A

Agente de seguridad en Java

A.1. Introducción

La ciberseguridad es un aspecto fundamental en el desarrollo de software moderno. A medida que las aplicaciones manejan datos sensibles y operan en entornos complejos, la necesidad de protegerlas contra amenazas como inyecciones de código, accesos no autorizados y el uso de criptografía débil es crucial. Este proyecto aborda estos desafíos mediante la creación de un agente de seguridad en Java, que se integra en aplicaciones existentes para detectar y prevenir vulnerabilidades comunes en tiempo de ejecución.

El proyecto se ha desarrollado utilizando tecnologías como ByteBuddy y OWASP ESAPI. ByteBuddy facilita la instrumentación del código, permitiendo interceptar métodos y modificar su comportamiento sin necesidad de modificar el código fuente original. OWASP ESAPI proporciona herramientas avanzadas para la validación de entradas, ayudando a identificar patrones peligrosos en los datos que se procesan.

El objetivo principal de este proyecto es proveer una herramienta robusta y fácilmente integrable para mejorar la seguridad de las aplicaciones Java, monitorizando y mitigando amenazas sin afectar la funcionalidad de la aplicación.

A.2. Estructura del Proyecto

El proyecto está organizado en diferentes módulos, cada uno diseñado para cumplir una función específica en el proceso de seguridad:

- **Agente de Seguridad (TFGSecurityAgent)**: este módulo es el corazón del proyecto. Se encarga de instrumentar el código y aplicar las transformaciones necesarias para la monitorización de seguridad.

- **Asesoramiento de Seguridad (TFGSecurityAdvice)**: contiene las reglas y mecanismos para identificar y manejar posibles amenazas. Este componente es el que realiza la validación y análisis de las entradas.

- **Anotación Personalizada (SecureClass)**: permite seleccionar qué clases deben ser monitorizadas, evitando así una instrumentación excesiva.

- **Instrumentación Mock (MockInstrumentation)**: proporciona un entorno de

pruebas para simular la instrumentación sin necesidad de ejecutar el agente en un entorno real.

- **Pruebas Unitarias (TFGSecurityAgentTest)**: verifica que las funcionalidades del agente funcionen correctamente, asegurando que el proyecto cumpla con sus objetivos de seguridad.

- **Archivo de Configuración de Maven (pom.xml)**: define las dependencias y plugins necesarios para construir, probar y empaquetar el proyecto.

A.3. Descripción Detallada de los Componentes

A.3.1. Agente de Seguridad (TFGSecurityAgent)

El archivo TFGSecurityAgent.java define la lógica para instrumentar las clases de una aplicación. Utilizando ByteBuddy, el agente intercepta los métodos de las clases que han sido anotadas con @SecureClass. Cuando se intercepta un método, se aplica el TFGSecurityAdvice, que analiza las entradas del método y busca patrones de comportamiento peligrosos.

```
1  public static void premain(String agentArgs, Instrumentation inst) {
2      new AgentBuilder.Default()
3          .type(ElementMatchers.isAnnotatedWith(SecureClass.class))
4          .transform(new AgentBuilder.Transformer() {
5              @Override
6              public DynamicType.Builder<?> transform(
7                  DynamicType.Builder<?> builder,
8                  TypeDescription typeDescription,
9                  ClassLoader classLoader,
10                 JavaModule module,
11                 ProtectionDomain protectionDomain) {
12                 return builder.visit(Advice.to(TFGSecurityAdvice.class)
13                     .on(ElementMatchers.any()));
14             }
15         })
16         .with(AgentBuilder.Listener.StreamWriting.toSystemOut())
17         .installOn(inst);
18 }
```

Descripción del Código:

- `premain`: Método principal del agente, llamado antes de que se ejecute la aplicación.
- `transform`: Define la transformación que se aplicará a las clases anotadas. Esta transformación aplica el `TFGSecurityAdvice` para monitorear la seguridad.

A.3.2. Asesoramiento de Seguridad (TFGSecurityAdvice)

El archivo TFGSecurityAdvice.java contiene la lógica para analizar y detectar posibles vulnerabilidades. Este componente se encarga de examinar las entradas de métodos interceptados y aplicar reglas de validación para detectar:

- Inyecciones SQL
- Datos sensibles expuestos
- Través de directorios
- Criptografía débil
- Inyecciones de código o comandos

```
1  public static void detectSQLInjection(String method, String input) {
2     try {
3         if (Pattern.compile("(?i).*\\b(SELECT | INSERT | DELETE | UPDATE |
   DROP | ALTER)\\b.*").matcher(input).find()) {
4             throw new ValidationException("Posible Inyección SQL detectada",
   "El input contiene posibles comandos SQL.");
5         }
6         ESAPI.validator().getValidInput("SQL input", input, "SQL", 200,
   false);
7     } catch (ValidationException | IntrusionException e) {
8         logError("[ALERTA] Posible inyección SQL detectada en: " + method +
   " con valor: " + input);
9     }
10 }
```

Descripción del Código:

- detectSQLInjection: Este método utiliza patrones regulares para identificar posibles inyecciones SQL en las entradas de texto. También se apoya en ESAPI para validar las entradas, asegurando que los datos sean seguros.

A.3.3. Anotación Personalizada (SecureClass)

SecureClass es una anotación personalizada que se utiliza para marcar las clases que deben ser monitoreadas por el agente de seguridad.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.TYPE)
3  public @interface SecureClass {
4  }
```

Descripción del Código:

- @Retention(RetentionPolicy.RUNTIME): Indica que la anotación estará disponible en tiempo de ejecución.
- @Target(ElementType.TYPE): Especifica que la anotación solo se puede aplicar a clases o interfaces.

A.3.4. Instrumentación Mock (MockInstrumentation)

MockInstrumentation proporciona un entorno de pruebas para simular el comportamiento de la instrumentación real sin necesidad de ejecutar el agente en un entorno de producción.

```
1  public static Instrumentation getInstrumentation() {
2      return new MockInstrumentation();
3  }
```

Descripción del Código:

- getInstrumentation: Devuelve una instancia simulada de Instrumentación para pruebas unitarias.

A.3.5. Pruebas Unitarias (TFGSecurityAgentTest)

El archivo TFGSecurityAgentTest.java define las pruebas para verificar que el agente de seguridad funcione correctamente y pueda detectar diferentes tipos de amenazas. Se utiliza JUnit para gestionar las pruebas.

```
1  @Test
2  public void testSQLInjectionDetection() {
3      String[] testCases = {
4          "1 OR 1=1",
5          "' OR '1'='1",
6          "SELECT * FROM users WHERE username = 'admin' --",
7          "safeString"
8      };
9      for (String input : testCases) {
10         if (!input.equals("safeString")) {
11             assertDoesNotThrow(() -> {
12                 TFGSecurityAdvice.detectSQLInjection("testMethod", input);
13             });
14         }
15     }
16 }
```

Descripción del Código:

- testSQLInjectionDetection: Verifica que el método detectSQLInjection pueda identificar correctamente entradas peligrosas sin lanzar excepciones inesperadas.

A.3.6. Archivo de Configuración de Maven (pom.xml)

El archivo pom.xml define las dependencias y plugins necesarios para construir, probar y empaquetar el proyecto.

```
<dependencies>
    <!-- ByteBuddy core -->
    <dependency>
        <groupId>net.bytebuddy</groupId>
        <artifactId>byte-buddy</artifactId>
        <version>1.12.18</version>
    </dependency>
    <!-- JUnit para pruebas unitarias -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Descripción del Código:

- Dependencias como ByteBuddy permiten la instrumentación del código y JUnit se utiliza para gestionar las pruebas unitarias.

B

Agente de monitoreo y rendimiento para aplicaciones Java

B.1. Introducción

El propósito de este proyecto es desarrollar un agente de monitoreo para aplicaciones Java que permita la evaluación de rendimiento, detección de problemas de seguridad y control de recursos en tiempo real. El proyecto está diseñado para ofrecer una herramienta que facilite la detección y resolución de problemas de rendimiento en aplicaciones Java, empleando técnicas de instrumentación dinámica mediante ByteBuddy y otras tecnologías relacionadas.

B.2. Objetivos del Proyecto

Los objetivos específicos de este proyecto son:

- **Instrumentación Dinámica**: implementar un agente de Java que pueda insertar código de monitoreo en tiempo de ejecución.
- **Medición del Rendimiento**: permitir la medición de métricas clave como el tiempo de ejecución de métodos, el uso de recursos (CPU, memoria) y la gestión de hilos.
- **Control de Seguridad**: mMonitorizar la seguridad mediante la detección de prácticas inseguras (inyección de código, criptografía débil, etc.).
- **Desarrollo de una Arquitectura Modular**: crear un sistema escalable y modular que permita añadir funcionalidades adicionales en el futuro.

B.3. Fundamentos Teóricos

La instrumentación de aplicaciones Java permite modificar o ampliar el comportamiento de un programa durante la ejecución, sin necesidad de alterar el código fuente original. Para lograr esto, se emplea la herramienta ByteBuddy, que facilita la creación de agentes Java capaces de

interceptar métodos y agregar código antes o después de su ejecución.

B.4. Descripción Técnica del Proyecto

B.4.1. Arquitectura General

El proyecto se divide en módulos que cubren diferentes aspectos del monitoreo:

- **Medición de Tiempo**: permite medir el tiempo que tardan los métodos anotados con @MedirTiempo.
- **Monitoreo de Recursos**: evalúa el uso de CPU y memoria de los métodos que se instrumentan usando @MonitorearRecursos.
- **Control de Hilos**: supervisa el comportamiento de los hilos en métodos anotados con @MonitorearHilos.

B.4.2. Anotaciones Personalizadas

Se utilizan anotaciones para definir qué métodos deben ser monitoreados, por ejemplo:

- **@MedirTiempo**: marca métodos para medir su tiempo de ejecución.
- **@MonitorearRecursos**: indica que se debe evaluar el uso de recursos.
- **@MonitorearHilos**: permite la supervisión de la gestión de hilos.

B.4.3. Agente de Instrumentación: PerformanceAgent

El agente principal (PerformanceAgent) se inicializa al cargar la aplicación y añade los transformadores necesarios para instrumentar las clases anotadas. Utiliza ByteBuddy para realizar estas operaciones en tiempo de ejecución.

```
1  public class PerformanceAgent {
2     public static void premain(String agentArgs, Instrumentation inst) {
3         new AgentBuilder.Default()
4             .type(ElementMatchers.isAnnotatedWith(MedirTiempo.class)
5               .or(ElementMatchers.isAnnotatedWith(MonitorearRecursos.class))
6                 .or(ElementMatchers.isAnnotatedWith(MonitorearHilos.class)))
7             .transform(new AgentBuilder.Transformer() {
8                 @Override
9                 public DynamicType.Builder<?> trans-
   form(DynamicType.Builder<?> builder, TypeDescription typeDescription,
   ClassLoader classLoader, JavaModule module, ProtectionDomain protectionDo-
   main) {
```

```
10          return builder.visit( Advice.to( MonitoringAdvice.class)
    .on(ElementMatchers.any()));
11        }
12       }).installOn(inst);
13     }
14 }
```

B.4.4. Simulación de Instrumentación: MockInstrumentation

Para facilitar las pruebas, se implementó una clase MockInstrumentation que simula la funcionalidad de la clase Instrumentation real, permitiendo probar la lógica sin necesidad de ejecutar el agente en un entorno de producción.

B.4.5. Tests Automatizados

Se han desarrollado pruebas unitarias usando JUnit y Mockito para verificar el comportamiento correcto del agente y las funcionalidades monitoreadas. Estas pruebas incluyen la detección de problemas de rendimiento, como el uso excesivo de CPU o tiempos de ejecución prolongados, y también problemas de seguridad.

B.5. Resultados Obtenidos

Tras la implementación, el agente fue capaz de monitorear y detectar:

- Problemas de Rendimiento: detección de métodos que exceden ciertos tiempos límite.
- Gestión de Recursos: identificación de uso elevado de CPU y memoria.
- Problemas de Seguridad: alertas sobre el uso de criptografía débil o consultas SQL sospechosas.
- Gestión de Hilos: detección de bloqueos y comportamientos anómalos en la gestión de hilos.

C

Agente de Filtrado de IP

C.1. Descripción General

IPFilterAgent es un agente desarrollado para el filtrado de direcciones IP, que permite controlar el acceso a ciertos métodos o recursos según la pertenencia de una IP a listas de control (whitelist o blacklist). Este sistema permite aplicar restricciones de seguridad mediante anotaciones en métodos, asegurando que solo las direcciones IP autorizadas puedan acceder a funciones sensibles o restringidas en el sistema.

Este agente es ideal para implementar medidas de seguridad en aplicaciones críticas, evitando accesos no deseados o potencialmente peligrosos basados en la IP de origen.

C.2. Componentes Principales

AgentConfig

Esta clase gestiona la configuración del agente y permite el manejo de dos listas de control:

- Whitelist: lista de IPs autorizadas explícitamente para acceder.
- Blacklist: lista de IPs a las que se deniega el acceso.

Principales métodos:

- addToWhitelist(String ip): añade una IP a la whitelist.
- addToBlacklist(String ip): añade una IP a la blacklist.
- removeFromWhitelist(String ip): elimina una IP de la whitelist.
- removeFromBlacklist(String ip): elimina una IP de la blacklist.
- saveConfig(): Guarda la configuración en un archivo llamado ip_filter.config, donde se almacenan ambas listas.

Manejo del archivo de configuración (ip_filter.config): El archivo de configuración se genera y actualiza dinámicamente. Cuando no existen IPs en una lista, se omite la línea completa correspondiente.

IPFilterAdvice

Este componente actúa como la lógica de filtrado principal. Mediante el uso de anotaciones, identifica si un método específico requiere restricción de acceso.

Principales funciones:

- @WhitelistIP y @BlacklistIP: anotaciones utilizadas para definir si un método requiere un filtro de whitelist o blacklist.

- checkIP(Method method): método que valida la IP de origen según las listas de control. Genera una excepción SecurityException si una IP en blacklist intenta acceder o si una IP fuera de la whitelist intenta acceder a un método restringido.

Simulación de IP: a través de setIpProvider(Supplier<String> ipProvider), se permite definir una IP simulada, útil para pruebas. En producción, este método tomaría la IP de origen real.

Anotaciones de Control (@WhitelistIP, @BlacklistIP)

Estas anotaciones se aplican en los métodos del sistema para indicar si el acceso debe ser controlado por whitelist o blacklist. Cada método anotado con @WhitelistIP será accesible solo para las IPs en la whitelist, mientras que @BlacklistIP bloqueará el acceso para IPs listadas en la blacklist.

C.3. Ejemplo de Uso

A continuación se muestra un ejemplo de cómo utilizar las anotaciones y el agente de filtrado:

```
@WhitelistIP
public void metodoProtegidoPorWhitelist() {
    // Lógica restringida a IPs en la whitelist
}

@BlacklistIP
public void metodoProtegidoPorBlacklist() {
    // Lógica restringida, no accesible para IPs en la blacklist
}
```

C.4. Casos de Prueba

La clase IPFilterAgentTest incluye un conjunto exhaustivo de pruebas que validan los diferentes escenarios de acceso:

- Permitir acceso a IP en whitelist.
- Denegar acceso a IP fuera de whitelist.
- Denegar acceso a IP en blacklist.
- Permitir acceso a IP fuera de blacklist.

Cada prueba utiliza una IP simulada para verificar el comportamiento de IPFilterAgent.

C.5. Modo de Ejecución

Inicialización: el archivo de configuración ip_filter.config se genera automáticamente si no existe. Este archivo se actualiza cada vez que se modifica una de las listas.

Filtrado: durante la ejecución, IPFilterAdvice verifica la IP del solicitante contra las listas de control antes de permitir el acceso a un método anotado.

C.6. Errores Comunes y Soluciones

Excepciones en el archivo de configuración: si el archivo ip_filter.config no se genera correctamente, verificar permisos de escritura o posibles errores de ruta.

NullPointerException en pruebas: asegurarse de que AgentConfig esté inicializado y que setUp() en IPFilterAgentTest esté configurado correctamente.

D

Agente de Validación de Orden de Métodos

D.1. Introducción

El MethodOrderAgent es un agente Java diseñado para interceptar y validar el orden de ejecución de métodos en clases anotadas con @ExecutionOrder. Este agente opera a nivel de bytecode utilizando Byte Buddy, permitiendo imponer reglas de ejecución basadas en expresiones regulares sin modificar directamente el código fuente.

D.2. Arquitectura General

El agente está compuesto por varios componentes clave que trabajan juntos para instrumentar clases y validar el orden de los métodos en tiempo de ejecución:

- **MethodOrderAgent**: configura el agente mediante el uso de Byte Buddy para interceptar constructores y métodos en clases anotadas con @ExecutionOrder.
- **MethodOrderAdvice**: proporciona la lógica de validación para los métodos interceptados.
- **ConstructorAdvice**: gestiona la inicialización y el registro de las instancias en ExecutionOrderRegistry al finalizar la ejecución del constructor.
- **ExecutionOrder**: anotación que define la regla de orden de ejecución de métodos mediante expresiones regulares.
- **ExecutionOrderRegistry**: registro global que asocia instancias con máquinas de estados (StateMachine) que controlan las transiciones de métodos.

D.3. Componentes Principales

D.3.1. MethodOrderAgent

El componente principal del agente, configurado para interceptar clases anotadas con @ExecutionOrder. Define las reglas para instrumentar métodos y constructores, y aplicar el Advice correspondiente.

Características clave:

- Detecta automáticamente las clases anotadas con @ExecutionOrder.
- Aplica el ConstructorAdvice para interceptar la creación de instancias.
- Aplica el MethodOrderAdvice para interceptar los métodos.

```
1  new AgentBuilder.Default()
2        .type(ElementMatchers.declaresAnnotation (ElementMatch-
   ers.annotationType(ExecutionOrder.class)))
3        .transform((builder, typeDescription, classLoader, module,     protec-
   tionDomain) -> builder
4           .constructor(ElementMatchers.any()) // Interceptar constructores
5           .intercept(net.bytebuddy.asm.Advice.to (MethodOrderAdvice.class))
6           .method(ElementMatchers.any()) // Interceptar todos los métodos
7           .intercept(net.bytebuddy.asm.Advice.to (MethodOrderAdvice.class)))
8        .installOn(inst);
```

D.3.2. MethodOrderAdvice

Valida el orden de ejecución de los métodos interceptados, invocando la máquina de estados asociada a la instancia. Si el orden es incorrecto, lanza una excepción.

Funciones principales:

- Valida la transición al método actual.
- Verifica si la instancia está en un estado final válido después de la ejecución del método.
- Propaga excepciones si el orden es inválido.

Puntos clave:

- Se ejecuta al inicio y al final de cada método interceptado.
- Informa el estado actual de la máquina al desarrollador mediante logs.

```
try {
   stateMachine.validateTransition(methodName);
} catch (IllegalStateException e) {
   System.out.println("[ERROR] Invalid method order: " + methodName + " for instance: " + instance);
   throw e; // Propagate the exception to enforce invalid state handling
}
```

D.3.3. ConstructorAdvice

Intercepta el final de la ejecución del constructor, registra la instancia y la asocia con una máquina de estados basada en las reglas definidas en @ExecutionOrder.

Características clave:

- Extrae el valor de la anotación @ExecutionOrder.
- Registra la instancia en ExecutionOrderRegistry.
- Valida que la máquina de estados se haya registrado correctamente.

```
// Registrar en el ExecutionOrderRegistry
String executionOrderValue = executionOrder.value();
System.out.println("[INFO] @ExecutionOrder value: " + executionOrderValue);
ExecutionOrderRegistry.register(instance, executionOrderValue);
```

D.3.4. ExecutionOrder

Anotación que define la regla de orden de ejecución de métodos como una expresión regular. Esta expresión regular se utiliza para configurar la máquina de estados.

```
@ExecutionOrder("(start process end)")
public class MyClass {
   public void start() { }
   public void process() { }
   public void end() { }
}
```

D.3.5. ExecutionOrderRegistry

Registro global que asocia instancias de clases con sus respectivas máquinas de estados (StateMachine). Actúa como un punto central para recuperar la máquina de estados

correspondiente a una instancia interceptada.

Funciones principales:

- Registrar instancias con reglas específicas.
- Recuperar la máquina de estados asociada a una instancia.

Puntos clave:

- Se asegura que cada instancia tenga un hash único e inmutable.
- Permite realizar validaciones centralizadas.

D.4. Flujo de Ejecución del Agente

Instrumentación:

- Al cargar una clase, Byte Buddy intercepta las clases anotadas con @ExecutionOrder.
- Instrumenta los constructores con ConstructorAdvice y los métodos con MethodOrderAdvice.

Inicialización:

- Al instanciar un objeto, se ejecuta ConstructorAdvice.
- Se registra la instancia y se configura su máquina de estados en ExecutionOrderRegistry.

Validación en Tiempo de Ejecución:

- Cada vez que se invoca un método, MethodOrderAdvice valida la transición a través de la máquina de estados.
- Si la transición es inválida, se lanza una excepción IllegalStateException.

D.5. Documentación de la Sintaxis para la Configuración del Agente

El agente permite configurar el orden esperado de ejecución de los métodos de una clase mediante expresiones regulares simplificadas. Estas definiciones se registran en el sistema y son validadas en tiempo de ejecución.

Las reglas básicas son:

1. **Transiciones Lineales**
   - Representan una secuencia estricta de métodos.
   - Sintaxis: (method1 method2 method3)
   - Ejemplo: (start process end)
     - Transiciones válidas: start -> process -> end.

2. **Operador AND (&)**
   - Ambos métodos deben ejecutarse, pero el orden no importa.
   - Sintaxis: (method1 & method2)
   - Ejemplo: (start & process) -> end
     - Transiciones válidas:
       - start -> process -> end
       - process -> start -> end.

3. **Operador OR (|)**
   - Solo uno de los métodos debe ejecutarse.
   - Sintaxis: (method1 | method2)
   - Ejemplo: (start | process) -> end
     - Transiciones válidas:
       - start -> end
       - process -> end.

4. **Repetición ({n})**
   - Un método debe ejecutarse exactamente n veces.
   - Sintaxis: (method){n}
   - Ejemplo: (repeat){2} -> end
     - Transiciones válidas:
       - repeat -> repeat -> end.

5. **Comodín (.*)**
   - Representa cualquier número de métodos intermedios, conocidos o desconocidos.
   - Sintaxis: start .* end
   - Ejemplo: (start .* end)
     - Transiciones válidas:
       - start -> intermediate1 -> intermediate2 -> end.

6. **Estados Finales Declarados**
   - Permite definir un estado final explícito y configurar transiciones especiales desde este estado.
   - Sintaxis: end [end:method1,method2]
   - Ejemplo: start -> end [end:fun1,fun2]
     - Transiciones válidas:
       - start -> end -> fun1
       - start -> end -> fun2.

7. **Combinaciones Complejas**
   - Permite combinar operadores lógicos, comodines y repeticiones.
   - Sintaxis: ((method1 & method2) | method3) -> end
   - Ejemplo: ((start & process) | middle) -> end -> FINAL
     - Transiciones válidas:
       - start -> process -> end
       - middle -> end.

D.6. Documentación de Capacidades y Límites

A continuación, se detallan las capacidades y limitaciones de la máquina de estados según las pruebas realizadas. Los casos se organizan en funcionalidades y ejemplos.

D.6.1. Consideraciones Importantes

Exclusividad del operador OR (|)

El operador OR es exclusivo. Esto significa que, en un estado (a | b), solo puede ejecutarse uno de los caminos posibles, nunca ambos.

Flexibilidad del operador AND (&)

En el operador AND, el orden de ejecución de los estados no importa. Es decir, en (a & b), puede ejecutarse primero a y luego b, o al revés.

Hash de la instancia

Para garantizar el correcto funcionamiento de la máquina de estados, el hash de la clase debe ser inmutable durante la ejecución. Si el hash cambia, la máquina de estados puede fallar al asociar la clase.

Métodos de clase que no intercepta

Para evitar fallos no deseados y errores el agente obvia y no intercepta los métodos tostring, equals, hashcode y clone.

D.6.2. Ejemplos

Transiciones Lineales: Válido

Ejemplo válido: (start process end)

Ejemplo no válido: (No aplica)

Comentarios: las transiciones secuenciales funcionan sin problemas, incluso con múltiples pasos.

Operador AND (&): Válido (Simple)

Ejemplo válido: (start & process) -> end

Ejemplo no válido: ((a & b) | (c & d)) -> end

Comentarios: el operador AND funciona bien en configuraciones simples. Sin embargo, puede fallar en combinaciones complejas con OR u otros operadores.

Operador OR (|): Válido

Ejemplo válido: (start | process) -> end

Ejemplo no válido: (start | process & extra) -> end

Comentarios: el operador OR es funcional en escenarios básicos y combinaciones simples, pero puede fallar en interacciones con AND.

Combinaciones OR y AND (AND-OR / OR-AND): Válido

Ejemplo válido: ((start & process) | middle) -> end -> FINAL

Ejemplo no válido: (start | process) & extra -> end

Comentarios: el sistema soporta combinaciones básicas de AND y OR, como (start & process) | middle o (start | process) & end.

Condiciones Comodín (.*): Válido

Ejemplo válido: (start .* end)

Ejemplo no válido: (start .* middle .* end)

Comentarios: permite múltiples transiciones dinámicas, pero falla con múltiples comodines o anidaciones más complejas.

Repeticiones ({n}): Válido (Simple)

Ejemplo válido: (repeat){2} -> end

Ejemplo no válido: (a .* b){2} -> end

Comentarios: las repeticiones funcionan bien en configuraciones simples, pero fallan si se combinan con comodines o estructuras más avanzadas.

Transiciones Especiales desde FINAL: Válido

Ejemplo válido: start -> end [end:fun1,fun2]

Ejemplo no válido: start -> end [end:invalid]

Comentarios: soporta transiciones especiales configuradas explícitamente desde estados finales. No permite valores fuera de las configuraciones.

Caminos Redundantes: Válido

Ejemplo válido: start -> (a | b) -> end

Ejemplo no válido: No aplica. Los caminos redundantes están soportados.

Comentarios: el sistema soporta caminos alternativos hacia un mismo estado final, como (a | b) hacia end.

Validación de Sintaxis de Regex: No Válido

Ejemplo válido: No aplica.

Ejemplo no válido: (start -> -> end)

Comentarios: la validación de regex no es robusta y puede generar comportamientos inesperados si el formato no es correcto.

Combinaciones Complejas (AND, OR, {n}): Válido (Con Precisión)

Ejemplo válido: ((a & b) | (c & d)) -> (x .* y) -> z -> FINAL

Ejemplo no válido: ((a & b) | (c & d)) -> (x .* y .* z) -> FINAL

Comentarios: el sistema soporta combinaciones complejas si se siguen caminos lógicos bien definidos.