CMOS en un proceso de 28 nanómetros usando aplicaciones de diseño de la empresa Synopsys,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2014.

[2] S. H. R. Vásquez, “Definición del Flujo de Diseño para Fabricación de un Chip con Tecnología VLSI CMOS,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2019.

[3] L. A. N. Vásquez, “Implementación de circuitos sintetizados a nivel netlist a partir de un diseño en lenguaje descriptivo de hardware como primer paso en el flujo de diseño de un circuito integrado,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2019.

[4] J. N. R. Orellana, “Definición del flujo en la herramienta VCS para la simulación de HDLs en la Fabricación de un Chip con Tecnología Nanométrica CMOS,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2020.

[5] J. R. G. Rubio, “Etapa de verificación física de Diseño en Silicio vs. Esquemático (LVS) en el flujo de diseño para un chip a nanoescala,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2020.

[6] M. G. F. Espino, “Corrección de anillo de entradas/salidas y pruebas de antenna y ERC para la definición del flujo de diseño del primer chip con tecnología nanométrica desarrollado en Guatemala,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2020.

[7] T. S. M. Company, *Taiwan Semiconductor Manufacturing Company*, Accessed: 2024-05-30, 2024. dirección: `https://www.tsmc.com/english`.

[8] P. A. Mendizábal, “Automatización del proceso de instalación de software de Synopsys e implementación de mejoras utilizando contenedores,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[9] C. Letona, “Diseño de un circuito integrado con tecnología de 180 nm usando librerías de diseño de TSMC: uso avanzado de StarRC para la generación de un archivo HSPICE con componentes parásitos para su correcta simulación,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[10] D. R. E. Pirir, “Diseño e implementación de interfaz gráfica de la automatización de las fases de diseño de un circuito integrado y uso avanzado de IC Compiler II,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[11] L. R. G. Velásquez, “Procesamiento de las señales generadas por un circuito integrado con tecnología de 180nm usando librerías de diseño de TSMC montado en un FPGA Digilent Genesys Board demostrando el funcionamiento mediante una aplicación y sintetizador digital,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[12] S. A. S. Farrington, “Automatización de las verificaciones físicas de un circuito integrado con tecnología de 180 nm utilizando librerías de diseño de TSMC,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[13] A. Aguilar, “Automatización de la etapa de síntesis lógica y creación de archivos *Verilog* para pruebas físicas en un FPGA Genesys Xilinx Virtex-5 LX50T y automatización de la verificación extracción de parásitos,” en *Trabajo de graduación en modalidad de Tesis*, Facultad de Ingeniería, Universidad del Valle de Guatemala, 2022.

[14] Synopsys, *Compute Platforms and Containers*, Accessed: 2024-05-28, 2024. dirección: https://www.synopsys.com/support/licensing-installation-computeplatforms/compute-platforms/containers.html.

[15] Matthew Bio, *A Brief History of the Multi-Core Desktop CPU*, Accessed: 2024-11-09, 2021. dirección: https://www.techspot.com/article/2363-multi-core-cpu/.

[16] Synopsys, *Compute Platforms and Containers*, Accessed: 2024-05-29, 2024. dirección: https://diadem.in/blog/docker-vs-vm-hypervisor-vs-docker/.

[17] Elton Stoleman, *Docker volume mounts*, Accessed: 2024-05-07, 2024. dirección: https://blog.sixeyed.com/docker-volumes-on-windows-the-case-of-the-g-drive/.

[18] O. Foundation, *Dockerfile para crear la imagen de node:18*, Accessed: 2024-06-04, 2024. dirección: https://github.com/nodejs/docker-node/blob/main/18/alpine3.20/Dockerfile.

[19] O. Foundation, *OpenSSH manual page*, Accessed: 2024-11-09, 2024. dirección: https://man.openbsd.org/ssh.

[20] Apptainer, *Apptainer documentation page*, Accessed: 2024-11-10, 2024. dirección: https://apptainer.org/documentation/.

[21] D. INC, *Docker Hub*, Accessed: 2024-11-10, 2024. dirección: https://hub.docker.com/.

[22] S. in the Public Interest, *Debian*, 2024. dirección: https://www.debian.org/.

[23] Canonical, *Ubuntu*, 2024. dirección: https://ubuntu.com/.

[24] A. a Series of LF Projects LLC, *Apptainer documentation*, Accessed: 2024-11-11, 2024. dirección: https://apptainer.org/docs/user/main/fakeroot.html#add-package-to-a-writable-overlay.

[25] Jenkins, *Jenkins Docs*, Accessed: 2024-05-07, 2024. dirección: https://www.jenkins.io/doc/book/installing/linux/.

[26] A. L. D. Team, *Alpine Linux*, 2024. dirección: https://alpinelinux.org/.

CAPÍTULO 17

Glosario

Alpine Linux Distribución de Linux ligera y segura, diseñada para ser pequeña en tamaño y eficiente en recursos [26]. Es ampliamente utilizada como base para contenedores Docker debido a su tamaño reducido y simplicidad, proporcionando solo las herramientas esenciales necesarias para ejecutar aplicaciones. 30

Dockerfile Archivo de texto que contiene instrucciones para construir una imagen de Docker. Especifica el sistema operativo base, las dependencias, configuraciones y comandos necesarios para crear un contenedor funcional. VIII, 16, 17

Runset Un conjunto de configuraciones predefinidas que especifican reglas y parámetros para herramientas de verificación física y síntesis plural. 56

Verilog Un lenguaje de descripción de hardware utilizado para modelar sistemas electrónicos a nivel de comportamiento y estructura. 4, 5, 7, 46, 51, 58, 61, 63

artifact Un archivo o conjunto de archivos generados como resultado de un proceso dentro de un *pipeline*, como un *zip*, `.log` o un archivo *GDS* plural. 12, 13, 55

filesystem Estructura organizada que define cómo se almacenan, gestionan y acceden los datos en un medio de almacenamiento, como un disco duro o SSD. Proporciona una jerarquía de directorios y archivos para facilitar la organización y recuperación de la información. 27

overlays Sistema de archivos en capas que permite combinar múltiples sistemas de archivos, superponiéndolos para que parezcan como uno solo. En el contexto de contenedores, un *overlay filesystem* permite mantener la imagen base en modo de solo lectura mientras se aplican cambios en capas adicionales sin modificar la base. 38

pipeline Es una secuencia de pasos automatizados en un entorno de integración continua, como Jenkins, que permite realizar tareas como compilación, pruebas y despliegue de software plural. VI, X, XI, 56–58, 61

sandbox Entorno de ejecución aislado que se utiliza para probar, ejecutar o desarrollar software sin afectar el sistema operativo anfitrión. En el caso de contenedores, un

sandbox puede ser una configuración donde el contenedor utiliza un directorio del sistema de archivos en lugar de un archivo de imagen. 38, 39

script Un archivo que contiene una serie de comandos ejecutables, utilizados para automatizar tareas específicas, como síntesis o simulación plural. V–VII, X, XI, 1, 4, 22, 28, 31, 34–36, 40, 41, 46, 50, 55, 60

workspace El directorio en el que Jenkins almacena los archivos necesarios para ejecutar un *pipeline*, como código fuente, resultados de compilación y *artifacts*. 51, 54, 59

EDA Acrónimo de *Electronic Design Automation*, se refiere a las herramientas y procesos utilizados para diseñar y verificar circuitos electrónicos. VI, X, XI, 56

GDS Acrónimo de *Graphic Database System*, es un formato estándar para representar diseños de circuitos integrados utilizados en la fabricación. VI, 55

Synopsys Un proveedor líder de herramientas de diseño automatizado de electrónica (*EDA*) utilizadas para la creación, síntesis y simulación de circuitos integrados. VI–VIII, X, XI, 1–8, 26, 32–37, 39, 40, 56, 58–60, 62