Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

MimicryMonitors:

Verificación de programas con fragmentos comunes.

Tesis de Licenciatura en Ciencias de la Computación

Felicitas García

Director: Javier Godoy
Codirector: Diego Garbervetsky
Buenos Aires, 2025

RESUMEN

En el desarrollo de software moderno, es común que los sistemas evolucionen mediante pequeños cambios incrementales. Esta característica se puede explotar en diversos escenarios, especialmente en el testing de regresión, donde surge una pregunta fundamental: cuando se modifica una parte de un programa previamente verificado, ¿es necesario re-evaluar todo el sistema o se puede aprovechar el trabajo de verificación ya realizado? En particular, ¿se puede utilizar la noción de que dos versiones comparten gran parte de su código para reducir el esfuerzo de testeo?

En este contexto, esta tesis explora e implementa los Mimicry Monitors (MM), una técnica que justamente capitaliza los fragmentos comunes entre dos versiones de un programa. La técnica permite verificar en tiempo de ejecución si el comportamiento de un programa bajo análisis (PUA) puede ser imitado por un programa de referencia u oráculo (OP), sin necesidad de ejecutar este último.

Para validar la técnica, se realizó una evaluación experimental sobre cinco herramientas de GNU Core Utilities, seleccionando dos versiones de cada una y utilizando sus conjuntos de pruebas existentes para simular escenarios reales de testing de regresión. Se define la efectividad de los Mimicry Monitors como su capacidad de emitir veredictos anticipados sobre la existencia de una contraparte del OP para la ejecución actual del PUA. Los resultados revelan entonces que la efectividad de los MMs está intrínsecamente ligada a la naturaleza de las modificaciones entre versiones. Para programas con modificaciones localizadas como cat, se obtuvo hasta un 60.87 % de casos verificados, permitiendo terminación temprana. En contraste, programas con modificaciones estructurales fundamentales como ls no mostraron resultados igualmente favorables.

Los resultados demuestran que los Mimicry Monitors constituyen una herramienta valiosa para la optimización de testing de regresión, validando parcialmente la hipótesis de que es posible evitar ejecuciones redundantes mediante el análisis de fragmentos comunes.

Palabras claves: Verificación, fragmentos comunes, testing de regresión, monitoreo en tiempo de ejecución, autómatas, LLVM.

ABSTRACT

In modern software development, it is common for systems to evolve through small incremental changes. This characteristic can be exploited in various scenarios, particularly in regression testing, where a fundamental question arises: when a part of a previously verified program is modified, is it necessary to re-evaluate the entire system, or can the verification effort already invested be reused? Specifically, can the fact that two versions share a large portion of their code be leveraged to reduce the testing effort?

In this context, this thesis explores and implements Mimicry Monitors (MM), a technique that precisely capitalizes on the common fragments between two versions of a program. The technique allows runtime verification, to determine whether the behavior of a program under analysis (PUA) can be mimicked by a reference or oracle program (OP), without the need to execute the latter.

To validate the technique, an experimental evaluation was conducted on five GNU Core Utilities tools, selecting two versions of each and using their existing test suites to simulate real-world regression testing scenarios. The effectiveness of Mimicry Monitors is defined as their ability to issue early verdicts about the existence of a counterpart in the OP for the current execution of the PUA. The results show that the effectiveness of MMs is intrinsically tied to the nature of the changes between versions. For programs with localized modifications such as cat, up to 60.87 % of cases were successfully verified, enabling early termination. In contrast, programs with more fundamental structural changes like ls did not yield equally favorable results.

These results demonstrate that Mimicry Monitors are a valuable tool for optimizing regression testing, partially validating the hypothesis that redundant executions can be avoided by analyzing shared code fragments.

Keywords: Verification, common fragments, regression testing, runtime monitor, automata, LLVM.

AGRADECIMIENTOS

Toda persona que esté leyendo esto debe saber lo mucho que ansié escribir mis agradecimientos. Este trabajo no hubiera sido posible sin el apoyo de tantas personas, que me cuesta ponerlo en un par de líneas. Voy a intentar ser concisa, aunque dudo que lo logre.
Agradezco:
A mis directores, Javi "JaviGod" Godoy, por motivarme siempre y por traerme Coca Zero cuando más lo necesitaba, pero sobre todo por aguantarme todos los días, y a Diego "Nimbus" Garbervetsky, por ser el mejor mentor, y ayudarme a enfrentar mis peores miedos (LLVM).
También a mis no-directores, Victor Braberman, por su gran capacidad de encontrar contraejemplos, e idear una herramienta tan interesante, y a Sebastián Uchitel, por su guía y sus valiosos consejos.
Al jurado, por su tiempo y buena predisposición.
A mis papás, quienes cultivaron siempre mi curiosidad y mis ganas de aprender, por su apoyo incondicional e infinita paciencia.
A mis hermanos, Joaquín y Nacho, por malcriarme siempre y ser mi ejemplo a seguir.
A mis amigas, que son lo más lindo que tengo, Martu, Chiari, Loli, Jua, Chuver y Puguis. Gracias por escucharme hablar siempre de autómatas y cosas complicadas, por reírse y hacerme reír, por atender todos mis llamados y venir al rescate, y por compartir su vida conmigo.
A mis amigos de la facultad, Gonza y Tincho, hacer la carrera con ustedes fue un placer y un honor.
A la gente de LAFHIS, por generar un ambiente tan cálido y divertido donde trabajar. En particular a mis amigos de la 2112, Flopynator, por mostrarme siempre fotos de la Murrú, y Herni, por prestarme su Stormtrooper para el parcial más difícil de mi carrera.
A la Universidad de Buenos Aires, por ofrecer una educación de tan altísima calidad, y permitirme formarme como profesional.
A la Facultad de Ciencias Exactas y Naturales, por volverse mi refugio, y por abrirme la cabeza.
A todos los docentes que atravesaron mi trayectoria académica, por su dedicación y buena voluntad.
Y por último, pero no menos importante, gracias a mis más fieles compañeras, Migui y Tosti, por acompañarme en incontables noches de estudio, y por enseñarme que, a veces, también hay que descansar.
¡Gracias!

Fig. 0.1: La gente de la 2112

"Pensé en un laberinto de laberintos, en un laberinto sinuoso y extenso que abarcaría el pasado y el futuro y de alguna manera involucraría a las estrellas."

- Jorge Luis Borges, *'El jardín de senderos que se bifurcan.'*

Índice general

1. INTRODUCCIÓN

> *Einstein repeatedly argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.*
>
> — Fred Brooks, *No Silver Bullet [9]*

En el desarrollo de software moderno, es común que los sistemas evolucionen a través de pequeños cambios incrementales. Cuando modificamos una parte de un programa que ya ha sido verificado, surge una pregunta fundamental: ¿debemos verificar todo el sistema nuevamente, o podemos aprovechar el trabajo de verificación ya realizado? Este problema es especialmente relevante en escenarios de pruebas de regresión, donde pequeños cambios requieren re-ejecutar extensos conjuntos de pruebas, en procesos de depuración donde queremos entender si un bug introducido afecta comportamientos previamente válidos, y en técnicas de fuzzing donde se generan múltiples inputs.

La verificación completa de sistemas complejos puede ser extremadamente costosa en tiempo y recursos computacionales. En proyectos de software industrial, estas verificaciones pueden tomar horas o incluso días, representando un cuello de botella en el ciclo de desarrollo. Sin embargo, cuando los programas comparten fragmentos significativos de código, gran parte de esta verificación puede ser redundante. El desafío central radica en determinar, sin ejecutar completamente ambos programas, si el comportamiento de un programa modificado puede ser imitado o replicado por el programa original ya verificado. Realizar este análisis en tiempo real, permite tomar decisiones inmediatas sobre si continuar con la ejecución completa o no. Para abordar esta incógnita, este trabajo implementa y analiza los Mimicry Monitors [16], una técnica innovadora de análisis dinámico que permite verificar en tiempo real si la ejecución de un programa bajo análisis (PUA) tiene una contraparte válida en un programa de referencia u oráculo (OP), evitando la ejecución del programa oráculo mediante la construcción de un autómata de monitoreo. Este simula sus posibles comportamientos, y al instrumentar el PUA con él, genera veredictos inmediatos sobre la validez de la ejecución basados en el análisis de fragmentos de código comunes.

1.1. Motivación

Antes de dar inicio al desarrollo propio del trabajo, resulta conveniente presentar un ejemplo concreto que servirá como hilo conductor a lo largo de esta tesis, permitiendo ilustrar de manera clara y sistemática cada uno de los pasos que componen la metodología propuesta. El ejemplo seleccionado constituye una versión adaptada del caso de estudio presentado en el paper original de Postolski et al. [16], manteniendo deliberadamente la estructura de control fundamental de ambos programas involucrados, así como el alineamiento σ que establece la correspondencia entre los fragmentos de código comunes. Esta adaptación preserva los aspectos esenciales del problema de verificación mientras simplifica ciertos detalles implementativos que podrían oscurecer la comprensión de los conceptos

centrales.

La elección de este ejemplo particular se fundamenta en varias consideraciones: en primer lugar, presenta un escenario realista de evolución de software donde una versión más reciente incorpora modificaciones típicas como validaciones adicionales, correcciones de bugs y optimizaciones de código. En segundo lugar, la complejidad del ejemplo es suficiente para demostrar la efectividad del enfoque propuesto, pero no tan elevada como para dificultar el seguimiento de los pasos algorítmicos. Finalmente, la estructura de control resultante permite observar claramente cómo los Mimicry Monitors pueden anticipar el comportamiento de ejecuciones alineadas sin necesidad de ejecutar efectivamente el programa oráculo. A lo largo de los siguientes capítulos, este ejemplo será retomado consistentemente para mostrar la aplicación práctica de cada etapa del proceso de construcción.

Programa Oráculo (OP)

```
1  int main(int argc, char *argv[]) {
2      int R = 10;
3      int P = argv[1] ? atoi(argv[1]) : 0;
4      int A = 0;
5
6      if (Cond1(P)) {
7          A = AComputation(P);
8          if (Cond(A)) {
9              R = HeavyRComputation(P, A);
10         } else {
11             R = RComputation(P, A);
12         }
13     }
14
15     return R;
16 }
```

Programa Bajo Análisis (PUA)

```
1  int main(int argc, char *argv[]) {
2      int R = 10;
3      int P = argv[1] ? atoi(argv[1]) : 0;
4      int A = 0;
5
6      if (Cond0(P)) {
7          logMessage("Error0");
8      } else {
9          if (Cond1(P)) {
10             A = AComputation(P);
11             if (Cond(A)) {
12                 R = HeavyRComputation(P, A);
13             } else {
14                 R = ChangedRComputation(P, A);
15             }
16         } else {
17             logMessage("Error");
18         }
19         logValue(R);
20     }
21
22     return R;
23 }
```

Fig. 1.1: Comparación entre el Programa Oráculo (OP) y el Programa Bajo Análisis (PUA). Resaltado en azul se observan los fragmentos comunes, y en amarillo los que difieren entre ambas versiones.

El mismo consiste en dos versiones de un mismo programa que comparten fragmentos significativos de código común. Como se observa en la Figura 1.1, el Programa Oráculo (OP) se corresponde a una versión estable del software, mientras que el Programa Bajo Análisis (PUA) representa una versión más reciente. Las principales diferencias entre ambas versiones incluyen:

- **Validación adicional**: El PUA incorpora una nueva precondición `Cond0(P)` en la línea 3 que no existe en el OP, añadiendo manejo de errores específicos.
- **Corrección de funcionalidad**: La línea 12 del PUA utiliza `ChangedRComputation(P, A)` en lugar de `RComputation(P, A)` del OP (línea 9), lo cual representa la corrección de un bug.

No obstante estas diferencias, ambos programas mantienen fragmentos comunes sustanciales, particularmente en las operaciones de inicialización (`R = InitialValueR()`), el

cómputo principal (`A = AComputation(P)` y `R = HeavyComputation(P, A)`), y la instrucción de retorno. Estos fragmentos comunes constituyen la base para establecer la función de alineamiento σ que permitirá la construcción del Mimicry Monitor correspondiente.

Tomando estos programas y la función de emparejamiento σ como inputs, se construye, a través de una serie de pasos descritos en la sección 3.2.2 el autómata monitor con el cual se instrumentará el PUA. El autómata resultante puede observarse en la figura 1.2

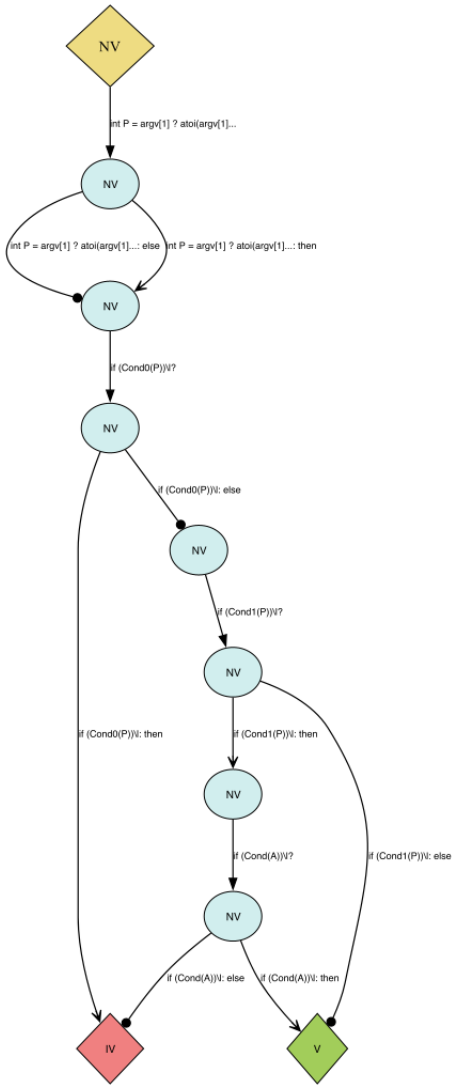


Fig. 1.2: Automata monitor resultante para los inputs presentados en la figura 1.1.

El comportamiento del monitor se puede analizar a través de diferentes trayectorias de ejecución. A continuación, algunos ejemplos para ilustrarlas:

Escenario 1 - Veredicto IV: Si el PUA se ejecuta con un input P tal que `Cond0(P)` evalúa a `True`, el autómata alcanza inmediatamente un estado final etiquetado como IV (Imposible de Verificar). Este resultado es coherente, ya que esta validación no existe en el OP, provocando que el PUA termine prematuramente mientras que el OP continuaría

ejecutando su cuerpo principal. Esta divergencia fundamental hace imposible encontrar ejecuciones σ-alineadas.

Escenario 2 - Veredicto V: En contraste, si el PUA se ejecuta con un input P donde `Cond0(P)` evalúa a `False`, `Cond1(P)` evalúa a `True`, y `Cond(A)` evalúa a `True`, el monitor emite un veredicto V (Verificado). Este veredicto indica que para toda continuación posible de esta ejecución del PUA existe una contraparte σ-alineada en el OP, permitiendo abortar la ejecución de forma temprana.

El valor práctico del monitor, en el contexto de testing de regresión, se evidencia en el segundo escenario. Al detectar el veredicto V justo después de que `Cond(A)` evalúe a `True`, es posible evitar la ejecución de `R = HeavyRComputation(P, A)`, una operación computacionalmente costosa. Esta optimización representa un ahorro significativo de recursos cuando se aplica a conjuntos extensos de pruebas de regresión.

1.2. Hipótesis

Este trabajo parte de la hipótesis de que, en situaciones donde los cambios entre versiones de un programa son pequeños, como ocurre comúnmente en escenarios de mantenimiento, es altamente probable que un alto porcentaje de las ejecuciones del nuevo programa conserven un comportamiento compatible con el programa original.

En esos casos, el Mimicry Monitor permitiría determinar, en tiempo real, si una ejecución del PUA podría haber sido producida por el OP, evitando ejecutar el OP en paralelo. Esta capacidad de anticipar veredictos podría aprovecharse para reducir ejecuciones redundantes, detectar rápidamente cambios de funcionalidad, y, en definitiva, ahorrar tiempo y recursos en diferentes contextos, como lo puede ser el testing de regresión.

1.3. Contexto

El trabajo desarrollado en el marco de esta tesis parte de las ideas preliminares presentadas en el position paper Verification of Programs with Common Fragments [16]. Este trabajo introduce una técnica novedosa en el ámbito del análisis de software, cuyo eje central es la explotación de fragmentos comunes entre programas distintos, o distintas versiones de un mismo programa, para facilitar procesos de verificación y monitoreo en tiempo de ejecución.

En ingeniería de software, es frecuente encontrarse con múltiples versiones de un sistema que comparten una proporción grande de código. Esta situación puede observarse, por ejemplo, durante el desarrollo incremental de un producto, en procesos de mantenimiento, e incluso en tareas de refactorización. En estos casos, aunque ciertas secciones del programa cambien, muchas otras permanecen idénticas o son funcionalmente equivalentes. El trabajo mencionado propone una manera sistemática de aprovechar estos fragmentos, de modo que se puedan correlacionar las ejecuciones de dos programas mediante una noción de *alineación semántica*.

A partir de esta alineación, se define una estructura denominada Mimicry Monitor (MM), que se construye sobre los grafos de flujo de control (CFG) de ambos programas y un conjunto de emparejamientos entre sus nodos correspondientes. El MM actúa como un autómata de monitoreo que opera en paralelo a la ejecución de un solo programa, el deno-

minado Program Under Analysis (PUA), y permite determinar, en tiempo de ejecución, si dicha ejecución podría ser reproducida por un segundo programa, el Oracle Program (OP), sin necesidad de ejecutarlo efectivamente. Esto habilita la emisión de veredictos parciales y anticipados sobre la capacidad del OP de imitar la ejecución observada, con aplicaciones directas en escenarios como testing de regresión, fuzzing diferencial y multi-ejecución controlada.

El trabajo original introduce el concepto de *ejecución alineada*, detalla las condiciones necesarias para construir un monitor válido, y ofrece una presentación conceptual del procedimiento. Sin embargo, la construcción del MM se presenta a un alto nivel de abstracción, y no se dispone de una herramienta automatizada que permita instanciar estos monitores de forma generalizada para distintos pares de programas.

Es en este punto donde se enmarca el aporte de esta tesis. En primer lugar, se profundiza el estudio del proceso de construcción del Mimicry Monitor, abordando los desafíos prácticos y teóricos que surgen al implementar el constructor de manera concreta. En segundo lugar, se propone una evaluación empírica del comportamiento de los monitores generados, aplicándolos a un conjunto de benchmarks. Esta evaluación tiene por objetivo analizar su eficacia en términos de precisión, y capacidad de anticipación de veredictos.

En síntesis, esta tesis se propone como un puente entre la idea introducida en [16] y una implementación robusta que permita explorar, validar y extender el alcance práctico de los Mimicry Monitors como herramienta de análisis y verificación en sistemas reales.

1.4. Contribuciones

Este trabajo presenta una serie de contribuciones tanto en el plano teórico como práctico, orientadas a validar y extender el uso de Mimicry Monitors:

1. Aportes en el diseño conceptual del monitor

   Se identificaron posibles optimizaciones en varias instancias del proceso de construcción formal del monitor. Entre ellas, la introducción de un mecanismo de *poda temprana*, y de un paso de determinización. Ambas modificaciones tienen como objetivo producir instancias del monitor más compactas y comprensibles. Adicionalmente, se eliminó la noción de *Veredictos sobre Prefijos* presentada en [16], ya que a fines de la aplicación práctica no tenían un propósito claro, y no aportaban información de valor para el análisis.

2. Implementación integral de la herramienta

   Se desarrolló una implementación completa del proceso de construcción e instrumentación del monitor. Para la instrumentación y el pre-procesamiento de los programas input, se utilizó LLVM como infraestructura base, permitiendo analizar e intervenir programas a nivel de LLVM IR. Esta implementación automatiza el análisis de accesos a memoria, la construcción del grafo de flujo de control (CFG), la construcción del monitor propiamente dicha, y la inserción de llamadas al monitor durante la ejecución.

3. Selección de benchmarks

   Se seleccionó un conjunto de benchmarks para simular escenarios reales donde los Mimicry Monitors podrían ser útiles: pruebas de regresión frente a pequeñas refac-

torizaciones. Los benchmarks consideran distintos tamaños, patrones de control y niveles de similitud entre OP y PUA.

4. Evaluación

   Se ejecutó la herramienta sobre estos benchmarks, observando el comportamiento del monitor y la proporción de veredictos positivos frente a veredictos negativos.

5. Análisis de resultados

   Los resultados obtenidos muestran que los Mimicry Monitors pueden emitir veredictos pertinentes en una porción significativa de las ejecuciones analizadas, especialmente cuando los cambios entre versiones son acotados y no afectan variables centrales del programa. Se observa una reducción sustancial en la cantidad de tests que deben ser ejecutados por completo en el contexto de testing de regresión, validando la hipótesis planteada. Además, se discuten las limitaciones actuales y posibles extensiones para mejorar su aplicabilidad general.

1.5. Trabajos Previos

La idea de comparar la ejecución dinámica de dos programas es estudiada en una serie de trabajos que abordan diferentes aspectos de la verificación diferencial y el análisis comparativo de software.

1.5.1. Comparación dinámica de múltiples ejecuciones

Palikareva et al.[15] introducen en su paper *Shadow of a Doubt*, una herramienta que utiliza ejecución simbólica para detectar divergencias de comportamiento entre versiones de software. Su enfoque se centra en identificar casos donde dos versiones de un programa producen salidas diferentes para las mismas entradas, lo cual es fundamental para detectar cambios introducidos durante el desarrollo. La técnica emplea análisis simbólico para explorar sistemáticamente el espacio de entradas y generar casos de prueba que expongan diferencias comportamentales. Para ello se mantiene una ejecución para cada versión del programa.

Postolski et al. [17] extienden esta línea de investigación al ámbito del testing diferencial de rendimiento, proponiendo técnicas basadas en simulación para comparar el comportamiento temporal de diferentes versiones de software. Su trabajo demuestra cómo el análisis diferencial puede aplicarse no solo a la corrección funcional sino también a propiedades no funcionales como el rendimiento. Es una sugerencia en este trabajo la que impulsa el desarrollo de los Mimicry Monitors.

En el contexto de model-checking para detectar errores de regresión, Jakobs y Pollandt [8] proponen un algoritmo que evita analizar caminos que no producirán violaciones de aserciones. Su enfoque construye un “autómata de diferencia” y es insensible al flujo de datos, lo que puede ser impreciso cuando se modifica o elimina código. Nuestra propuesta, en cambio, es más adecuada para monitoreo en tiempo de ejecución, ya que es sensible tanto al flujo de control como al de datos. Técnicas anteriores de selección de tests de regresión [18, 19] comparten ciertas similitudes, pero suelen ser conservadoras y menos precisas. Finalmente, enfoques formales como la verificación relacional o de equivalencia [20, 21, 12, 11, 10, 7, 2, 1], permiten detectar alineamientos complejos, aunque implican altos costos computacionales.

1.5.2. Limitaciones y Oportunidades

Aunque estos trabajos previos establecen una base sólida para la comparación de programas, presentan ciertas limitaciones que motivan el desarrollo de nuevas aproximaciones. En primer lugar, varios de los trabajos previos presentados presentan cierto grado de imprecisión, debido a que son insensibles al flujo de datos [8], o a que son conservadores al encontrarse con áreas de código modificadas [18] [19]. Estas limitaciones identificadas en el estado del arte motivan el desarrollo de nuevas técnicas que aborden específicamente estos desafíos. En particular se busca remediar estas limitaciones con un mecanismo que es sensible tanto al data-flow como al path de la ejecución analizada.

2. CONCEPTOS Y DEFINICIONES PRELIMINARES

2.1. Nuevos Conceptos

Antes de dar inicio al desarrollo propio del trabajo de implementación, se presentará el marco teórico y conceptual necesario para comprender de forma integral los Mimicry Monitors (MM). El objetivo principal de esta herramienta es permitir al programador monitorear la ejecución de un programa respecto a otro, sin necesidad de mantener una ejecución sombra de ninguno de ellos. Es decir, no es necesario ejecutar simultáneamente ambas versiones de un programa para establecer comparaciones entre sus comportamientos. Para ello, se consideran tres actores fundamentales: dos programas, denominados Programa Oráculo (OP, por Oracle Program) y Programa Bajo Análisis (PUA, por Program Under Analysis), y una relación de emparejamiento parcial σ entre los nodos de sus respectivos Control Flow Graphs (CFG). Esta correspondencia σ refleja fragmentos de código comunes o equivalentes entre ambos programas y constituye la base sobre la cual se definen las ejecuciones alineadas. En el alcance de este trabajo, no se aborda la construcción de σ, se considera que es dada como input por el usuario. Se dice que dos ejecuciones están σ-alineadas si son equivalentes en su comportamiento cuando se restringen a los nodos emparejados por σ. Intuitivamente, esto significa que, en los fragmentos de código compartido, ambas ejecuciones recorren las mismas instrucciones en el mismo orden, produciendo estados de memoria observacionalmente equivalentes. El MM tiene por finalidad verificar, en run-time, si la ejecución parcial del PUA podría ser replicada por el OP en los fragmentos compartidos.

2.1.1. Seguimiento del Ejemplo motivador

Retomando el ejemplo propuesto en la figura 1.1, la relación de emparejamiento estaría conformada por los pares: (2, 2), (3, 3), (4, 4), (6, 9), (7, 10), (8, 11), (9, 12) y (15, 22). Donde cada coordenada hace referencia a una línea del OP y del PUA respectivamente. Notar que, formalmente, σ está definida sobre nodos de los CFGs, y cada nodo se corresponde con una línea del programa. Considerando los escenarios planteados en la sección 1.1, puede ayudar a ilustrar con mayor claridad el concepto de ejecuciones σ-alineadas:

Escenario 1 - Ejecución imposible de alinear: Si el PUA se ejecuta con un input P tal que `Cond0(P)` evalúa a `True`, las líneas ejecutadas serían 2-3-4-6-7-22, de las cuales 2, 3, 4 y 22 pertenecen a σ. Por el otro lado, si bien no hay una ejecución concreta del OP, sabemos por su estructura de control que toda secuencia de líneas que represente una ejecución válida, tendrá como prefijo la secuencia 2-3-4-6, todas pertenecientes a σ. Inmediatamente surge una inconsistencia: Cualquier ejecución del OP debe pasar por la línea 6, que está emparejada con la línea 9 del PUA. Sin embargo, la ejecución actual del PUA pasará por la línea 22, que está emparejada con la línea 15 del OP, sin pasar nunca por la 9, lo que implica la ejecución de fragmentos compartidos en distinto orden. Lo que resulta en una ejecución sin posibilidad de σ-alineamiento.
Escenario 2: Ejecución σ-alineada: Considerando el segundo escenario donde el PUA se ejecuta con un input P tal que `Cond0(P)` evalúa a `False` y `Cond1(P)` evalúa a `True`,

las líneas ejecutadas serían 2-3-4-9-10-11-12-19-22, de las cuales 2, 3, 4, 9, 10, 11, 12 y 22 pertenecen a σ. Por el lado del OP, una ejecución válida que satisfaga `Cond1(P) = True` seguiría la secuencia 2-3-4-6-7-8-9-15, donde todas estas líneas pertenecen a σ.

En este caso, se observa consistencia: ambas ejecuciones recorren los fragmentos compartidos en el mismo orden relativo. La secuencia de nodos σ en el PUA (2, 3, 4, 9, 10, 11, 12, 22) se mapea correctamente con la secuencia correspondiente en el OP (2, 3, 4, 6, 7, 8, 9, 15) a través de la relación σ. Los estados de memoria en cada par de nodos emparejados son observacionalmente equivalentes, lo que resulta en ejecuciones **σ-alineadas**. El MM puede verificar exitosamente que la ejecución parcial del PUA podría ser replicada por el OP en los fragmentos compartidos, cumpliendo con la condición fundamental para el σ-alineamiento.

Para visualizar esta noción se creó una herramienta con la que pueden apreciarse los dos escenarios presentados de forma dinámica, la misma se encuentra disponible en https://felicitasgarcia.github.io/SigmaVisualizer/. En la figura 2.1 se puede observar la pantalla inicial del visualizador. En ella se encuentra el código del OP y del PUA del ejemplo del motivador, junto a su monitor final. Se pueden reproducir ejecuciones distintas del PUA que se corresponden con los escenarios 1 y 2 planteados anteriormente, y observar cómo se va recorriendo el monitor para obtener un veredicto.

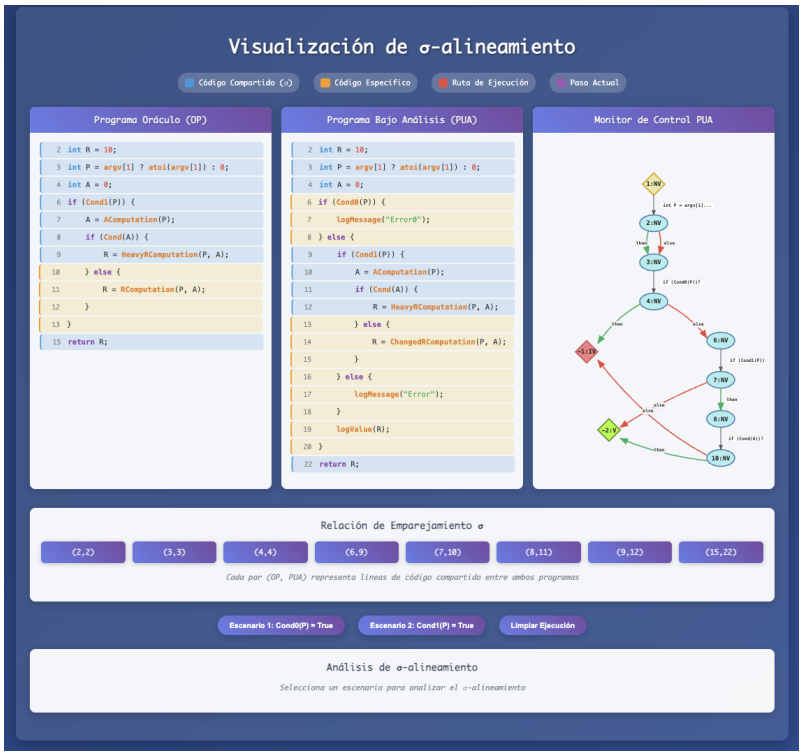


Fig. 2.1: Escenario inicial de la herramienta de visualización. En la parte central se encuentran los programas OP y PUA, con su emparejamiento resaltado en azul. A su izquierda el monitor. Debajo se encuentran 3 botones que permiten visualizar el escenario 1, escenario 2, y limpiar la imagen. Una vez seleccionado un escenario, se incluye en la porción inferior una breve descripción de lo sucedido, junto a los botones necesarios para reproducir el paso a paso de la ejecución.

2.1.2. Ejecuciones σ-alineadas

Formalmente, una ejecución se representa como una secuencia de triplas $\langle$before:State , node:CFG.Node, after:State$\rangle$, donde cada tripla describe el estado de memoria antes y después de ejecutar un nodo del CFG. Sean t_P y t_Q dos ejecuciones de programas P y Q respectivamente, y una correspondencia $\sigma \subseteq \text{Nodes}(CFG(P)) \times \text{Nodes}(CFG(Q))$. Sean $i_P = [0...\|t_P\| - 1]$ e $i_Q = [0...\|t_Q\| - 1]$ los conjuntos de índices válidos para ambas ejecuciones, de ser estas finitas. De lo contrario $i_P = \mathbb{N}$ e $i_Q = \mathbb{N}$. Se define la noción de σ-alineación como sigue:

Definición 1. *Sean t_P y t_Q ejecuciones de los programas P y Q, y σ un emparejamiento parcial entre nodos de sus CFG. Se dice que t_P y t_Q están **σ-alineadas** si existe una función parcial inyectiva estrictamente creciente $m \subseteq i_P \to i_Q$ tal que:*

- $\forall i \in dom(t_P) : \ t_P(i).node \in Dom(\sigma) \Rightarrow i \in Dom(m)$,
- $\forall j \in dom(t_Q) : \ t_Q(j).node \in Img(\sigma) \Rightarrow j \in Img(m)$,
- $\forall (i,j) \in m : \ (t_P(i).node, \ t_Q(j).node) \in \sigma$,
- $\forall i \in Dom(m) : \ Read(stmt_P, \ t_P(i).before) \equiv Read(stmt_Q, \ t_Q(m(i)).before)$.

Aquí, $stmt_P$ y $stmt_Q$ son los fragmentos de código asociados a los nodos correspondientes en cada ejecución:

$$stmt_P = Stmt(t_P(i).node) \qquad y \qquad stmt_Q = Stmt(t_Q(m(i)).node)$$

Esta definición asegura que, en los nodos compartidos, ambas ejecuciones acceden a las mismas áreas del estado y realizan transformaciones equivalentes. En consecuencia, si se verifica la σ-alineación entre la ejecución observada del PUA y alguna ejecución posible del OP, entonces cualquier propiedad que dependa únicamente del estado observable en dichos nodos será preservada.

2.1.3. Mimicry Monitors

Los Mimicry Monitors son autómatas construidos a partir de los CFG del OP y del PUA, junto con la correspondencia σ de forma estática. La herramienta de monitoreo propiamente dicha funciona a través de la instrumentación del PUA con el autómata monitor. De ahora en más se referirá a los autómatas de monitoreo y a la herramienta instrumentada con ellos como Mimicry Monitor indistinguidamente. La versión instrumentada del PUA es capaz de emitir veredictos sobre cualquier ejecución en función de si existe (o no) una ejecución σ-alineada correspondiente en el OP. Crucialmente, este análisis se realiza sin ejecutar el OP, lo que representa una mejora sustancial en términos de eficiencia y escalabilidad.

El MM opera como un autómata que, en cada paso de ejecución del PUA que influya sobre el flujo de control del programa, actualiza su estado interno y clasifica la ejecución de acuerdo con uno de los siguientes veredictos:

- **NV (Non Verified):** existe un prefijo de ejecución del OP que es σ-alineado con la ejecución observada del PUA hasta ese punto. Aún no está determinado que se haya conseguido un veredicto concluyente.

- **V (Verified):** todas las posibles continuaciones del PUA son compatibles con una ejecución σ-alineada del OP.
- **IV (Impossible to Verify):** Ninguna continuación del PUA será determinada como Verificada.

Los veredictos V e IV son terminales, es decir que indican la finalización del monitoreo. Estos veredictos se emiten en tiempo real y pueden utilizarse para tomar decisiones en sistemas de testing, ejecución múltiple o fuzzing, por ejemplo abortando ejecuciones innecesarias o priorizando la exploración de caminos no cubiertos por versiones anteriores del software. Para asegurar que la técnica sea *sound*, el enfoque supone que las instrucciones OP no compartidas no producen errores y que los ciclos en código no compartido finalizan, lo cual garantiza *soundness* para veredictos NV. Esta suposición es razonable si se ha verificado que una ejecución OP termina exitosamente para una entrada dada. De manera similar, la correctitud de los veredictos V anticipados requiere que las instrucciones PUA no compartidas no generen fallas y que sus ciclos terminen adecuadamente.

En los capítulos siguientes se abordará la implementación detallada del constructor de MM, así como la aplicación de estos monitores a un conjunto de benchmarks que permitirá evaluar su rendimiento y aplicabilidad en distintos escenarios.

2.2. Conceptos Preliminares

Para comprender en profundidad la construcción de los Mimicry Monitors y su aplicación en tareas de verificación dinámica, es necesario presentar algunos conceptos fundamentales del análisis formal de sistemas y teoría de autómatas. En esta sección se introducen tres nociones centrales: la composición paralela de autómatas sincronizada, la minimización por bisimulación, y la determinización de autómatas. Estas operaciones son componentes clave en el proceso de construcción del monitor descrito en [16].

2.2.1. Composición paralela sincronizada en etiquetas compartidas

La composición paralela de autómatas permite modelar la ejecución conjunta de dos o más sistemas que interactúan parcialmente a través de un conjunto compartido de acciones [6]. En el contexto de los Mimicry Monitors, esta operación se utiliza, junto a otras, para combinar los autómatas correspondientes al OP, al PUA y a los rastreadores de flujo de datos (*data-flow tracking automata*, o DFTAs de ahora en más), generando así un único autómata compuesto que modela todas las posibles interacciones entre ellos.

Definición 2 (Composición paralela sincronizada)**.** *Sean* $A_1 = (Q_1, \Sigma_1, \delta_1, q_1^0)$ *y* $A_2 = (Q_2, \Sigma_2, \delta_2, q_2^0)$ *dos autómatas finitos no deterministas, donde* Q_i *es el conjunto de estados,* Σ_i *el conjunto de etiquetas (acciones),* $\delta_i : Q_i \times \Sigma_i \to \mathcal{P}(Q_i)$ *la función de transición, y* q_i^0 *el estado inicial.*

La ***composición paralela sincronizada*** $A = A_1 \parallel A_2$ *es el autómata definido como:* $A = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_1^0, q_2^0))$

donde la función de transición δ *se define como:*

- *Si* $a \in \Sigma_1 \cap \Sigma_2$*:* $\quad \delta((q_1, q_2), a) = \{(q_1', q_2') \mid q_1' \in \delta_1(q_1, a),\ q_2' \in \delta_2(q_2, a)\}$
- *Si* $a \in \Sigma_1 \setminus \Sigma_2$*:* $\quad \delta((q_1, q_2), a) = \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, a)\}$

- *Si* $a \in \Sigma_2 \setminus \Sigma_1$*:* $\quad \delta((q_1, q_2), a) = \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, a)\}$

Esta operación sincroniza ambos autómatas en las etiquetas compartidas, forzándolos a avanzar conjuntamente solo cuando ambos pueden ejecutar una misma acción. Para las acciones exclusivas, cada componente puede avanzar independientemente.

2.2.1.1 Propiedades fundamentales de la composición paralela

La composición paralela sincronizada presenta varias propiedades importantes que son relevantes para la construcción de monitores:

1. **Asociatividad**: $(A_1 \parallel A_2) \parallel A_3 = A_1 \parallel (A_2 \parallel A_3)$. Esta propiedad permite construir composiciones de múltiples autómatas de manera incremental.

2. **Conmutatividad**: $A_1 \parallel A_2 = A_2 \parallel A_1$ salvo isomorfismo de estados. Esto significa que el orden en que se componen los autómatas no afecta el comportamiento resultante.

2.2.1.2 Extensión a múltiples componentes

En la práctica, la composición paralela puede extenderse a más de dos autómatas.

Definición 3. *Sea* $\{A_i = (Q_i, \Sigma_i, \delta_i, q_i^0)\}_{i=1}^n$ *una colección de n autómatas. La composición paralela* $A = \|_{i=1}^n A_i$ *se define como:*

$A = (Q_1 \times Q_2 \times \cdots \times Q_n, \bigcup_{i=1}^n \Sigma_i, \delta, (q_1^0, q_2^0, \ldots, q_n^0))$

donde la función de transición δ *sincroniza todas las componentes que comparten una etiqueta particular. Específicamente, para cualquier estado compuesto*

$(q_1, q_2, \ldots, q_n)$ *y acción* $a \in \bigcup_{i=1}^n \Sigma_i$*:*
$\delta((q_1, q_2, \ldots, q_n), a) = \{(q_1', q_2', \ldots, q_n') \mid q_i' \in \delta_i(q_i, a)$ *si* $a \in \Sigma_i$*, o* $q_i' = q_i$ *si* $a \notin \Sigma_i\}$

2.2.2. Minimización por bisimulación

La minimización de autómatas por bisimulación es una técnica utilizada para reducir la cantidad de estados de un sistema, preservando su comportamiento observable. A diferencia de la minimización clásica para autómatas deterministas, la bisimulación puede aplicarse también en entornos no deterministas, y resulta especialmente útil cuando se quiere preservar información estructural como las ramas de decisión.

Definición 4 (Bisimulación)**.** *Sea* $A = (Q, \Sigma, \delta, q_0)$ *un autómata de transición. Una relación binaria* $R \subseteq Q \times Q$ *es una* ***bisimulación*** *si, para todo* $(p, q) \in R$ *y para toda* $a \in \Sigma$*:*

- *Si* $p \xrightarrow{a} p'$*, entonces existe* $q' \in Q$ *tal que* $q \xrightarrow{a} q'$ *y* $(p', q') \in R$*.*
- *Simétricamente, si* $q \xrightarrow{a} q'$*, entonces existe* $p' \in Q$ *tal que* $p \xrightarrow{a} p'$ *y* $(p', q') \in R$*.*

Definición 5 (Minimización por bisimulación)**.** *La* ***minimización por bisimulación*** *consiste en computar la relación de bisimulación más grande R (la mayor relación de equivalencia que satisface la definición anterior), y luego colapsar todos los estados equivalentes entre sí en un único estado. El resultado es un autómata más pequeño, bisimilar al original.*

En la construcción del Mimicry Monitor, se utiliza una variante de esta minimización que preserva las ramas condicionales observables, conocida como minimización sensible a bifurcaciones (branch-preserving minimization), para mantener la distinción entre decisiones de control que son relevantes en el análisis de alineación. El pseudocódigo puede observarse en el algoritmo 1.

2.2.2.1 Algoritmo de particionamiento para bisimulación

El algoritmo de particionamiento es la técnica más común para calcular la relación de bisimulación máxima. Este algoritmo comienza con una partición inicial del conjunto de estados (generalmente basada en alguna equivalencia observable, como la etiqueta de los estados) y refina iterativamente esta partición hasta alcanzar un punto fijo.

Algorithm 1 Minimización por Bisimulación

1. **Inicialización**: Crear una partición inicial P_0 del conjunto de estados Q. Esta partición puede basarse en cualquier propiedad observable de los estados.
2. **Refinamiento**: Para cada iteración $i \geq 0$:
   a) Para cada bloque B en la partición P_i, y para cada acción $a \in \Sigma$:
      1) Identificar los bloques $C_1, C_2, \ldots, C_k$ en P_i que son alcanzables desde estados en B mediante la acción a.
      2) Refinar B en subbloques $B_1, B_2, \ldots, B_m$ tales que todos los estados en un mismo subbloque B_j pueden alcanzar exactamente los mismos bloques $C_1, C_2, \ldots, C_k$ mediante la acción a.
   b) Si no se produjo ningún refinamiento, detenerse. En caso contrario, sea P_{i+1} la nueva partición refinada y continuar.
3. **Construcción del autómata mínimo**: El autómata mínimo tiene como estados los bloques de la partición final. Las transiciones se establecen entre los bloques según las transiciones entre los estados originales contenidos en esos bloques.

2.2.2.2 Bisimulación débil

En ciertos contextos, especialmente cuando se trabaja con acciones internas o no observables (tradicionalmente denotadas con la letra griega τ), es conveniente utilizar variantes más débiles de la bisimulación:

Definición 6 (Bisimulación débil)**.** *Una relación binaria $R \subseteq Q \times Q$ es una* ***bisimulación débil*** *si, para todo $(p, q) \in R$:*

- *Si $p \xrightarrow{\tau^* a \tau^*} p'$ (donde τ^* representa cero o más transiciones τ consecutivas), entonces existe $q' \in Q$ tal que $q \xrightarrow{\tau^* a \tau^*} q'$ y $(p', q') \in R$.*
- *Simétricamente para transiciones desde q.*

La bisimulación débil permite "saltar"sobre secuencias de acciones internas, tratándolas como si fueran invisibles desde el punto de vista del observador externo. Esto es especialmente útil cuando se quiere abstraer detalles de implementación que no son relevantes para el análisis de comportamiento.

2.2.3. Determinización

La determinización es el proceso mediante el cual se transforma un autómata no determinista en uno determinista, es decir, donde para cada estado y cada símbolo de entrada, existe a lo sumo una transición definida. Esta operación es fundamental para simplificar el análisis formal y aplicar técnicas de verificación que requieren determinismo.

Definición 7 (Determinización de autómatas finitos)**.** *Dado un autómata no determinista* $A = (Q, \Sigma, \delta, q_0)$*, su* ***autómata determinista equivalente*** $A_D = (Q_D, \Sigma, \delta_D, q_0^D)$ *es aquel tal que:*

- $Q_D \subseteq \mathcal{P}(Q)$*: el conjunto de estados del determinista está formado por subconjuntos de estados del autómata original.*
- $q_0^D = q_0$*: el estado inicial es el conjunto formado por el estado inicial del autómata no determinista.*
- $\delta_D(S, a) = \bigcup_{q \in S} \delta(q, a)$*: la función de transición se define como la unión de las transiciones posibles desde cada estado* q *en el conjunto* S*, bajo el símbolo* a*.*

El procedimiento para construir el autómata determinista equivalente se puede observar en el algoritmo 2.

Algorithm 2 Determinización

1. Comenzar con el conjunto inicial q_0 como el estado inicial del nuevo autómata.
2. Crear una estructura (como una lista o cola) para mantener los conjuntos de estados pendientes de procesar.
3. Mientras haya conjuntos no procesados:
   - Extraer un conjunto S de la estructura.
   - Para cada símbolo $a \in \Sigma$:
     - Calcular $T = \bigcup_{q \in S} \delta(q, a)$.
     - Añadir T al conjunto de estados del autómata determinista si aún no ha sido agregado.
     - Definir la transición $\delta_D(S, a) = T$.
4. Repetir hasta que no haya nuevos conjuntos por procesar.

En el contexto de Mimicry Monitors, la determinización puede aplicarse para simplificar el análisis de rutas posibles en tiempo de ejecución, aunque no es estrictamente necesaria en todos los casos.

2.2.3.1 Limitaciones y complejidad de la determinización

La determinización de un autómata no determinista puede resultar en un incremento exponencial en el número de estados. Específicamente, si el autómata original tiene n estados, el autómata determinizado puede tener hasta 2^n estados. Este crecimiento exponencial puede hacer que la determinización sea impráctica para autómatas grandes.

3. DISEÑO

3.1. Introducción a la técnica

En este capítulo se utilizarán las nociones ya presentadas para ofrecer una explicación completa e ilustrativa del proceso de creación de un Mimicry Monitor. El proceso de *setup* para el monitoreo puede dividirse en tres etapas principales, cada una con objetivos y desafíos particulares:

1. **Procesamiento de los programas *input***: Esta etapa implica el análisis estático de los dos programas de referencia. A partir de estos programas se extrae información relevante sobre sus estructuras de control y patrones de ejecución, que servirán como base para la construcción del monitor. Adicionalmente se extrae información sobre los accesos a memoria de cada uno, para la posterior construcción de los DFTAs.

2. **Construcción del autómata monitor**: Utilizando los datos obtenidos en la etapa anterior, se genera un autómata que modela el comportamiento esperado del sistema. Este autómata puede ser no determinista en un principio, y requerirá transformaciones como la determinización y minimización para ser lo suficientemente compacto, para instrumentar el programa bajo análisis de forma *lightweight*.

3. **Instrumentación del programa bajo análisis con el monitor**: Finalmente, se modifica el programa objetivo —el que se desea monitorear— para insertar llamadas al monitor en puntos estratégicos, generalmente asociados a condiciones de control. De esta forma, durante la ejecución, el monitor podrá verificar si las transiciones observadas son consistentes con el modelo y emitir un veredicto si se detecta una desviación.

Cada una de estas fases será detallada en las subsecciones siguientes, ilustrando su lógica subyacente. El objetivo es proporcionar una comprensión clara de cómo se lleva a cabo el monitoreo en tiempo de ejecución mediante un enfoque basado en autómatas, destacando tanto las ventajas de esta técnica como las decisiones de diseño adoptadas.

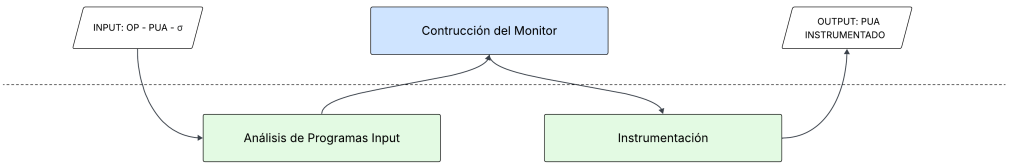


Fig. 3.1: Esquema del proceso de Construcción e Instrumentación del Monitor

La figura 3.1 ilustra el esquema del proceso de construcción del monitor para el sistema de monitoreo de mímica. El diagrama presenta claramente los elementos de entrada (INPUT: OP - PUA - σ) y salida (OUTPUT: PUA INSTRUMENTADO) del sistema, así como los tres componentes principales que conforman el proceso de transformación. La representación visual distingue dos niveles de abstracción mediante una línea punteada

horizontal. Los componentes ubicados por encima de esta línea (Construcción del Monitor) operan a un nivel conceptual más alto, mientras que los dos módulos situados por debajo (Análisis de Programas Input e Instrumentación) representan operaciones de bajo nivel que requieren manipulación directa del código fuente y binario de los programas de entrada. Esta separación refleja la necesidad de utilizar infraestructuras especializadas de análisis e instrumentación de bajo nivel, específicamente LLVM, para realizar las transformaciones necesarias sobre el código de los programas objetivo. De esta manera, se establece una clara distinción entre las operaciones de alto nivel relacionadas con la lógica del monitor y las operaciones de bajo nivel requeridas para la manipulación efectiva del código.

3.2. Arquitectura del proceso

En esta sección se presentan, de manera abstracta y general, los pasos esenciales involucrados en cada una de las etapas mencionadas anteriormente, con especial énfasis en la construcción del monitor, dado que constituye el componente de mayor complejidad conceptual del trabajo. Para ello, se describen los pseudoalgoritmos correspondientes, junto con una explicación detallada de sus etapas principales. Como guía para ilustrar los distintos pasos del algoritmo de construcción, se retomará el ejemplo presentado en la motivación 1.1.

Si bien la figura 3.1 da una intuición de la estructura del proyecto, cada una de las etapas descritas, se divide a su vez en subetapas más concretas.

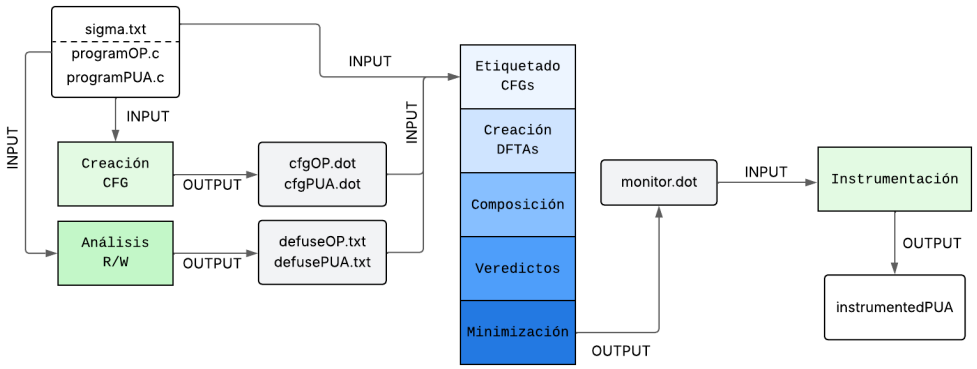


Fig. 3.2: Esquema en mayor detalle del proceso de Construcción e Instrumentación del Monitor. En blanco se observan el input de usuario, y el output final del proceso, en gris los inputs y outputs intermedios. Por el otro lado, en verde y azul las distintas etapas del proceso.

En la figura 3.2 se profundiza en la técnica desarrollada. Se observa cómo el proceso comienza con la creación de los grafos de flujo de control (CFGs) de los programas dados como input, junto con un análisis de operaciones de lectura y escritura (R/W). Esta información, junto con el σ-alineamiento, provisto también como input de usuario, alimenta el proceso de construcción del autómata, que a su vez consiste de 5 subetapas. Finalmente la monitor resultante es utilizada en la etapa de instrumentación del programa bajo análisis.

3.2.1. Análisis de programas Input

Como primera instancia se realiza un análisis de los programas input. Esencialmente son dos las tareas que deben realizarse en esta etapa.

3.2.1.1 Obtención de CFG del OP y del PUA

Para comenzar con la construcción del Mimicry Monitor, es imprescindible contar con información precisa sobre el flujo de control de ambos programas involucrados: el Programa Oráculo (OP) y el Programa Bajo Análisis (PUA). Esto se debe a que las decisiones de control, tales como condicionales, ciclos y llamadas a funciones, influyen de manera directa en la posibilidad de que exista una ejecución del OP que pueda replicar (o imitar) la trayectoria seguida por el PUA. En otras palabras, la estructura del flujo de control incide directamente en la existencia o ausencia de ejecuciones alineadas entre ambos programas.

Una vez construidos los grafos de flujo de control (CFGs), es necesario identificar aquellas partes del código que son equivalentes en ambos programas. Para ello se recurre a una relación de emparejamiento entre nodos de los CFGs, que actúa como insumo fundamental para el etiquetado y la posterior construcción del monitor. Esta relación, denotada como σ, vincula instrucciones o bloques que no han sido modificados entre versiones del programa. La construcción de esta relación está fuera del alcance de esta tesis, por lo que en los experimentos realizados se asumió que σ es provista como entrada externa. Concretamente, se consideraron como equivalentes aquellas líneas de código que permanecieron inalteradas entre dos commits, una suposición razonable en el contexto de pruebas de regresión.

Esta relación de emparejamiento σ es esencial para establecer correspondencias entre las ejecuciones de ambos programas, y constituye la base para la alineación semántica que permite verificar si las decisiones de control observadas en el PUA pueden ser, en principio, imitadas por el OP.

3.2.1.2 Análisis de Escrituras y Lecturas de Variables

Adicionalmente, resulta crucial extraer información detallada sobre las operaciones de lectura y escritura realizadas sobre las variables del programa. Este análisis es particularmente relevante debido a que las escrituras no sincronizadas, aquellas que modifican el valor de una variable y no están en σ, pueden afectar directamente la validez de ciertas aseveraciones. En tales casos, un valor escrito por el PUA podría no estar presente en ninguna ejecución del OP, dificultando así una correspondencia válida entre ambos.

Una inconsistencia introducida por una escritura no-alineada, se evidencia con las operaciones de lectura. Si el PUA accede a una variable y obtiene un valor inconsistente, es decir, un valor que no podría ser producido por ninguna ejecución válida del OP, entonces se estaría transitando una trayectoria de ejecución que carece de contraparte en el programa oráculo. En consecuencia, el análisis de accesos a memoria, tanto de lectura como de escritura, es esencial en la segunda etapa del proceso.

3.2.2. Construcción del autómata monitor

En este apartado se describe en detalle el proceso de construcción del monitor. A partir de los insumos generados durante la etapa inicial, como los CFGs y la información sobre accesos a memoria, se desarrollan las siguientes fases del pipeline, que incluyen la

construcción de autómatas, su composición, la propagación de veredictos y la posterior minimización. Cada una de estas etapas se fundamenta en principios formales y persigue el objetivo de generar un monitor preciso, eficiente y correcto, bajo los supuestos presentados en la sección 2.1.3. Para ilustrar los resultados obtenidos en cada una de estas etapas, se utilizará el ejemplo presentado en la motivación como guía.

3.2.2.1 Etiquetado inicial

En primera instancia se debe realizar el etiquetado de los CFG. Para ello los mismos deben encontrarse en su versión dual, es decir que las aristas son las que están etiquetadas con instrucciones, y no los nodos. Luego se los procesa etiquetándolos según el siguiente criterio:

- Para las transiciones que representan nodos del CFG que están en σ (el matching), se eligen las instrucciones como etiquetas, por simplicidad.
- Se agregan etiquetas para las instrucciones condicionales para distinguir la rama verdadera (true) de la falsa (false).
- Para el PUA, se agregan etiquetas para cualquier instrucción condicional que no esté en σ .
- Cualquier transición que represente una instrucción que no esté en σ y que pueda escribir una variable que es leída por una instrucción en σ, también debe ser etiquetada. Estas últimas transiciones, destinadas a rastrear el flujo de datos, deben ser únicas para el autómata del OP y del PUA, por lo que se les agrega el sufijo OP o PUA.
- El resto de las aristas quedan sin etiquetar.

Este etiquetado permite posteriormente construir los autómatas de seguimiento de flujo de datos y realizar la composición en paralelo que sincroniza las etiquetas de transición compartidas entre los programas.

La figura 3.3 presenta los Grafos de Flujo de Control (CFGs) etiquetados correspondientes al Programa Oráculo (OP) y al Programa Bajo Análisis (PUA) respectivamente. Ambos diagramas ilustran la estructura de control de cada programa en su versión dual, donde las aristas están etiquetadas con las instrucciones correspondientes siguiendo los criterios de etiquetado establecidos en la metodología.

En el CFG del PUA (imagen derecha) se puede observar la presencia de la validación adicional `Cond0(P)` que genera una bifurcación temprana en el flujo, así como las instrucciones modificadas como `ChangedRComputation(P, A)`. El CFG del OP (imagen izquierda) muestra una estructura más simple sin la precondición inicial.

Las etiquetas en las aristas reflejan tanto las instrucciones comunes que forman parte del matching σ (como `R = InitialValueR()`, `A = AComputation(P)`, y `R = HeavyComputation(P, A)`), como las instrucciones específicas de cada programa marcadas con los sufijos “PUA” y “OP” para el rastreo del flujo de datos. Las bifurcaciones condicionales están claramente diferenciadas con las etiquetas `:then` y `:else`, permitiendo distinguir las diferentes trayectorias de ejecución posibles en cada programa.

Esta representación dual constituye la base fundamental para la posterior construcción de los autómatas de seguimiento de flujo de datos y la composición en paralelo que permitirá verificar la existencia de ejecuciones σ-alineadas entre ambos programas.

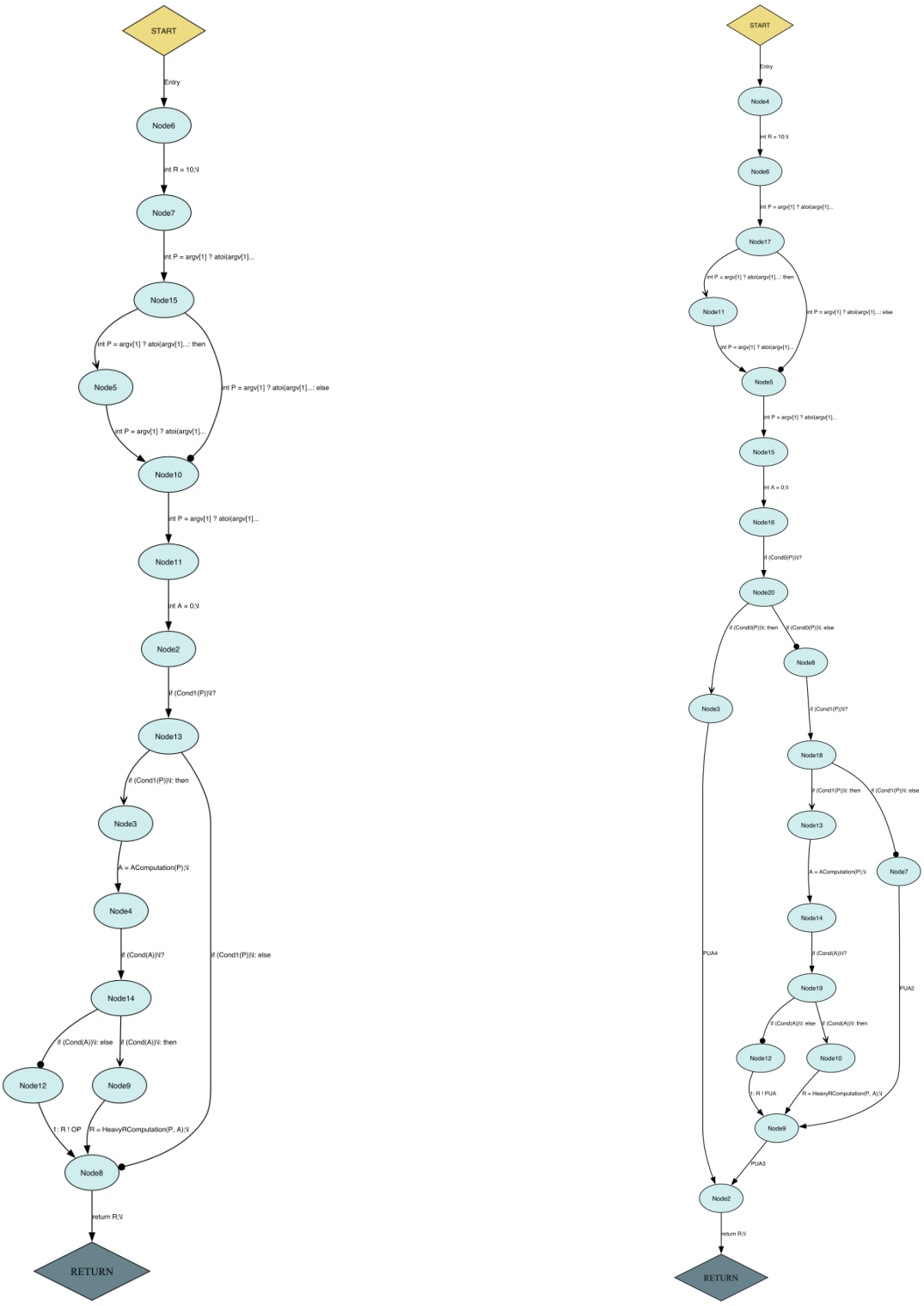


Fig. 3.3: CFGs etiquetados del Programa Oráculo (OP) y el Programa Bajo Análisis (PUA).

3.2.2.2 Creación de DFTAs

En esta etapa del proceso se busca generar los objetos encargados de registrar qué operaciones estarían permitidas en ejecuciones alineadas y cuáles no. La idea fundamental es trackear en qué estado se encuentra cada variable luego de que se ejecute cada operación, determinando si la misma mantiene la condición de σ-alineada o no.

Los Autómatas de Seguimiento de Flujo de Datos (Data-Flow Tracking Automata, DFTAs) constituyen un mecanismo formal para monitorear el estado de sincronización de las variables entre OP y el PUA. Estos autómatas modelan el impacto que tienen las escrituras no sincronizadas sobre la validez de las ejecuciones alineadas. Se definen dos DFTAs para cada variable v que puede ser escrita en un nodo del CFG que no pertenece a σ y que puede ser leída en un nodo que sí pertenece a σ, uno para el OP y otro para el PUA.

Estructura del DFTA

Cada DFTA es un autómata de dos estados que modela el estado de sincronización de la variable:

- **Estado 0 (Sincronizado)**: La variable tiene el mismo valor en ambos programas, permitiendo la ejecución de fragmentos comunes que la utilicen.
- **Estado 1 (Desincronizado)**: La variable ha sido modificada por código no común, por lo que puede tener valores diferentes en el OP y el PUA, inhibiendo la ejecución de fragmentos comunes que la lean.

Transiciones del DFTA

Las transiciones entre estados se definen según las siguientes reglas:

- **Transición** $0 \to 1$: Ocurre cuando se ejecuta una instrucción $stmt \notin \sigma$ que escribe la variable v. Esta transición se etiqueta con `stmt!OP` o `stmt!PUA` según corresponda. A su vez, estas instrucciones también agregan una transición reflexiva $1 \to 1$. Esto se debe a que sí es posible ejecutar una instrucción $\notin \sigma$ desde un estado desincronizado, solamente que el autómata permanecerá en ese estado.
- **Transición** $1 \to 0$: Ocurre cuando una instrucción $stmt \in \sigma$ escribe la variable v, re-sincronizando su valor entre ambos programas. De forma similar al primer caso, estas instrucciones también agregan una transición reflexiva $0 \to 0$. Esto situación es análoga a la presentada en el inciso anterior.
- **Habilitación de lecturas en estado 0**: Solo cuando el autómata se encuentra en el estado sincronizado (estado 0) se permite ejecutar instrucciones comunes que lean la variable v, garantizando que ambos programas accederán al mismo valor.
- **Inhibición de lecturas en estado 1**: Cuando el autómata se encuentra en el estado desincronizado, cualquier intento de ejecutar una instrucción común que lea la variable v es bloqueado, ya que los valores leídos podrían ser inconsistentes entre el OP y el PUA.

Semántica de las operaciones

El comportamiento del DFTA respecto a las operaciones de lectura y escritura se puede formalizar de la siguiente manera:

- **Escrituras no sincronizadas**: Si $stmt \notin \sigma$ y $v \in writes(stmt)$, entonces el estado del DFTA transiciona de 0 a 1.
- **Escrituras sincronizadas**: Si $stmt \in \sigma$ y $v \in writes(stmt)$, entonces el estado del DFTA se establece en 0.
- **Lecturas permitidas**: Si $stmt \in \sigma$ y $v \in reads(stmt)$, la operación solo es válida si el DFTA está en estado 0.

Esta aproximación garantiza que las ejecuciones σ-alineadas solo procedan cuando las variables involucradas en los fragmentos comunes mantienen valores consistentes entre ambos programas, preservando así la validez semántica de la alineación.

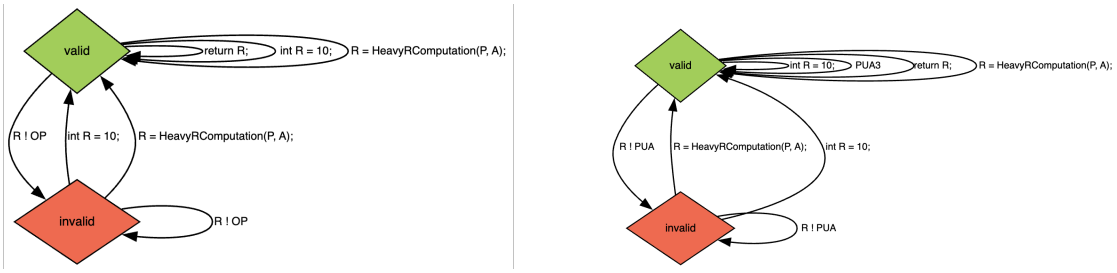


Fig. 3.4: DFTAs para la variable R, para el OP y el PUA respectivamente.

En la figura 3.4 se presentan los DFTAs correspondientes a la variable R para el Programa Oráculo (OP) y Programa Bajo Análisis (PUA) respectivamente.

Las transiciones sincronizadas (`int R = 10;`, `R = HeavyRComputation(P, A);`, y las evaluaciones condicionales) son idénticas en ambos DFTAs. La única diferencia radica en la transición de desincronización: `R ! OP` para el programa oráculo y `R ! PUA` para el programa bajo análisis, las cuales representan escrituras específicas de cada programa que invalidan la sincronización de la variable.

3.2.2.3 Composición Paralela

Una vez computados los autómatas principales (CFGs y DFTAs), se procede con la composición paralela de autómatas. Utilizando la definición del capítulo 2, esta etapa tiene como objetivo crear un autómata producto que capture todas las posibles interacciones entre el OP, el PUA y los autómatas de seguimiento de flujo de datos.

Proceso de Composición

La composición paralela se realiza sincronizando todos los autómatas a través de las etiquetas de transición compartidas. El resultado es un autómata producto donde cada estado representa una tupla que contiene:

- El estado actual del CFG del OP
- El estado actual del CFG del PUA
- Los estados de todos los DFTAs correspondientes a las variables rastreadas

Formalmente, si tenemos n variables rastreadas $v_1, v_2, \ldots, v_n$, el estado del autómata producto se representa como:

$$(q_{OP}, q_{PUA}, s_1^{OP}, s_1^{PUA}, s_2^{OP}, s_2^{PUA}, \ldots, s_n^{OP}, s_n^{PUA})$$

donde q_{OP} y q_{PUA} son los estados de los CFGs respectivos, y s_i^{OP}, s_i^{PUA} son los estados de los DFTAs para la variable v_i.

Sincronización de Transiciones

Las transiciones en el autómata producto están determinadas por la coordinación entre los subestados que componen cada nodo del autómata compuesto. Una transición etiquetada α aparece en la composición entre dos nodos si y solo si los subestados correspondientes pueden coordinarse para ejecutar dicha transición:

- **Transiciones sincronizadas**: Una transición $\alpha \in \sigma$ existe entre dos nodos compuestos $(q_{OP}, q_{PUA}, \vec{s})$ y $(q'_{OP}, q'_{PUA}, \vec{s'})$ si:
  - Existe $q_{OP} \xrightarrow{\alpha} q'_{OP}$ en el CFG del OP
  - Existe $q_{PUA} \xrightarrow{\alpha} q'_{PUA}$ en el CFG del PUA
  - Todos los DFTAs de variables leídas por α están en estado válido
  - Los DFTAs se actualizan según corresponda: $\vec{s} \xrightarrow{\alpha} \vec{s'}$
- **Transiciones exclusivas del PUA**: Una transición α específica del PUA existe si:
  - Existe $q_{PUA} \xrightarrow{\alpha} q'_{PUA}$ en el CFG del PUA
  - El estado del OP permanece inalterado: $q_{OP} = q'_{OP}$
  - Los DFTAs se actualizan según la semántica de α
- **Transiciones exclusivas del OP**: Una transición α específica del OP existe si:
  - Existe $q_{OP} \xrightarrow{\alpha} q'_{OP}$ en el CFG del OP
  - El estado del PUA permanece inalterado: $q_{PUA} = q'_{PUA}$
  - Los DFTAs se actualizan según la semántica de α

Esta coordinación entre subestados garantiza que el autómata producto capture fielmente todas las interacciones posibles respetando las restricciones de sincronización y flujo de datos.

La figura anterior ilustra el resultado de la composición paralela entre los CFGs del OP y el PUA junto con los DFTAs correspondientes. Cada nodo del autómata compuesto está etiquetado con una tupla que representa el estado conjunto (q_{OP}, q_{PUA}, s_R), donde los dos primeros elementos corresponden a los estados actuales de los grafos de flujo de control y el último elemento indica el estado de los DFTAs para la variable R.

Se puede observar cómo el autómata captura las diferentes trayectorias de ejecución posibles, incluyendo las bifurcaciones condicionales y las transiciones que afectan la sincronización de variables. Las etiquetas en las aristas reflejan tanto las operaciones sincronizadas (comunes a ambos programas) como las operaciones específicas de cada programa, marcadas con los sufijos correspondientes.

Esta representación compuesta constituye la base para la siguiente etapa del proceso, donde se asignarán y propagarán los veredictos que determinarán si existe una ejecución del OP que pueda ser σ-alineada con una ejecución dada del PUA.

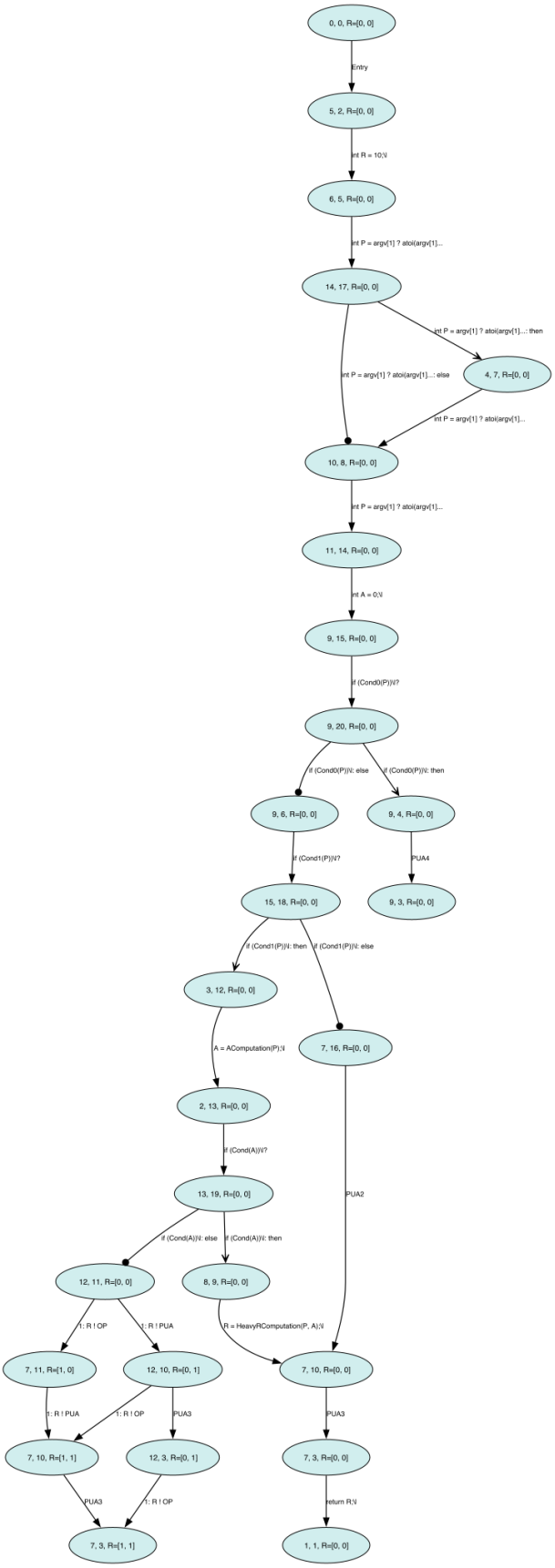


Fig. 3.5: Composición paralela de los autómatas de control de flujo y tracking de variables.

3.2.2.4 Veredictos

Una vez realizada la composición paralela, que constituye el paso fundamental del proceso de construcción, resta interpretar los nodos compuestos obtenidos para determinar qué veredicto les corresponde de acuerdo a la existencia o no de una contraparte en el programa que se busca imitar. Esta etapa es crucial, ya que establece las bases para que el monitor pueda emitir juicios sobre la viabilidad de las ejecuciones σ-alineadas. A continuación se detallarán los distintos pasos de esta fase. Se puede observar la totalidad del proceso en el algoritmo 3.

El proceso de asignación de veredictos se divide en tres fases secuenciales que refinan progresivamente la información contenida en el autómata compuesto:

1. Etiquetado de veredictos

En esta fase inicial se realiza un marcado directo de los nodos del autómata compuesto basándose en criterios estructurales. El resultado de aplicar este paso, se observa la figura 3.7(a). Los veredictos se asignan según las siguientes reglas:

- **NV (Not Yet Verified)**: Se asigna a todos los nodos que tienen al menos una transición saliente, indicando que hasta ese punto existe una ejecución del OP que puede imitar la trayectoria del PUA.
- **IV (Impossible to Verify)**: Se asigna a los nodos deadlock donde los estados del PUA y OP no han terminado simultáneamente, indicando que no existe contraparte válida en el OP.
- **V (Verified)**: Se asigna a los nodos deadlock donde tanto el PUA como el OP han alcanzado sus estados finales simultáneamente, indicando éxito completo en la alineación.

2. Propagación de veredictos

Esta fase utiliza algoritmos de punto fijo para propagar los veredictos IV y V a través del grafo, como se observa en la figura 3.7(b):

- **Propagación IV**: Utilizando un punto fijo mínimo, se marcan como IV todos los nodos cuyos sucesores estén marcados como IV, o que tengan un sucesor IV alcanzable mediante transiciones no comunes (correspondientes a movimientos no observables del OP o del PUA).
- **Propagación V**: Mediante un punto fijo máximo, se comienza con el conjunto de todos los nodos no-IV y se eliminan aquellos que tienen sucesores fuera del conjunto, sucesivamente, hasta alcanzar estabilidad. Los nodos remanentes se marcan como V.

3. Poda temprana

Una de las contribuciones al algoritmo propuesto para construir los MM fue la inclusión de este paso intermedio antes de la fase de compactación. Durante el proceso de propagación de veredictos es posible que un nodo marcado como IV mantenga nodos hijos con veredictos no-IV, lo cual constituye una inconsistencia semántica en el contexto de nuestro análisis.

Esta situación puede ocurrir cuando una operación del OP que no pertenece a σ permite rescatar la ejecución desde un estado IV, creando un camino hacia un veredicto más

Algorithm 3 Algoritmo Principal de Asignación de Veredictos

```
1: procedure VerdictAutomataMain(composedAutomata)
2:     automata ← composedAutomata
3:
4:     // Etiquetado inicial de veredictos
5:     labeledAutomata ← labelWithVerdicts(automata)
6:
7:     // Propagación de veredictos por punto fijo
8:     propagatedAutomata ← verdictPropagation(labeledAutomata)     ▷ Propagar IV (punto fijo mínimo) y V (punto fijo máximo)
9:
10:    // Poda temprana de inconsistencias
11:    initialNode ← findInitialNode(propagatedAutomata)
12:    prunedAutomata ← prune(propagatedAutomata, initialNode)  ▷ Eliminar nodos IV con sucesores no-IV por transiciones no observables
13:
14:    // Compactación final
15:    finalAutomata ← compactFinalStates(prunedAutomata)   ▷ Fusionar todos los nodos V en uno, todos los IV en otro
16:    return finalAutomata
17: end procedure
18:
19: procedure VerdictPropagation(automata)
20:    ivAutomata ← IVPropagation(automata)   ▷ Punto fijo mínimo para veredictos IV
21:    vAutomata ← VPropagation(ivAutomata)  ▷ Punto fijo máximo para veredictos V
22:    return vAutomata
23: end procedure
24:
25: // Etiquetado inicial según criterios estructurales
26: procedure LabelWithVerdicts(automata)
27:    newAutomata ← copy(automata)
28:    for all node ∈ newAutomata.nodes do
29:        if node.children ≠ ∅ then
30:            node.verdict ← ""
31:            node.prefixVerdict ← "NV"              ▷ Nodo con transiciones salientes
32:        else
33:            if ¬node.isCompExit() then
34:                node.verdict ← "IV"              ▷ Deadlock sin terminación simultánea
35:            else
36:                node.verdict ← "V"               ▷ Deadlock con terminación simultánea
37:            end if
38:        end if
39:    end for
40:    return newAutomata
41: end procedure
```

favorable. Sin embargo, esto no es de interés para nuestro análisis, precisamente porque se trata de comportamientos que no pueden ser observados desde la perspectiva del PUA[1].

En la figura 3.6 se presenta un recorte del autómata monitor para uno de los casos del benchmark. Se presenta este otro caso para ilustrar la utilidad de la poda, ya que esta patología no se presenta en el ejemplo de motivación dada su simplicidad. El autómata del cual se tomó este recorte es el producto de aplicar los veredictos y propagarlos, sin efectuar la poda. En él se puede observar como nodos IV, que deberían ser terminales, tienen transiciones salientes. Lo que no es coherente con el significado que se le da a dichos nodos. En la figura 3.7(c), se observa el resultado final de esta etapa.

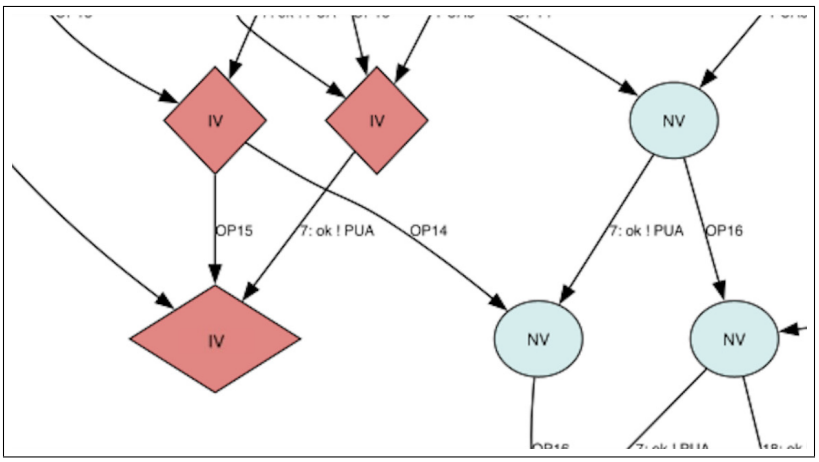


Fig. 3.6: Ejemplo de inconsistencia semántica: un nodo marcado como IV (izquierda) presenta una transición saliente OP14 hacia un nodo NV, violando la coherencia esperada en la asignación de veredictos. Este ejemplo es tomado del benchmark, en particular la CoreUtil `cat`.

Algoritmo de poda implementado

El algoritmo realiza una exploración en profundidad desde el nodo inicial, aplicando las siguientes reglas de retención:

- **Nodos terminales**: Los nodos con veredictos definitivos V o IV se mantienen pero no se exploran sus sucesores, actuando como puntos de corte en la exploración.
- **Nodos intermedios**: Solo los nodos con veredictos no terminales (NV) propagan la exploración hacia sus sucesores, manteniendo únicamente los caminos que conducen a estados observables.
- **Aristas remanentes**: Se preservan únicamente las aristas que conectan nodos alcanzables a través de trayectorias semánticamente válidas, eliminando automáticamente las conexiones hacia estados rescatados por operaciones no observables.

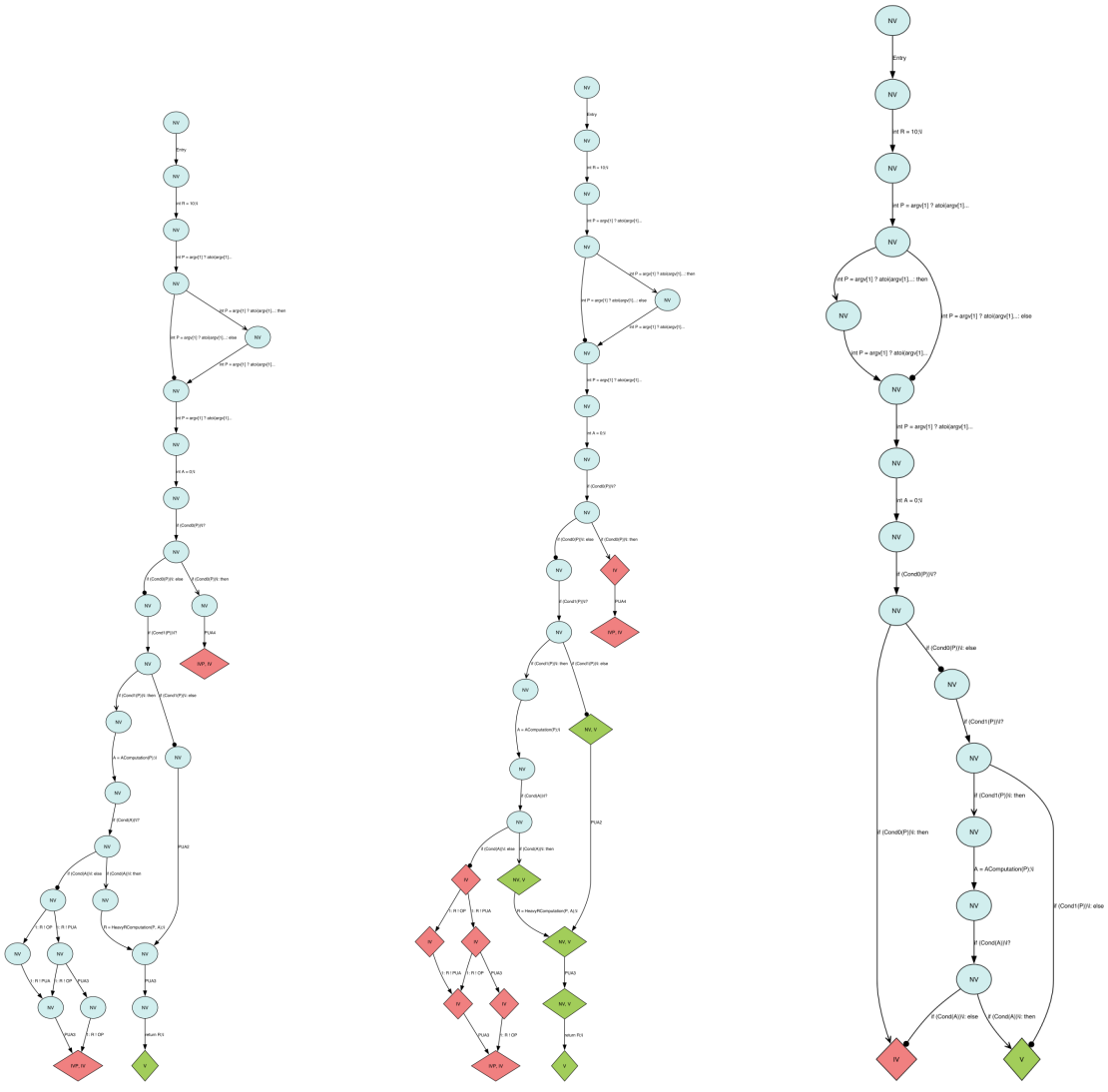
Resultado de la poda

[1] Si el OP tuviese una condición donde una de las ramas sincroniza el valor de una variable y la otra no, podría darse esta situación. Aunque en el PUA el valor aparezca sincronizado, no hay forma de determinar qué está ocurriendo internamente en el OP. Por tanto, las transiciones sin etiquetar que lleven de IV a NV carecen de sentido desde la perspectiva del análisis, ya que el veredicto IV se asignó correctamente: a partir de ese punto, el alineamiento entre dos ejecuciones depende únicamente de decisiones internas del OP que son inobservables, dado que este nunca se ejecuta.

Este proceso garantiza que el autómata resultante contenga únicamente nodos y transiciones que representan ejecuciones genuinamente alcanzables y verificables desde la perspectiva del programa bajo análisis. La poda elimina efectivamente el ruido que podría surgir de comportamientos internos del programa oráculo que no tienen correspondencia observable en el PUA, asegurando la coherencia semántica del monitor final.

Compactación de veredictos En la fase final se simplifica la representación del autómata mediante la fusión de nodos con veredictos equivalentes:

- Todos los nodos marcados como V se fusionan en un único nodo representativo.
- Todos los nodos marcados como IV se fusionan en un único nodo representativo.



(a) Etiquetado (b) Propagación (c) Etiquetados

Fig. 3.7: Etiquetado, Propagación y Compactación de veredictos.

3.2.2.5 Minimización

Si bien el monitor obtenido hasta el momento es funcional, esta última instancia se propone volverlo aún más compacto y eficiente. Para ello se aplica una serie de transformaciones que buscan reducir al máximo la complejidad del monitor manteniendo su corrección semántica. Las mismas se pueden observar en el algoritmo 4.

Algorithm 4 Algoritmo Principal de Minimización de Autómatas

```
1: procedure MinimizerMain(automata)
2:     // Inicialización
3:     initialNode ← findInitialNode(automata)
4:     hiddenAutomata ← hideLabels(automata)        ▷ Ocultar etiquetas no relevantes para bisimulación
5:
6:     // Primera pasada: Minimización por bisimulación
7:     closureAutomata ← computeClosure(hiddenAutomata)   ▷ Calcular clausura del autómata original
8:     minimizedAutomata ← minimize(closureAutomata)      ▷ Aplicar algoritmo de bisimulación
9:
10:    // Determinización
11:    closureMinAutomata ← computeClosure(minimizedAutomata)   ▷ Recalcular clausura del autómata minimizado
12:    deterministicAutomata ← determinize(closureMinAutomata)  ▷ Determinización
13:    compactedAutomata ← compactFinalStates(deterministicAutomata)  ▷ Primera compactación
14:
15:    // Segunda pasada de optimización post-determinización
16:    propagatedAutomata ← IVPropagation(compactedAutomata)  ▷ Re-propagación de veredictos IV
17:    newInitialNode ← findInitialNode(propagatedAutomata)
18:    prunedAutomata ← prune(propagatedAutomata, newInitialNode)   ▷ Poda de sucesores de IV
19:
20:    // Compactación final
21:    finalAutomata ← compactFinalStates(prunedAutomata)   ▷ Compactación final
22:    return finalAutomata
23: end procedure
```

El proceso de minimización se estructura en varias etapas secuenciales, cada una abordando aspectos específicos de la optimización:

1. Ocultamiento de etiquetas y cómputo de clausura

Como paso preparatorio, se ocultan todas las etiquetas de transición que no corresponden a instrucciones condicionales o sus ramas, marcándolas como transiciones no etiquetadas. Este proceso busca que el monitor final contenga únicamente transiciones asociadas

a instrucciones de control. Adicionalmente, se eliminan las etiquetas de las transiciones marcadas únicamente con los prefijos OP o PUA, dado que representan comportamientos que no requieren monitoreo: las transiciones OP por ser no observables (al no pertenecer a σ), y las transiciones PUA por carecer de relevancia analítica (no afectan el flujo de control del programa ni las variables de interés). Esta simplificación permite obtener un autómata más compacto al concluir el proceso de construcción. Finalmente, se calcula la clausura transitiva del autómata para capturar todas las transiciones implícitas que surgen a través de secuencias de movimientos no observables.

La clausura se construye mediante un algoritmo que:

- Establece transiciones reflexivas para todos los nodos
- Computa la clausura transitiva de las transiciones no etiquetadas usando Floyd-Warshall [5]
- Crea transiciones directas etiquetadas que incorporan secuencias de movimientos no observables

2. Minimización por bisimulación

Se implementó un algoritmo de minimización basado en bisimulación[4] que agrupa estados equivalentes en bloques. Previo al mismo, es necesario calcular la clausura transitiva del autómata actual. Esto es necesario, ya que a causa del paso inicial de esta fase (ocultamiento de etiqueta), se tendrán transiciones sin etiquetar, que deben considerarse en la minimización igualmente. El proceso utiliza refinamiento iterativo:

- **Partición inicial**: Los nodos se agrupan según sus veredictos, creando bloques de estados con comportamiento similar.
- **Refinamiento iterativo**: Se refinan los bloques dividiendo aquellos cuyos estados tienen diferentes *signatures* o firmas de transición , es decir, estados que transicionan a diferentes bloques bajo las mismas etiquetas.
- **Construcción del autómata minimizado**: Se crea un nuevo autómata donde cada bloque se convierte en un estado único, preservando la estructura de transiciones entre bloques.

3. Determinización

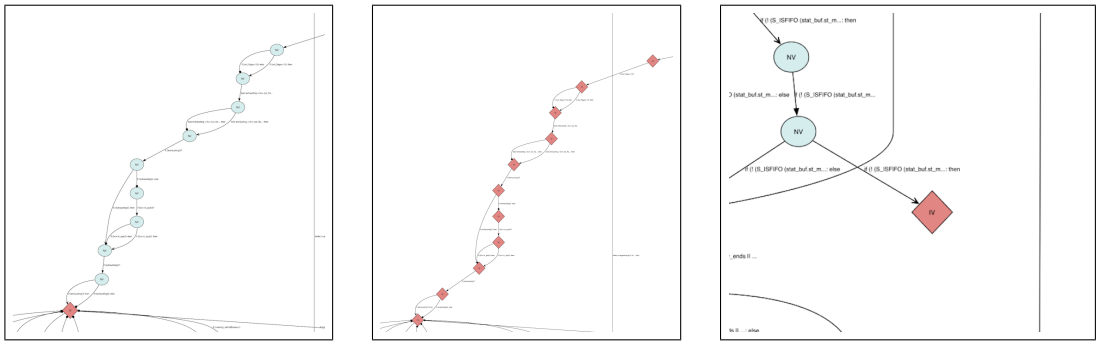
Dado que la presencia de transiciones no etiquetadas puede generar no-determinismo que entorpece el uso práctico del monitor, se aplica un proceso de determinización basado en construcción de subconjuntos. La clausura transitiva debe recomputarse, ya que el autómata que atravesó la minimización por bisimulación tiene otra clausura. Este proceso construye estados del autómata determinístico como conjuntos de estados del autómata original, A su vez, maneja las transiciones etiquetadas de manera estándar, agrupando todos los estados alcanzables bajo una misma etiqueta. Cabe destacar que utiliza un criterio de "pesimismo"para asignar veredictos a los nuevos estados: un conjunto solo puede ser V si todos sus estados componentes son V, priorizando IV sobre NV sobre V.

4. Segunda pasada de optimización

Una contribución importante al algoritmo general es la incorporación de una segunda ronda de optimizaciones post-determinización. Esto es necesario porque el proceso de determinización puede desestabilizar la estructura de veredictos del monitor, requiriendo:

- **Nueva compactación de veredictos finales**: Re-fusión de nodos V e IV que puedan haberse fragmentado durante la determinización.
- **Re-propagación de veredictos IV**: Aplicación del algoritmo de propagación para actualizar veredictos que puedan haber cambiado.
- **Poda adicional**: Eliminación de inconsistencias que puedan haber surgido, aplicando el mismo algoritmo de poda descrito anteriormente en el apartado 3.2.2.4.
- **Compactación final**: Una última fusión de estados terminales para obtener la representación más compacta posible.

Utilizando nuevamente un ejemplo del benchmark que será introducido más adelante, es posible ver el beneficio introducido por esta segunda propagación y sus pasos adjuntos. La figura 3.8 ilustra la importancia crítica de la segunda pasada de optimización en el algoritmo de minimización propuesto.



(a) Resultado tras minimización y determinización inicial (b) Efecto de la segunda propagación de veredictos (c) Resultado final tras compactación

Fig. 3.8: Efecto de la segunda pasada de optimización en el proceso de minimización.

En la figura 3.8(a) se observa el resultado inmediato tras la minimización por bisimulación y determinización, donde varios nodos intermediaros mantienen veredictos NV a pesar de que todas sus trayectorias conducen inevitablemente a estados IV. Esta inconsistencia surge porque el proceso de determinización puede desestabilizar la estructura de veredictos previamente establecida. La figura 3.8(b) muestra el efecto correctivo de la segunda propagación de veredictos IV, donde estos nodos intermediarios son apropiadamente reclasificados para reflejar su verdadera naturaleza semántica. Finalmente, la figura 3.8(c) presenta el resultado tras la compactación final, donde todos los nodos con veredictos equivalentes se fusionan, resultando en un monitor significativamente más compacto y semánticamente coherente.

Resultado de la minimización

El monitor resultante mantiene la corrección semántica del original pero con una representación significativamente más compacta y eficiente. La combinación de bisimulación, determinización y múltiples pasadas de optimización garantiza que el Mimicry Monitor final sea tanto preciso en sus veredictos como eficiente en su ejecución durante el monitoreo en tiempo real. El mismo se puede apreciar en la figura 3.9

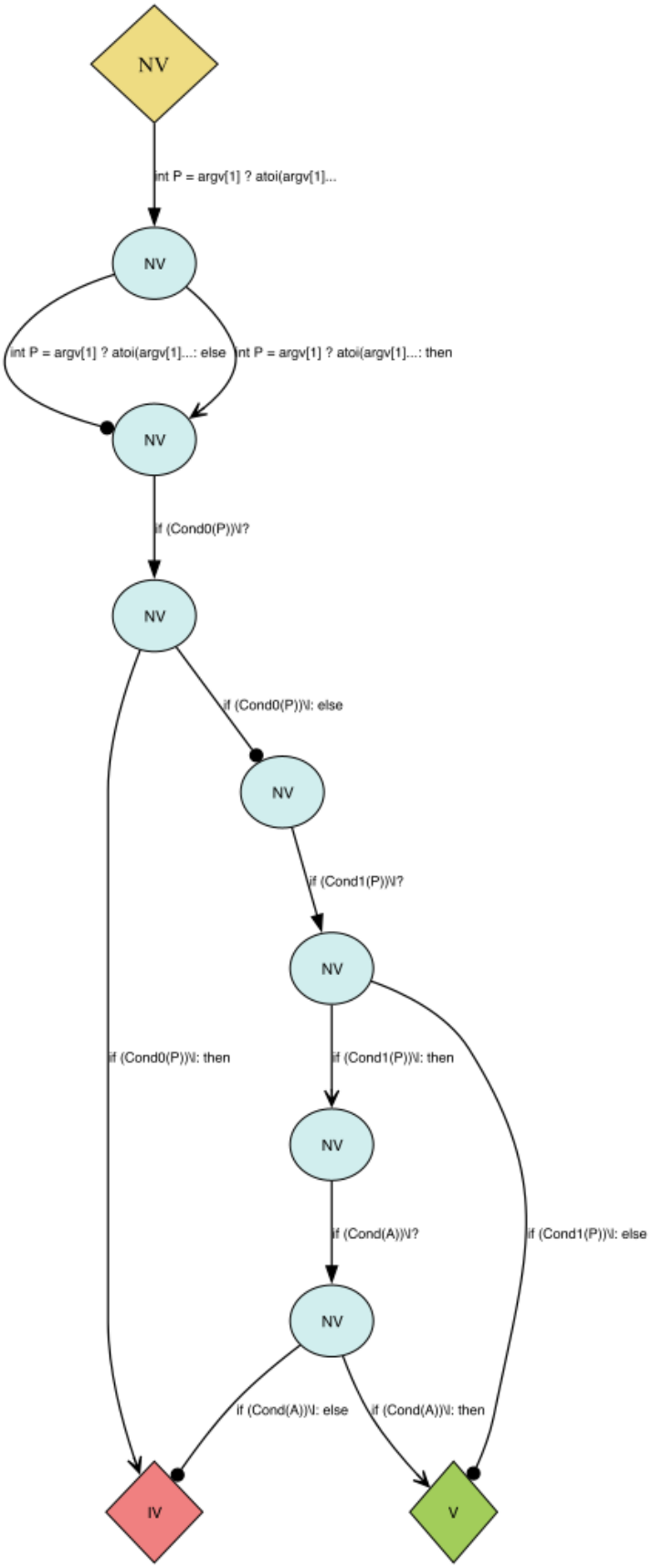


Fig. 3.9: Resultado final de la minimización.

3.2.3. Síntesis del proceso de construcción

El proceso de construcción del Mimicry Monitor presentado en esta sección constituye una metodología sistemática que transforma la información estructural y semántica de dos programas relacionados en un autómata compacto y eficiente capaz de verificar la existencia de ejecuciones σ-alineadas en tiempo de ejecución.

La arquitectura del proceso se caracteriza por su naturaleza modular e incremental. Cada etapa construye sobre los resultados de la anterior, refinando progresivamente la representación hasta obtener un monitor optimizado. La secuencia que va desde el análisis inicial de los programas, pasando por la construcción de autómatas especializados, su composición paralela, la asignación y propagación de veredictos, hasta llegar a la minimización final, garantiza que el monitor resultante capture fielmente las relaciones de alineación mientras mantiene una representación eficiente.

Las contribuciones metodológicas introducidas, particularmente la poda de inconsistencias y la segunda pasada de optimización post-determinización, abordan limitaciones prácticas identificadas durante la implementación del algoritmo original. Estas mejoras no solo optimizan el tamaño y eficiencia del monitor, sino que también garantizan la coherencia semántica de los veredictos emitidos.

El monitor final obtenido mediante este proceso constituye una estructura autónoma que encapsula el conocimiento necesario para determinar durante la ejecución del programa bajo análisis, si existe una ejecución correspondiente en el programa oráculo que pueda ser considerada σ-alineada. Esta capacidad de anticipación, lograda sin necesidad de ejecutar efectivamente el programa oráculo, representa el fundamento técnico que habilita la aplicación de verificación que se explorará en los capítulos siguientes.

La metodología presentada establece así las bases teóricas y prácticas para la instrumentación y uso efectivo de Mimicry Monitors en escenarios reales de desarrollo de software, proporcionando una herramienta versátil para tareas de verificación, testing de regresión y análisis diferencial de programas.

4. IMPLEMENTACIÓN

La materialización de los conceptos teóricos y metodológicos presentados en los capítulos anteriores requiere de decisiones de diseño e implementación que permitan traducir los algoritmos formales en herramientas de software funcionales y eficientes. En el capítulo anterior se definió el diseño general del proceso. Este capítulo, en cambio, describe en detalle las estrategias de implementación adoptadas para cada una de las etapas del proceso de construcción de Mimicry Monitors, así como las tecnologías y frameworks seleccionados para su desarrollo.

La implementación del sistema se estructura en torno a una arquitectura modular que refleja la naturaleza secuencial del proceso de construcción descrito en el Capítulo 3. Cada módulo corresponde a una etapa específica del pipeline de construcción, desde el análisis inicial de los programas de entrada hasta la generación del monitor minimizado final. Esta organización modular no solo facilita el desarrollo y mantenimiento del código, sino que también permite la evaluación independiente de cada componente y la adaptación del sistema a diferentes contextos de uso.

Las decisiones de implementación se guiaron por varios criterios fundamentales: la fidelidad a los algoritmos teóricos propuestos, la eficiencia computacional en el procesamiento de programas de tamaño real, la extensibilidad para incorporar futuras mejoras metodológicas, y la integración efectiva con herramientas existentes del ecosistema de análisis de programas. Estos criterios influyeron tanto en la selección de las tecnologías base como en las decisiones arquitectónicas específicas de cada módulo.

La implementación del primer y tercer módulo se basa principalmente en el framework LLVM [14] para el análisis de programas. Esta elección permite trabajar con programas escritos en múltiples lenguajes de programación de manera uniforme.

El desarrollo se complementa con herramientas especializadas para la manipulación y visualización de autómatas. La integración de estos componentes resulta en un sistema completo que automatiza todo el proceso de construcción de Mimicry Monitors.

A lo largo de este capítulo se detallan las características específicas de cada módulo de implementación, los desafíos técnicos encontrados y las soluciones desarrolladas para superarlos.

El proyecto está implementado combinando componentes en Java y LLVM. Utiliza un *pass* especializado de LLVM para extraer los gráficos de flujo de control (CFGs) y otro para rastrear operaciones de lectura y escritura sobre variables en los dos programas escritos en C (OP y PUA). En Java, se construye un autómata de monitoreo a partir de los insumos generados en la primera instancia, y de la correspondencia σ. Este autómata es luego utilizado por un pass de LLVM para instrumentar el PUA e insertar lógica de verificación en tiempo de ejecución. El sistema requiere LLVM 19.1.7, Clang, Java JDK 22, Maven, GraphViz, Bash y el generador Ninja. Para ejecutar el flujo completo de manera automática, se utiliza el script `run-mimicry.sh`, que analiza, construye e instrumenta el programa. Alternativamente, los pasos pueden ejecutarse manualmente mediante los scripts `analyze.sh` (para el análisis inicial) e `instrument.sh` (para la instrumentación). Un ejemplo completo se puede ejecutar directamente con `./run-mimicry.sh`, utilizando los archivos de demostración ya preparados en el repositorio, al cual se puede acceder mediante este link: https://git.exactas.uba.ar/fegarcia/mimicrymonitor/. El re-

positorio cuenta con un README que da más detalles sobre el proyecto y su estructura.

4.1. Introducción a LLVM

Como se anticipó en el apartado anterior, para llevar a cabo las etapas 1 y 3 del proceso fue necesario contar con una infraestructura capaz de analizar y modificar programas a un nivel suficientemente bajo. Esta necesidad surge por dos motivos principales, estrechamente ligados a la naturaleza del monitoreo que se busca implementar.

En primer lugar, durante la etapa de procesamiento de los programas *input*, es fundamental realizar un análisis detallado del comportamiento de cada uno de ellos. Específicamente, es necesario identificar qué instrucciones acceden a qué variables, distinguiendo entre lecturas y escrituras. Esta información resulta crucial para la construcción posterior del autómata monitor. Además, se requiere extraer el grafo de flujo de control (CFG) de cada programa, tanto del programa oráculo como del programa bajo análisis, dado que sobre estos grafos se construye la composición que finalmente dará lugar al modelo formal del monitor.

En segundo lugar, durante la etapa de instrumentación, es necesario modificar el código del PUA para insertar llamadas al monitor en puntos específicos de su ejecución. Estas llamadas permiten observar, en tiempo de ejecución, qué decisiones toma el programa, y así determinar qué veredicto se obtiene. Para ello, se requiere una herramienta que no solo permita inspeccionar el programa, sino también transformarlo con precisión.

En otras palabras, era indispensable contar con un entorno que ofreciera soporte tanto para el análisis estático detallado de los programas como para su transformación. En este contexto, se optó por utilizar LLVM como infraestructura base, debido a su flexibilidad, representación intermedia rica (LLVM IR), y amplio soporte para análisis y transformación de código. Esta herramienta resultó clave para la implementación eficiente y precisa del proceso de monitoreo.

Si bien el *tool* actual solo trabaja con programas en C/C++, utilizar LLVM y una separación modular del proyecto, permitiría en un futuro poder dar soporte a otros lenguajes que LLVM soporte, sin un gran esfuerzo adicional.

4.1.1. LLVM como infraestructura de compilación

LLVM es una infraestructura de compilación modular y reutilizable diseñada como un *framework* para el análisis y transformación de programas a lo largo de todo su ciclo de vida [13]. Su arquitectura se fundamenta en una representación intermedia universal que permite optimizaciones transparentes desde el tiempo de compilación hasta la ejecución, como se puede observar en la figura 4.1.

4.1.1.1 Representación Intermedia

El núcleo de LLVM es su Representación Intermedia (LLVM IR), que actúa como una interfaz universal entre diferentes frontends y backends. LLVM IR está basada en la forma SSA (Static Single Assignment) [3], donde cada variable se asigna exactamente una vez y cada uso de una variable es alcanzado por exactamente una definición.

La representación intermedia presenta características fundamentales que la distinguen de otras infraestructuras de compilación. Mantiene información de tipos explícita durante

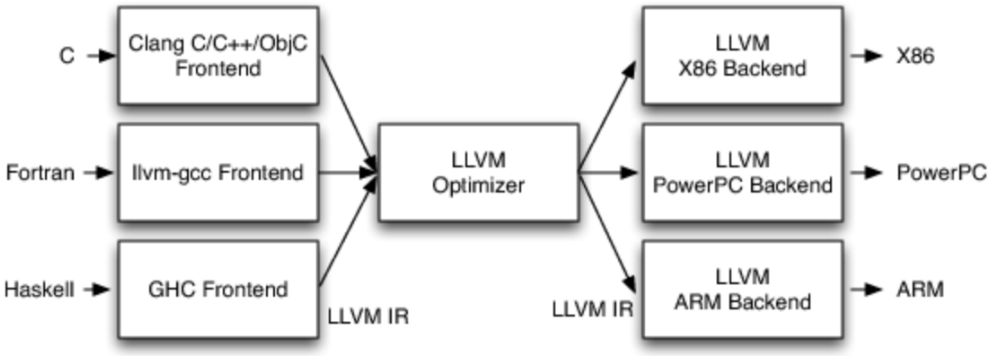


Fig. 4.1: Esquema del framework de LLVM [13]

toda la compilación, utiliza un formato de código de tres direcciones con flujo de control explícito, y opera sobre registros virtuales en cantidades ilimitadas. Esta estructura jerárquica organiza el código en módulos, funciones y bloques básicos, exponiendo la estructura del programa de manera explícita mientras mantiene la flexibilidad para representar patrones complejos de flujo de control.

El sistema de tipos de LLVM soporta tanto tipos primitivos como agregados con semántica independiente del lenguaje. Incluye aritmética de punteros tipada a través de la instrucción `getelementptr`, que calcula direcciones sin realizar operaciones de memoria, habilitando análisis de alias precisos y optimizaciones de estructuras de datos complejas.

4.1.1.2 Framework de Passes

El Framework de passes de LLVM proporciona la infraestructura fundamental para organizar las transformaciones del compilador en unidades discretas y componibles. Implementa dos sistemas de gestión de passes: el moderno New Pass Manager (PM), que es utilizado en este trabajo, para pipelines de optimización y el Legacy Pass Manager, que constituye la versión anterior del PM.

El framework organiza los passes en una estructura jerárquica basada en la granularidad del IR que procesan. Los Module passes operan sobre unidades de compilación completas, los Function passes se ejecutan independientemente en cada función, los Loop passes procesan cada anidamiento de ciclo de forma independiente, y los Basic Block passes se restringen a transformaciones locales dentro de bloques básicos individuales.

Esta jerarquía habilita tanto perspectivas de optimización globales como locales mientras mantiene límites claros entre los alcances de transformación. El sistema de gestión de análisis maneja automáticamente las dependencias entre passes, asegurando que los análisis requeridos se computen antes que los passes de transformación que los necesitan.

4.1.1.3 Pipeline de Optimización

El pipeline de optimización implementa una estrategia de procesamiento multinivel que transforma sistemáticamente el código desde representaciones de alto nivel hasta código máquina optimizado. Sigue un modelo de ejecución jerárquico que refleja la jerarquía de tipos de passes.

El PassBuilder del New Pass Manager sirve como componente central para construir pipelines de optimización, creando secuencias de optimización estándar adaptadas a diferentes niveles de optimización. La organización basada en fases estructura el pipeline en simplificación temprana, optimización central y optimización tardía.

El sistema implementa cacheo de resultados de análisis e invalidación selectiva como optimizaciones críticas de rendimiento. Cuando los passes consultan resultados de análisis, el gestor verifica la validez del caché y recomputa solo cuando es necesario, minimizando la presión sobre la memoria y mejorando la localidad del caché.

La flexibilidad de LLVM permitió integrar los passes desarrollados específicamente para este proyecto, con la infraestructura existente, resultando en una implementación robusta y eficiente que aprovecha décadas de desarrollo en tecnologías de compilación.

4.2. Desarrollo de nuevos passes

Por las características presentadas en la sección anterior, y considerando las necesidades específicas del análisis requerido para la construcción de Mimicry Monitors, se decidió utilizar LLVM como herramienta principal de análisis y transformación de los programas de entrada. Esta decisión requirió tanto el uso de passes preexistentes como el desarrollo de passes especializados para cumplir con los objetivos del proyecto.

4.2.1. Passes preexistentes

En la primera etapa del proceso, correspondiente al análisis de los programas de entrada, era necesario obtener información precisa sobre el flujo de control de los programas. Para ello se utilizó el pass existente `view-cfg`, que genera un grafo de flujo de control en formato DOT del programa en representación intermedia.

Esta aproximación presentó un desafío significativo, ya que la función de σ-alineamiento está definida sobre nodos del código fuente original, mientras que el CFG obtenido opera sobre la representación intermedia de LLVM. Para resolver esta discrepancia, fue necesario desarrollar un proceso de postprocesamiento que transformara el CFG obtenido de la IR en un CFG con nodos que mapeen directamente a líneas del código fuente original, y posteriormente convertirlo a su versión dual como requiere la metodología.

Este postprocesamiento de los CFGs fue implementado fuera del entorno de LLVM utilizando Java, al igual que la construcción del monitor automata. Esta separación permitió mayor flexibilidad en el manejo de las estructuras de datos complejas requeridas para la construcción del autómata final.

4.2.2. Passes desarrollados

Si bien LLVM cuenta con una gran variedad de passes de análisis y transformación ya existentes, ninguno se adaptaba exactamente a las necesidades del proyecto. Por esta razón, se desarrollaron dos passes nuevos.

4.2.2.1 Pass Def-Use (FeliDU)

Una vez obtenidos los CFGs, como se anticipó en secciones anteriores, se requería conocer qué operaciones realizan las distintas instrucciones sobre las variables del programa. Para ello se diseñó e implementó un pass especializado denominado `FeliDU` (por Def-Use),

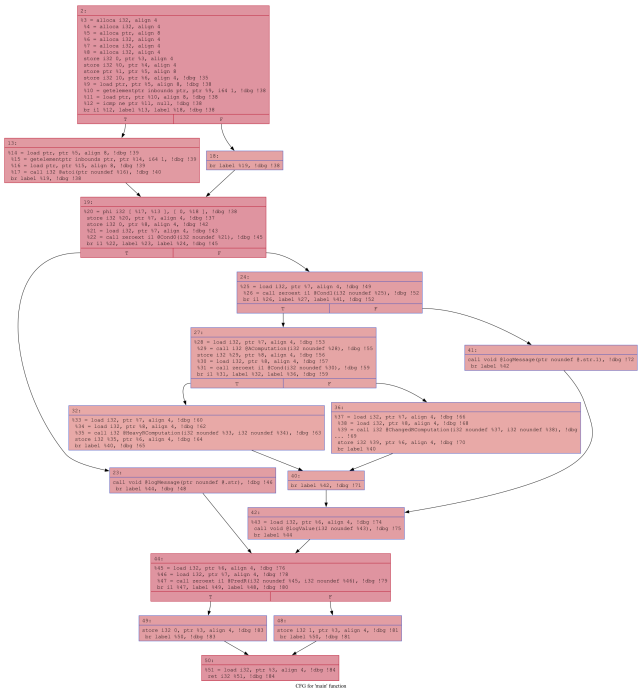


Fig. 4.2: CFG generado por LLVM para el PUA. Se puede observar que cada nodo es un Basic Block, y están conformados por las instrucciones del IR del PUA.

cuya estructura y funcionamiento se detallan a continuación.

Estructura del pass:

El pass `FeliDU` hereda de la clase Function Pass del nuevo sistema de gestión de passes de LLVM, implementando la interfaz requerida a través del método `run(Function &F, FunctionAnalysisManager &AM)`. La arquitectura del pass se organiza en torno a tres componentes principales:

- **Análisis de instrucciones:** La función `getReadWriteVars` examina cada instrucción para determinar qué variables lee y cuáles modifica. Esta función maneja específicamente:
  - `StoreInst`: Identifica escrituras a memoria y lecturas de valores puntero
  - `LoadInst`: Detecta lecturas desde ubicaciones de memoria
  - `CallInst`: Analiza argumentos de funciones considerando parámetros como potenciales lecturas y escrituras
  - Otras instrucciones: Examina operandos que sean punteros para identificar accesos implícitos
- **Resolución de nombres de variables:** La función `getVariableName` utiliza información de depuración disponible en el IR para mapear valores de LLVM a nombres de variables del código fuente original. Esta función verifica si el valor tiene un nombre explícito asignado y luego examina los registros de depuración asociados (`DbgVariableRecord`) para encontrar correspondencias. Por último, maneja casos especiales donde la información de depuración puede estar ausente

- **Análisis de alias:** Integra el sistema de análisis de alias de LLVM para determinar qué variables pueden ser afectadas indirectamente por operaciones de escritura, reportando alias potenciales como escrituras adicionales.

Salida del pass:
El pass genera un reporte estructurado que incluye, para cada instrucción analizada, la identificación del bloque básico y número de línea correspondiente, junto con las listas de variables leídas marcadas como `R` y variables escritas marcadas como `W`. Adicionalmente, el reporte incorpora información sobre variables afectadas por alias, las cuales se marcan como `W (via alias)` para indicar que pueden ser modificadas indirectamente a través de operaciones de escritura sobre otras variables que comparten la misma ubicación de memoria.

4.2.2.2 Pass de Instrumentación (MimicryInstrument)

El segundo pass desarrollado fue el de instrumentación, cuyo objetivo es utilizar el autómata monitor para realizar llamadas de monitoreo después de cada evento que modifique el flujo de control del programa bajo análisis.

Estructura del pass:
El pass `MimicryInstrument` implementa una arquitectura compleja que combina análisis estático y generación de código:

- **Instrumentación de flujo de control**: El componente principal del pass modifica el CFG insertando llamadas de monitoreo:
  - Identifica instrucciones de terminación con múltiples sucesores (branches y switches)
  - Crea bloques *trampoline* que interceden entre la instrucción de control y sus destinos
  - Inserta llamadas a `monitorAction` con etiquetas apropiadas ('then', 'else', 'case_N', 'default')
  - Actualiza nodos PHI para mantener la semántica SSA del programa
- **Componente de runtime:** El sistema se complementa con un componente de runtime implementado en C (`monitorAction.c`) que mantiene el estado actual del autómata durante la ejecución del programa instrumentado. Este componente procesa las transiciones de acuerdo a las etiquetas recibidas desde el programa instrumentado. Adicionalmente, el componente genera logs detallados del proceso de monitoreo que incluyen información sobre transiciones ejecutadas, cambios de estado y veredictos alcanzados. A su vez, maneja apropiadamente casos de error y estados terminales del autómata para garantizar la robustez del sistema de monitoreo.

Salida del Pass Como output de aplicar este pass a un programa, se obtiene una versión instrumentada del mismo. Esta monitorea su comportamiento de acuerdo al autómata provisto, y emite los veredictos correspondientes a cada punto del programa.

4.2.2.3 Desafíos de implementación:

El modelo SSA (Static Single Assignment) de LLVM agregó una capa extra de complejidad al momento de implementar el pass `FeliDU`. Fue especialmente desafiante identificar con claridad qué operaciones se aplicaban sobre qué variables, ya que estas aparecen con nombres temporales y versiones múltiples en la IR. El análisis requería inferir con precisión, a partir de valores SSA, cuáles eran las variables del código fuente original involucradas, y en qué líneas del programa se encontraban las operaciones correspondientes. Esta tarea exigió una comprensión profunda del sistema de información de debug de LLVM, para poder reconstruir la semántica original del programa desde su representación intermedia.

Por el otro lado, el desarrollo de `MimicryInstrument` presentó varios desafíos técnicos significativos. Uno de los principales fue la preservación de la semántica original del programa: resultaba crucial asegurar que las llamadas al sistema de monitoreo no alteraran su comportamiento, lo cual implicaba un manejo cuidadoso de los valores de retorno y de los posibles efectos laterales. Otro aspecto crítico fue la gestión de los nodos PHI. La inserción de bloques trampoline requería modificar adecuadamente estos nodos para mantener la forma SSA del programa, evitando referencias inválidas a bloques básicos. Asimismo, fue necesario definir con precisión el nivel de instrumentación, es decir, determinar el punto exacto dentro de la representación intermedia (LLVM IR) donde debían insertarse las llamadas de monitoreo, teniendo en cuenta que todas las transformaciones operan exclusivamente sobre esta capa.

A esto se sumó la integración con el entorno de ejecución, que implicó coordinar la generación de código instrumentado con las funciones de runtime implementadas en C, encargadas de procesar la lógica del monitor en tiempo de ejecución.

Finalmente, un desafío importante fue el procesamiento del autómata de monitoreo. Como este se genera a partir de un archivo en formato DOT, fue necesario implementar un procedimiento de parseo que permitiera identificar, en cada momento de la ejecución, el nodo actual dentro del autómata y el veredicto correspondiente al estado alcanzado por el programa.

5. EVALUACIÓN

La validación de la metodología propuesta para la construcción y uso de Mimicry Monitors requiere una evaluación que demuestre tanto la viabilidad técnica como la efectividad práctica del enfoque desarrollado en escenarios reales de verificación. Este capítulo presenta los resultados de una serie de experimentos diseñados para evaluar el comportamiento del sistema implementado utilizando programas reales y sus conjuntos de pruebas asociados.

La evaluación se centra en el análisis del comportamiento de Mimicry Monitors aplicados a diferentes herramientas de las GNU Core Utilities, aprovechando los conjuntos de pruebas existentes que estas utilidades incorporan. Esta aproximación experimental permite evaluar cómo el sistema responde ante casos de prueba reales que han sido diseñados y refinados a lo largo de años de desarrollo, proporcionando un entorno de evaluación robusto y representativo de escenarios de uso práctico.

La metodología experimental utiliza pares de versiones de Core Utilities seleccionadas estratégicamente para representar diferentes tipos de evolución de software: correcciones de bugs, adición de funcionalidades, refactorizaciones, entre otros. Para cada par de programas, se construye el Mimicry Monitor correspondiente y se ejecutan los tests asociados, registrando los veredictos obtenidos (V, IV, NV) junto con las trayectorias de ejecución que los producen. Este análisis permite caracterizar qué tipos de cambios entre versiones generan diferentes clases de veredictos y bajo qué condiciones.

Los experimentos buscan responder preguntas fundamentales sobre el comportamiento práctico del sistema:

- ¿qué proporción de tests resulta en ejecuciones σ-alineadas versus no alineadas?
- ¿cómo influyen las características específicas de cada programa en la distribución de veredictos?
- ¿qué patrones de ejecución conducen a terminaciones tempranas con veredictos definitivos versus aquellas que requieren ejecución completa?

La selección de Core Utilities como base experimental se fundamenta en varias consideraciones estratégicas. Estas herramientas representan programas ampliamente utilizados con funcionalidades bien definidas, cuentan con conjuntos de pruebas extensivos y maduros, y presentan historiales de evolución documentados que permiten identificar pares de versiones con características específicas de cambio. Además, su naturaleza relativamente autocontenida facilita el análisis de resultados y la identificación de patrones de comportamiento.

Los resultados obtenidos proporcionan insights valiosos sobre la aplicabilidad práctica de los Mimicry Monitors, revelando tanto las fortalezas como las limitaciones del enfoque cuando se aplica a software real. El análisis de los veredictos obtenidos permite caracterizar cuándo y cómo los monitores pueden ser efectivos para detectar diferencias significativas entre versiones de programas.

5.1. Benchmark

Como fue anticipado en la sección anterior, el benchmark utilizado para evaluar esta herramienta fue extraído del proyecto de GNU Core Utilities, una colección de utilidades fundamentales del sistema operativo Unix/Linux que proporcionan funcionalidades básicas de manipulación de archivos, texto y procesos. La selección de este conjunto de programas se fundamenta en su amplia adopción, madurez del código, y disponibilidad de historial de versiones bien documentado.

5.1.1. Criterios de selección

Para el benchmark se seleccionaron cinco programas que presentan variaciones significativas en tamaño, complejidad algorítmica y patrones de uso: `cat`, `timeout`, `expand`, `mv`, y `ls`. Esta selección fue diseñada para cubrir un espectro representativo de las características típicas encontradas en software de sistemas, desde utilidades simples de procesamiento de entrada/salida hasta herramientas más complejas que involucran manipulación del sistema de archivos. La tabla 5.1 presenta los cinco programas seleccionados (subjects), junto con su tamaño en líneas de código para las versiones OP y PUA, la cantidad de tests disponibles, y los identificadores de commit correspondientes a cada versión analizada. En algunos casos fue necesario realizar adaptaciones menores al OP. Utilizar el archivo original de un commit anterior podía generar incompatibilidades con el entorno de compilación actual o introducir complejidad innecesaria.

Programa	Líneas OP	Líneas PUA	Tests	Commit OP	Commit PUA
`cat`	813	813	23	dff821d	7386c29
`expand`	312	296	31	97807bf	28b1760
`ls`	5612	5612	459	0c195b6	0d04b98
`mv`	558	528	144	a6ab944	1040610
`timeout`	631	631	24	cb7c210	–

Tab. 5.1: Resumen de las características de los programas analizados, para sus versiones OP y PUA

Para cuatro de los cinco programas se identificó un commit específico que alterara su comportamiento de manera significativa, priorizando cambios que impactaran la lógica de control principal o introdujeran nuevas funcionalidades. En tres de estos casos, las modificaciones se concentraron en la función principal (`main`), facilitando el análisis de σ-alineamiento. El caso de `expand` requirió un enfoque especial, ya que los cambios se ubicaron en una función auxiliar, lo que requirió realizar inlining manual para incorporar la funcionalidad modificada dentro del flujo principal de ejecución.

Para el programa restante, `timeout`, el PUA fue producido artificialmente para propósitos demostrativos de este análisis, sirviendo como ejemplo introductorio. Es decir, que no se tomó ningún commit de referencia para el PUA, sino que se produjo manualmente una nueva versión con un cambio simple.

5.1.2. Descripción de las utilidades evaluadas

cat: Esta utilidad fundamental se encarga de concatenar y mostrar el contenido de archivos. Su simplicidad algorítmica la convierte en un caso de estudio ideal para validar el comportamiento básico del sistema de monitoreo. El programa presenta un flujo de

control relativamente lineal con decisiones basadas en argumentos de línea de comandos y disponibilidad de archivos de entrada.

timeout: Implementa funcionalidad de límite temporal para la ejecución de otros programas, terminando procesos que excedan un tiempo especificado. Esta utilidad introduce complejidad adicional a través de manejo de señales, creación de procesos hijos y sincronización temporal, proporcionando un caso de estudio valioso para evaluar el comportamiento del sistema en presencia de concurrencia y manejo de eventos asíncronos.

expand: Se encarga de convertir caracteres de tabulación en espacios según configuraciones de parada de tabulación especificadas. El programa procesa texto carácter por carácter aplicando reglas de transformación, lo que genera patrones de ejecución intensivos en procesamiento de datos con múltiples decisiones condicionales basadas en el contenido de entrada.

mv: Implementa operaciones de movimiento y renombramiento de archivos y directorios en el sistema de archivos. Esta utilidad presenta alta complejidad debido a la necesidad de manejar múltiples casos especiales: movimientos dentro del mismo sistema de archivos versus entre sistemas diferentes, verificación de permisos, manejo de conflictos de nombres, y preservación de metadatos de archivos.

ls: Proporciona listado de contenidos de directorios con múltiples opciones de formato y filtrado. Representa uno de los programas más complejos del conjunto evaluado, incorporando ordenamiento configurable, formateo de variables de salida, manejo de enlaces simbólicos, y procesamiento de metadatos extendidos de archivos.

5.1.3. Diferencias entre OP y PUA

El análisis comparativo entre las versiones OP (original) y PUA (actualizada) de las GNU Core Utilities revela diferentes patrones de modificación. Mientras algunas utilidades presentan mejoras en funcionalidad y calidad de código, otras muestran cambios que comprometen la funcionalidad esperada. La siguiente tabla resume los cambios más relevantes identificados en cada utilidad:

Utilidad	Cambio Principal	Observación
`ls`	Agregado `print_scontext` en `format_needs_stat`	Es una variable central en el flujo de control
`cat`	Refactorización detección auto-ref.	Mejora legibilidad/rendimiento
`expand`	Agrega chequeo de overflow	Cambio tipo de variable `column number`
`mv`	Simplificación lógica opciones interactivas	Refactor menor en opciones
`timeout`	Fuerza estado de salida a 9	Funcionalidad completamente rota

Tab. 5.2: Resumen de cambios principales en GNU Core Utilities (OP vs PUA)

5.1.4. Particularidades metodológicas

Una consideración importante en el diseño del benchmark fue la disponibilidad de conjuntos de pruebas. Mientras que la mayoría de las utilities seleccionadas cuentan con

test suites extensivas desarrolladas y mantenidas por la comunidad, `expand` presentó una excepción notable al carecer de tests formales en el repositorio principal. Para abordar esta limitación y mantener la consistencia metodológica, se desarrolló una test suite específica para `expand` con la asistencia de un modelo de lenguaje, diseñando casos de prueba que cubrieran los escenarios de uso típicos y casos límite relevantes. Esta aproximación se adoptó específicamente para evitar sesgos inconscientes en el diseño de tests que pudieran favorecer artificialmente el comportamiento del sistema desarrollado.

La implementación del benchmark requirió un manejo cuidadoso de los entornos de compilación, considerando que el proyecto GNU Coreutils presenta un setup de construcción considerablemente complejo con múltiples dependencias y configuraciones específicas. Fue necesario asegurar que todas las bibliotecas requeridas estuvieran disponibles durante las fases de análisis e instrumentación de los programas, incluyendo dependencias tanto estáticas como dinámicas del sistema de construcción autotools. Esta complejidad del entorno se extendió también a las particularidades de cada programa en términos de interfaces de entrada, manejo de argumentos, y patrones de interacción con el sistema de archivos. Esta diversidad permitió evaluar la robustez del sistema de instrumentación ante diferentes estilos de programación y paradigmas de interacción, proporcionando una evaluación comprensiva de la aplicabilidad práctica de los Mimicry Monitors en software real.

5.2. Resultados

En esta sección se presenta un análisis exhaustivo de los resultados obtenidos durante la evaluación de los programas PUA mediante el framework de testing desarrollado. Los resultados revelan patrones significativos en el comportamiento de las herramientas de línea de comandos evaluadas y proporcionan perspectivas valiosas sobre la efectividad del enfoque de verificación implementado.

Programa	**Archivos de Prueba**	**Casos Totales**	**V (Verificado)**	**IV (Imposible Ver.)**	**Sin Veredicto**	**% Éxito (V/Total)**
catPUA	4	23	14	8	1	60.87 %
expandPUA	10	31	12	19	0	38.71 %
lsPUA	59	459	0	459	0	0.00 %
mvPUA	49	144	51	90	3	35.42 %
timeoutPUA	4	24	1	18	5	4.17 %
TOTAL	**126**	**681**	**78**	**594**	**9**	**11.55 %**

Tab. 5.3: Resumen Cuantitativo de Resultados de Pruebas PUA

5.2.1. Evaluación General

Los resultados presentados en la tabla 5.3 muestran la efectividad de los Mimicry Monitors aplicados a diferentes herramientas de GNU Core Utils. La evaluación comprende un total de 126 archivos de prueba que incluyen 681 casos de prueba individuales, revelando patrones distintos de comportamiento según el tipo y la naturaleza de las modificaciones entre versiones de los programas analizados.

Se puede observar que para algunos casos se obtienen ejecuciones que no alcanzan ningún veredicto. Más específicamente, el 1.32 % de los casos de test obtuvieron este resultado. Esto puede suceder si el programa termina por fuera de la estructura de control

representada del monitor. Si bien en algunos casos se puede hacer *inlining* de las funciones llamadas para hacer un análisis más completo, y evaluar todos los puntos posibles de retorno de una función (incluso aquellos fuera del main), en este caso no se creyó primordial, ya que se buscaba dar una idea general del funcionamiento de los monitores. Se puede observar que `timeout` obtuvo la mayor proporción, alcanzando un 20,8 %. Este resultado es coherente con la naturaleza del cambio implementado, dado que al incorporar una modificación que frecuentemente resulta en error, es probable que se requiera invocar métodos externos para abortar la ejecución del programa.

5.2.2. Análisis por Categorías de Programas

En esta sección se presenta un análisis cualitativo de los resultados obtenidos a partir de la ejecución de los Mimicry Monitors sobre distintos programas. Para ello, se agrupan los subjects en tres categorías según la naturaleza e impacto de las modificaciones realizadas entre versiones: (1) programas con cambios localizados y bajo impacto estructural, (2) un caso con modificación artificial controlada, y (3) un programa con divergencia estructural profunda. Cada categoría se analiza en detalle a continuación, considerando el comportamiento del monitor, los porcentajes de veredictos obtenidos y el tipo de transformación observada entre versiones.

5.2.2.1 Programas con modificaciones de bajo impacto: cat, mv y expand

Las herramientas `cat`, `mv` y `expand` conforman una categoría de programas que comparten una característica fundamental: las modificaciones entre versiones afectan el programa de manera localizada, permitiendo la existencia de flujos equivalentes en ambas versiones. Esta categoría muestra porcentajes de éxito que varían entre el 35.42 % y el 60.87 %, estableciendo un rango considerable pero consistentemente positivo de casos verificados.

El programa `cat`, con su 60.87 % de éxito, representa el extremo superior de esta categoría debido a que las modificaciones entre versiones se concentran principalmente en la validación de parámetros y el manejo de casos *edge*, sin alterar significativamente el núcleo algorítmico de concatenación y visualización de archivos.

Por otro lado, `expand` alcanza un 38.71 % de casos verificados, reflejando modificaciones más profundas en la lógica de transformación de caracteres. Las diferencias entre versiones afectan el manejo de tabulaciones y la expansión de caracteres especiales, introduciendo variaciones en el flujo de control que reducen la cantidad de fragmentos comunes ejecutables de manera σ-alineada. Su monitor puede apreciarse en la figura 5.1

La herramienta `mv` presenta un comportamiento intermedio con 35.42 % de éxito, donde las modificaciones se concentran en la validación de operaciones del sistema de archivos y el manejo de metadatos. Aunque estas modificaciones introducen cierta complejidad en el flujo de control, el núcleo de la operación de movimiento y renombrado mantiene suficientes fragmentos comunes como para generar un porcentaje significativo de veredictos V.

Esta categoría demuestra que los Mimicry Monitors obtienen una cantidad considerable de veredictos positivos cuando las modificaciones entre versiones son acotadas, permitiendo que una proporción considerable del código original permanezca semánticamente equivalente y ejecutable de manera alineada. Esto es coherente con la naturaleza de estos monitores, ya que cuanto menor sea la diferencia entre dos versiones de un mismo programa, más probable es que sus ejecuciones estén σ-alineadas.

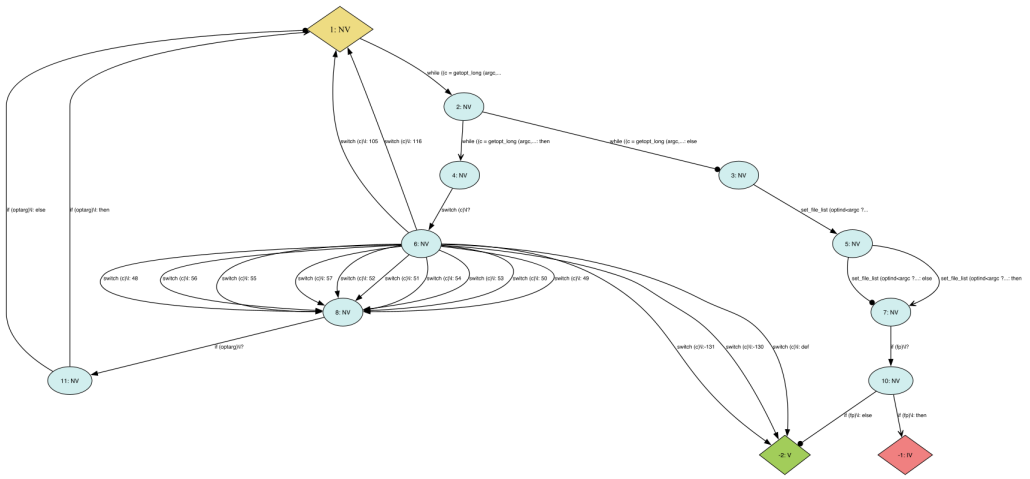


Fig. 5.1: Monitor de `expand` mostrando la estructura de estados y transiciones del autómata generado para la verificación de alineación σ.

5.2.2.2 timeout: Modificación Artificial

El caso de `timeout` merece un análisis particular debido a la naturaleza de las modificaciones introducidas. En este caso, se agregó una modificación de forma artificial, es decir que no estaba vinculada con ningún commit del proyecto. Esto se debe a que fue uno de los ejemplos introductorios, y resultaba de interés observar la proporción de veredictos IV cuando el cambio introducido alteraba muy claramente el funcionamiento del programa. El autómata monitor generado puede observarse en la figura 5.3

Con apenas un 4.17 % de casos verificados, este resultado nos demuestra que el Mimicry Monitor efectivamente pudo reconocer que la mayor parte de las ejecuciones ejecutaban este código agregado que cambiaba el resultado de retorno. La modificación introducida consistió en alterar el valor de salida del programa de manera uniforme, estableciendo un código de retorno específico independientemente del resultado real de la operación.

Esta modificación, aunque aparentemente menor, tiene un impacto devastador en la capacidad de σ-alineación del programa. El cambio sistemático del valor de salida crea una divergencia fundamental en el estado final del programa que el Mimicry Monitor detecta tempranamente, resultando en veredictos IV para la gran mayoría de los casos de prueba. Los 18 casos con veredicto IV de los 24 totales reflejan esta situación, donde el monitor puede determinar con certeza que no existe una ejecución alineada posible en el programa oráculo.

5.2.2.3 ls: Divergencia Estructural Fundamental

El programa `ls` representa el caso más extremo en la evaluación, con un 0.00 % de casos verificados a pesar de contar con 459 casos de prueba distribuidos en 59 archivos. Este resultado no refleja una falla en la implementación de los Mimicry Monitors, sino una situación donde las modificaciones entre versiones han creado una divergencia estructural fundamental que impide cualquier forma de σ-alineación.

El análisis del código revela que las modificaciones en `ls` afectan variables críticas que controlan el flujo principal del programa desde etapas muy tempranas de la ejecución. Estas variables actúan como puntos de decisión que determinan qué ramas del código serán

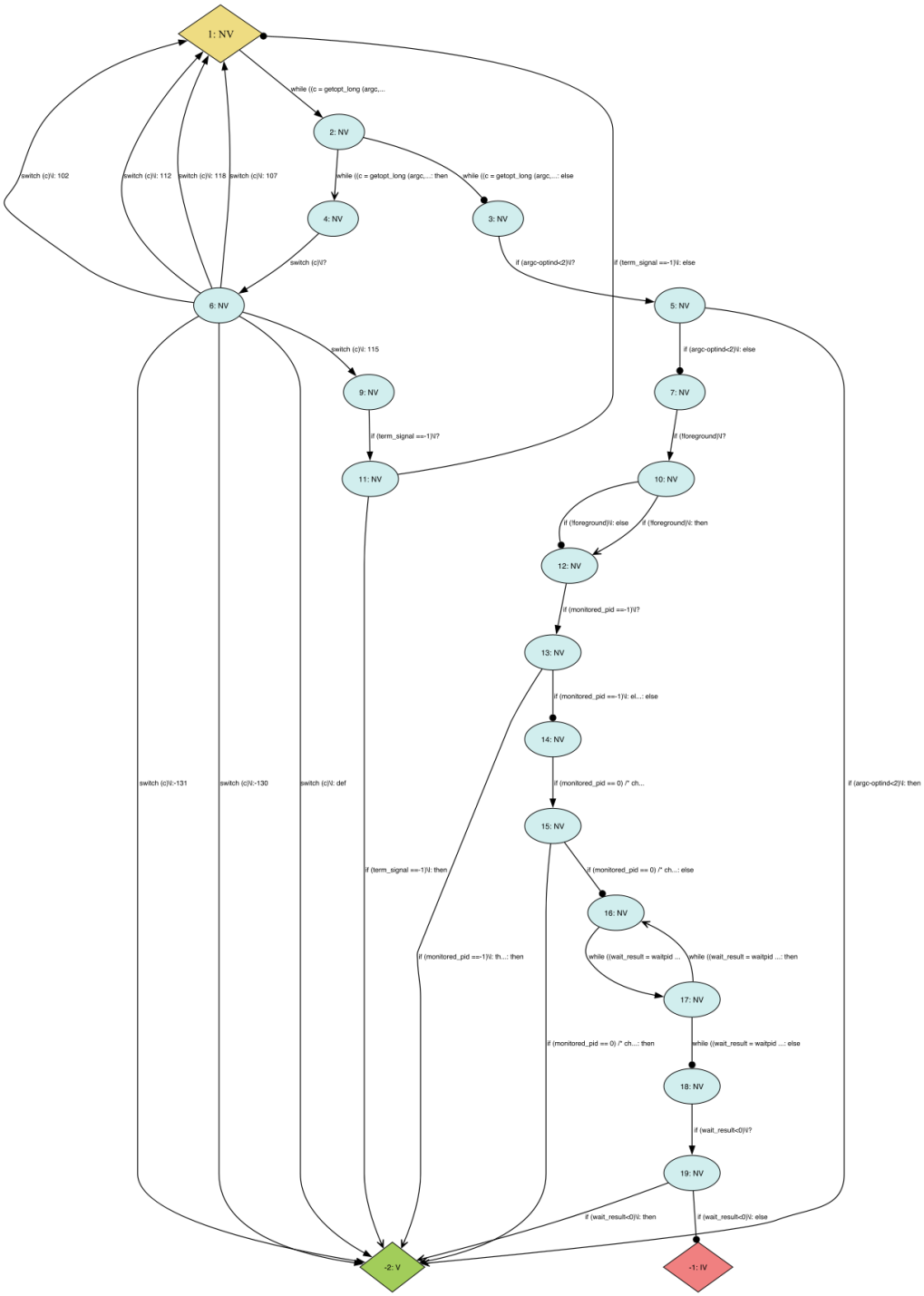


Fig. 5.2: Monitor de timeout mostrando cómo la modificación artificial del valor de retorno genera un autómata que detecta tempranamente la divergencia entre versiones.

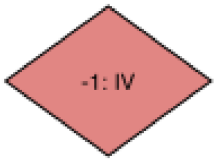


Fig. 5.3: Monitor de ls mostrando la estructura simplificada resultante cuando las modificaciones estructurales impiden cualquier posibilidad de alineación σ.

ejecutadas, y su modificación crea un efecto cascada que propaga diferencias a través de toda la estructura de ejecución del programa.

Cuando el Mimicry Monitor detecta que una variable crítica ha sido modificada fuera del matching σ y posteriormente es utilizada para tomar decisiones de control de flujo dentro de fragmentos que sí pertenecen al matching, el sistema de tracking de data-flow inmediatamente invalida la posibilidad de alineación. Esta situación se traduce en un veredicto IV temprano y definitivo, haciendo imposible que cualquier ejecución posterior genere un veredicto V.

La consistencia del 100 % de veredictos IV en `ls` demuestra que el monitor está funcionando correctamente al detectar esta incompatibilidad fundamental.

5.2.3. Implicaciones para las Aplicaciones Prácticas

Estos resultados revelan que la efectividad de los Mimicry Monitors está intrínsecamente ligada a la naturaleza de las modificaciones entre versiones de software. En contextos de testing de regresión, las herramientas de la primera categoría pueden beneficiarse significativamente de la terminación temprana cuando se alcanza un veredicto V, particularmente en `cat` donde casi el 70 % de los casos pueden ser resueltos sin ejecución completa.

Para aplicaciones de fuzzing diferencial, los casos como `timeout` y `ls` proporcionan información valiosa al identificar rápidamente que toda nueva entrada dirigirá la ejecución hacia comportamientos no equivalentes, permitiendo enfocar los recursos de generación de inputs en la exploración de estas diferencias.

El análisis confirma que los Mimicry Monitors son más efectivos como herramienta de optimización en escenarios donde existen modificaciones graduales y localizadas, mientras que su valor en casos de modificaciones estructurales fundamentales radica más en la detección temprana y definitiva de incompatibilidades que en la optimización de ejecuciones equivalentes.

6. CONCLUSIÓN

Este trabajo ha presentado una implementación integral y evaluación empírica de los Mimicry Monitors, una técnica de verificación en tiempo de ejecución que permite determinar si el comportamiento de un programa bajo análisis (PUA) puede ser imitado por un programa de referencia u oráculo (OP) sin necesidad de ejecutar este último. La investigación abarcó el desarrollo completo del pipeline de construcción de monitores, desde el análisis estático de programas hasta la generación de autómatas optimizados, incluyendo contribuciones metodológicas como la introducción de mecanismos de poda temprana y segunda pasada de optimización post-determinización. Se implementó una herramienta funcional utilizando LLVM como infraestructura base para el análisis e instrumentación de programas en C/C++, y se realizó una evaluación experimental sobre cinco herramientas de GNU Core Utilities (cat, expand, ls, mv, timeout). Esta reveló que la efectividad de los Mimicry Monitors está intrínsecamente ligada a la naturaleza de las modificaciones entre versiones.

A lo largo de este trabajo se han propuesto diversos objetivos e hipótesis sobre los cuales se han podido emitir diferentes veredictos y conclusiones basados en la implementación y evaluación empírica de los Mimicry Monitors aplicados a herramientas reales de GNU Core Utils.

La hipótesis principal del trabajo planteaba que los Mimicry Monitors podían evitar ejecuciones redundantes gracias a su capacidad de anticipar si una ejecución del Programa Bajo Análisis (PUA) tiene una versión σ-alineada correspondiente en el Programa Oráculo (OP). Los resultados obtenidos y presentados en la sección de evaluación confirman parcialmente esta hipótesis, demostrando que efectivamente es posible lograr terminación temprana en una proporción significativa de casos de prueba.

El análisis cuantitativo reveló que se obtuvo un porcentaje considerable de veredictos V en tres de las cinco herramientas evaluadas, destacándose particularmente el caso de `cat` donde se es posible abortar de forma temprana hasta el 60.87 % de los casos de prueba. Este resultado es especialmente relevante considerando que cada terminación temprana representa un ahorro directo en tiempo de ejecución y recursos computacionales. Para escenarios de testing donde se realizan operaciones intensivas en memoria o CPU, incluso un porcentaje reducido de casos V puede ser significativo en términos de ahorro de poder de procesamiento, especialmente cuando se consideran test suites extensas con miles de casos de prueba ejecutados de manera regular en procesos de integración continua.

Los resultados experimentales han permitido caracterizar el comportamiento de los Mimicry Monitors en diferentes contextos, identificando tres categorías distintas de programas según la naturaleza de las modificaciones entre versiones. La primera categoría, representada por `cat`, `mv` y `expand`, demuestra que los MMs son más propensos a encontrar veredictos positivos cuando las modificaciones son graduales y localizadas, manteniendo el núcleo algorítmico estable mientras se introducen cambios periféricos en validaciones o manejo de casos especiales.

La segunda categoría, ejemplificada por `timeout`, ilustra cómo modificaciones aparentemente menores pero que alteran variables centrales pueden tener un impacto desproporcionado en la capacidad de σ-alineación. Esta observación es crucial para comprender las limitaciones inherentes de la técnica cuando se aplica a modificaciones que afectan el

estado global del programa de manera uniforme.

La tercera categoría, representada por `ls`, evidencia situaciones donde las modificaciones estructurales fundamentales hacen imposible cualquier forma de alineación. Aunque estos casos no permiten optimizaciones mediante terminación temprana, proporcionan información valiosa al detectar de manera temprana y definitiva la incompatibilidad entre versiones.

Los resultados obtenidos tienen implicaciones directas para las aplicaciones propuestas en el marco teórico original. En el contexto de testing de regresión, la técnica demuestra mayor utilidad en programas con modificaciones menores, donde el 35-60 % de terminación temprana puede traducirse en ahorros significativos de tiempo de ejecución. Para aplicaciones de fuzzing diferencial, los Mimicry Monitors proporcionan orientación valiosa para dirigir la generación de inputs, tanto identificando casos donde es probable encontrar comportamiento equivalente (veredictos V) como señalizando áreas donde se espera divergencia (veredictos IV).

En el ámbito de depuración con slices ejecutables, los resultados sugieren que la técnica es más aplicable a programas con modificaciones localizadas, donde existe mayor probabilidad de mantener la equivalencia semántica entre el slice y el programa original. Para aplicaciones de multi-ejecución, la capacidad de detectar tempranamente la alineación entre ambas versiones permite tomar decisiones informadas sobre cuándo abortar la ejecución *shadow*, optimizando el uso de recursos computacionales.

6.1. Contribuciones del Trabajo

Este trabajo ha realizado varias contribuciones significativas al campo de verificación diferencial y análisis de programas con fragmentos comunes. En primer lugar, se desarrolló una implementación inicial y completa de la herramienta, con soporte para programas en C/C++. Esta implementación cubre todo el flujo de trabajo: análisis estático de las versiones, generación del monitor correspondiente, e instrumentación automática de la versión PUA para su ejecución monitoreada. Segundo, proporciona la primera evaluación empírica de los Mimicry Monitors aplicados a herramientas reales de software, estableciendo una base de datos experimental valiosa para futuras investigaciones. Por otro lado, caracteriza el comportamiento de la técnica en diferentes tipos de modificaciones de software, proporcionando guías prácticas para determinar cuándo es más probable que la técnica sea efectiva.

Finalmente, demuestra la viabilidad práctica de la técnica en contextos reales de ingeniería de software, validando parcialmente las aplicaciones propuestas en el marco teórico original.

Los resultados confirman que, aunque los Mimicry Monitors no constituyen una solución universal para la optimización de testing de regresión o verificación diferencial, representan una herramienta valiosa en contextos específicos donde las modificaciones entre versiones de software mantienen suficientes fragmentos comunes semánticamente equivalentes. La técnica es especialmente prometedora como componente de frameworks más amplios de análisis diferencial, donde puede combinarse con otras técnicas para proporcionar optimizaciones complementarias.

6.2. Limitaciones Identificadas y Trabajo Futuro

La evaluación ha revelado varias limitaciones importantes que definen el alcance de aplicabilidad de los Mimicry Monitors. La dependencia crítica de la calidad del matching σ se evidencia particularmente en casos como `ls`. Esta limitación sugiere la necesidad de desarrollar técnicas más sofisticadas de análisis semántico que puedan capturar equivalencias no detectables mediante análisis sintáctico simple.

La presencia de casos sin veredicto, aunque minoritaria (1.32 % del total), señala una limitación inherente del diseño actual cuando los programas terminan mediante mecanismos externos al flujo principal. Esta situación podría abordarse en futuras implementaciones mediante la extensión del framework de monitoring para capturar y analizar terminaciones abruptas.

El trabajo futuro debería enfocarse en el desarrollo de algoritmos para la construcción automática del matching σ, explorando técnicas de análisis semántico que vayan más allá de la equivalencia sintáctica. Adicionalmente, sería valioso investigar la aplicación de la técnica a diferentes niveles de granularidad, desde instrucciones individuales hasta bloques básicos más grandes, para optimizar el balance entre precisión y eficiencia.

Una línea futura de trabajo de especial interés consiste en realizar una evaluación del impacto temporal de la herramienta. En particular, se propone medir los tiempos de ejecución con y sin la presencia del Mimicry Monitor, con foco en los casos que resultan en veredictos V. El objetivo es determinar en qué momento de la ejecución se alcanza dicho veredicto y si esto permite abortar tempranamente la ejecución del programa instrumentado, reduciendo así el costo computacional. Este análisis permitiría cuantificar el beneficio práctico de utilizar la herramienta en términos de ahorro de tiempo.

Finalmente, un posible eje de expansión consiste en extender el soporte de la herramienta a otros lenguajes de programación. Gracias a la arquitectura modular del sistema, esta extensión podría lograrse con un esfuerzo razonable, ya que gran parte de los componentes puede reutilizarse o adaptarse con mínimas modificaciones.

Bibliografía

[1] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: Scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pages 13–24, Virtual Event, USA, 2020. ACM.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214, Limerick, Ireland, 2011. Springer.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[4] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, Volume 13, Issues 2–3:219–236, 1990.

[5] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962.

[6] Ferenc Gécseg. Composition of automata. In Jacques Loeckx, editor, *Automata, Languages and Programming*, pages 351–363, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[7] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[8] Marie-Christine Jakobs and Tim Pollandt. diffdp: Using data dependencies and properties in difference verification with conditions. In Paula Herber and Anton Wijs, editors, *iFM 2023 - 18th International Conference on Integrated Formal Methods*, volume 14300 of *Lecture Notes in Computer Science*, pages 40–61, Heidelberg, Germany, 2023. Springer.

[9] Frederick P. Brooks Jr. No silver bullet— essence and accident in software engineering. 1986.

[10] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012*, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717, Berkeley, CA, USA, 2012. Springer.

[11] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, pages 345–355, Saint Petersburg, Russia, 2013. ACM.

[12] Shuvendu K. Lahiri, Andrzej Murawski, Ofer Strichman, and Mattias Ulbrich. Program equivalence (dagstuhl seminar 18151). *Dagstuhl Reports*, 8(4):1–19, 2018.

[13] Chris Lattner. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume 1*, chapter 11. 2011. Available under Creative Commons Attribution 3.0 Unported License.

[14] LLVM Project. LLVM Documentation. https://llvm.org/docs/, 2024. Official LLVM documentation.

[15] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1181–1192, 2016.

[16] Ivan Postolski, Víctor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Verification of programs with common fragments. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 487–491, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Ivan Postolski, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Simulator-based diff-time performance testing. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, pages 81–84, 2019.

[18] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In David N. Card, editor, *Proceedings of the Conference on Software Maintenance (ICSM 1993)*, pages 358–367, Montréal, Quebec, Canada, 1993. IEEE Computer Society.

[19] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[20] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Property directed self composition. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 161–179, New York City, NY, USA, 2019. Springer.

[21] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 57–69, Santa Barbara, CA, USA, 2016. ACM.