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

Verificación Automática de Smart Contracts Move en Sui

Tesis de Licenciatura en Ciencias de la Computación

Leo Mansini

Director: Dr. Diego Garbervetsky

Buenos Aires, 2025

RESUMEN

En el ecosistema de las blockchains, la verificación formal de contratos inteligentes es fundamental para garantizar su seguridad y confiabilidad, evitando vulnerabilidades que podrían resultar en pérdidas económicas o fallas de funcionamiento. *Sui*, una blockchain que utiliza el lenguaje *Move*, presenta un enfoque innovador para la gestión de objetos y transacciones, pero aún cuenta con un ecosistema de herramientas de verificación en desarrollo.

Este trabajo tiene como objetivo explorar métodos para la verificación automática de contratos en Sui. Se realiza un estudio del estado actual de la verificación en Move, evaluando la herramienta *Move Prover* en su capacidad para comprobar propiedades de seguridad en módulos escritos para Sui. Además, se propone un flujo de trabajo alternativo que traduce código Move a Rust de manera controlada, con el fin de habilitar el uso del verificador *Kani*, herramienta enfocada en análisis exhaustivo de propiedades y detección de errores en tiempo de compilación.

Los resultados muestran que Move Prover no es capaz de interpretar el modelo actual de objetos de Move, mientras que la traducción a Rust permitió aprovechar verificadores externos con mucha mejor capacidad de verificación, aunque requirió ajustes manuales y simplificaciones del código para poder representar el funcionamiento de Move en Rust.

Palabras clave: Verificación automática, Smart Contracts, Move, Sui, Rust, Kani Verifier.

ABSTRACT

In the blockchain ecosystem, the formal verification of smart contracts is essential to ensure their security and reliability, preventing vulnerabilities that could lead to economic losses or functional failures. *Sui*, a blockchain that uses the *Move* language, offers an innovative approach to object and transaction management, but still has a verification tool ecosystem in development.

This work aims to explore methods for the automated verification of contracts in Sui. It examines the current state of verification in Move, evaluating the *Move Prover* tool in its ability to check security properties in modules written for Sui. In addition, it proposes an alternative workflow that translates Move code into Rust in a controlled manner, in order to enable the use of the *Kani* verifier, a tool focused on exhaustive property analysis and error detection at compile time.

The results show that Move Prover is not capable of interpreting Move's current object model, whereas translation to Rust made it possible to leverage external verifiers with much greater verification capabilities, although it required manual adjustments and code simplifications in order to represent Move's behavior in Rust.

Keywords: Automated verification, Smart Contracts, Move, Sui, Rust, Kani Verifier.

Índice general

1. INTRODUCCIÓN

La verificación automática de programas permite al programador asegurar propiedades sobre un programa dado. Se basa en contrastar el contrato de un programa (propiedades esperadas de la entrada - precondiciones - y la salida - postcondiciones -) con las propiedades deducidas directamente del código. Mediante el uso de técnicas como el Model Checking, es posible detectar fallas en los programas ante determinados inputs, por ejemplo, una función que falla si es llamada con una variable que vale cero.

Dichas herramientas son particularmente útiles en escenarios donde formas clásicas de verificación del comportamiento de un programa, como el testing, son ineficaces. Tal es el caso de los contratos inteligentes o *"Smart Contracts"*.

Los Smart Contracts son programas ejecutados dentro de una blockchain, es decir, corren dentro de un escenario distribuido y sin completo control del autor una vez desplegado. En general son utilizados para el manejo de recursos financieros, comúnmente transacciones de criptomonedas. Un error en estos programas no solo puede habilitar pérdidas millonarias [1] sino que no puede ser corregido, ya que, por su naturaleza, el código de un Smart Contract no puede ser modificado luego de ser desplegado. Existen variedad de servicios de auditoría dedicados a la búsqueda de estas fallas [2], por lo que la disciplina de verificación es ampliamente valiosa en el contexto de los Smart Contracts.

Además de ser importante, la verificación en Smart Contracts también es desafiante. Los contratos no suelen tener especificación, son ejecutados de forma concurrente y, al interactuar con muchos actores, persisten su propio estado.

1.1. Objetivo y Contribuciones

En el presente estudio se estudiará la factibilidad de verificación automática en contratos escritos en **Move** [3] en la blockchain **Sui** [4].

Se estudiará el estado de verificación actual del lenguaje. En particular, la factibilidad de usar el verificador que en el pasado se incluía en el lenguaje, **Move Prover** [5] en contratos de Sui actual, es decir, en la sintaxis actual del lenguaje, que constantemente es actualizada.

Como método alternativo de verificación, se probará el **Kani Verifier** [6], que en realidad es un verificador del lenguaje **Rust** que utiliza model checking, por lo que además será necesario una traducción desde Move. Para ello se aprovecharán las similitudes sintácticas entre los lenguajes, pero se requerirá especial atención al sistema de objetos universal de Move y al frecuente uso de su librería estándar, ambos no compatibles directamente en Rust, o modelados sin trabajo previo en Kani.

Las contribuciones de este trabajo son:

- Prueba del MoveProver en contratos de Sui.
- Módulo traductor de contratos Sui a programas en Rust.
- Experimentación de verificación automática de contratos traducidos a Rust mediante harnesses de llamadas no determinísticas con Kani Verifier.

1.2. Trabajos relacionados

Se ha estudiado previamente la verificación de contratos escritos en el lenguaje Solidity.

Wang et. al desarrollaron en 2018 VeriSol [7] y en 2019, se desarrolló SolCMC (Solidity Compiler's Model Checker) [8]. Ambos son verificadores de Solidity y funcionan con Model Checking.

Bogdanich [9] realizó un estudio de las técnicas existentes, un benchmark consolidado, y una aplicación llamada VeriMan, un verificador que utiliza VeriSol para obtener las llamadas a funciones (trace) que produjeron el error en un contrato, y Manticore [10] para generar los parámetros concretos, compatibles con la blockchain (incluyendo tipos como `address` o `bytes`). Además incluyó un lenguaje de especificación para Solidity utilizable con VeriMan.

También se ha trabajado la aplicación de EPAs (Enableness Program Abstraction) para verificación de Smart Contracts. Es posible modelar los contratos con abstracciones de máquinas de estado que describen cuales operaciones pueden ejecutarse en un estado dado de un contrato. Con esta abstracción se puede determinar si una secuencia de operaciones lleva a un error, generando potencialmente un ejemplo de *exploit*, es decir, un uso no esperado del contrato que podría producir pérdidas monetarias a los involucrados.

En 2013 De Caso et. al [11] presentan la técnica de EPA como tal, y en 2022 Godoy et. al [12] se propone el uso de EPAs y de abstracciones por predicado en validación de smart contracts, de forma que se encuentren secuencias de llamadas a métodos del contrato que resulten en violaciones de la especificación, como se da en los hackeos que comúnmente sufren los smart contracts.

Posteriormente se estudiaron aplicaciones de estas abstracciones y su construcción automática con el uso de herramientas de verificación estática.

Torres [13] y Wappner [14] experimentaron con VeriSol y Manticore respectivamente. Ambos implementaron EPAs exitosamente para contratos de Solidity, validando con benchmarks de Azure y Smartpulse. Por su parte, Grinspan [15] evaluó las técnicas de validación con EPAs pero utilizando verificación dinámica con la herramienta de fuzzing Echidna [16].

Sobre verificación en Move, en 2022 Dill et al. [17] desarrollaron un verificador automático para el lenguaje en el contexto de la blockchain Diem, Move Prover. Este verificador es una de las herramientas que se evaluarán en este trabajo.

2. CONTEXTO TEÓRICO

2.1. Verificación Automática

La verificación automática de programas es el proceso de comprobar formalmente que un sistema cumple con ciertas propiedades deseadas, como correctitud dada una especificación, o ausencia de errores. Se basa en herramientas que analizan el comportamiento del programa contrastándola con una especificación, es decir una descripción formal de qué debe hacer el programa. El valor de estas técnicas está en la capacidad de detectar errores que podrían pasar desapercibidos en pruebas tradicionales, como el testing. Comúnmente utilizado en contextos de sistemas críticos, la verificación automática proporciona garantías más sólidas sobre el comportamiento del software.

Una de las principales técnicas de verificación automática es el **Model Checking** [18]. Esta consiste en explorar de forma sistemática todos los posibles estados que puede alcanzar un programa para verificar si cumple ciertas propiedades lógicas, como seguridad o terminación. Si se encuentra una violación a la propiedad, se puede obtener un contraejemplo, es decir, valores concretos para las variables, que demuestra el error. Para aplicar esta técnica a un programa concreto, suele ser necesario escribir funciones especiales llamadas *harnesses*, que definen los escenarios de entrada que deben ser explorados por el verificador.

Otra técnica relevante es la **Verificación Deductiva** [19], que se basa en demostrar formalmente que un programa cumple una especificación lógica mediante el uso de solvers lógicos. A diferencia del model checking, la verificación deductiva traduce tanto el código como sus especificaciones en fórmulas lógicas que deben ser válidas para todo posible input. Esta técnica permite razonar sobre programas más complejos y con estructuras dinámicas, pero requiere mayor intervención manual, especialmente en la escritura de invariantes y especificaciones precisas.

Además de la verificación estática, existe la verificación dinámica. Model checking y verificación deductiva pertenecen a la categoría de técnicas de verificación estática, mientras que un ejemplo de verificación dinámica es la verificación mediante testing. Los primeros analizan el programa sin necesidad de ejecutarlo, examinando la estructura y lógica del programa, pero el segundo necesita ejecutar el código para verificarlo. Así, mientras la verificación estática puede llegar a garantizar la corrección respecto de todas las posibles ejecuciones dentro de un programa dado, el testing inspecciona un subconjunto finito de comportamientos posibles del programa, por lo que puede encontrar errores, pero nunca garantizar su ausencia.

2.2. Smart Contracts

Los smart contracts son programas que se ejecutan automáticamente en una blockchain cuando se cumplen ciertas condiciones predefinidas. Estos permiten automatizar acuerdos entre partes sin necesidad de intermediarios, manteniendo la transparencia, trazabilidad e inmutabilidad que caracteriza a este entorno. Una blockchain es una estructura de datos distribuida diseñada para registrar transacciones de forma segura, transparente e inmutable. Está compuesta por una cadena de bloques, donde cada bloque contiene un conjunto

de transacciones y un hash que enlaza con el bloque anterior. Esta arquitectura hace que alterar la información registrada en un bloque implique modificar todos los bloques siguientes, para lo cual es necesario el consenso de la red.

Una blockchain no depende de una autoridad central. En su lugar, múltiples nodos de una red descentralizada participan en la validación y el mantenimiento del registro. Estos nodos siguen un protocolo de consenso, que asegura que todos tengan una copia coherente del historial de transacciones. Esta descentralización proporciona resiliencia frente a fallos y ataques, y garantiza que los datos sean verificables y confiables.

La blockchain es una tecnología base para una variedad de aplicaciones más allá de las criptomonedas. Una de las más destacadas son los smart contracts.

Los smart contracts corren de manera descentralizada y sin intermediarios, garantizando la inmutabilidad del código una vez desplegado. Debido a que manejan activos digitales y no pueden ser modificados después de su lanzamiento, estos contratos requieren un nivel particularmente alto de confianza y corrección.

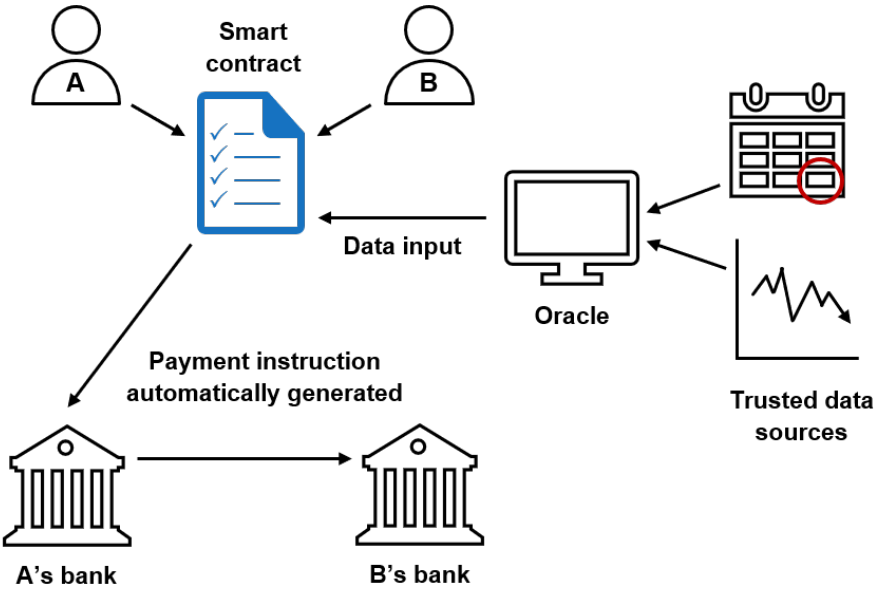

Fig. 1: Ejemplo de uso de un Smart Contract. Al ponerse dos actores de acuerdo, se activa el programa que, ante condiciones determinadas, genera movimientos de activos. Para mantener el acuerdo, el programa no puede ser modificado con libertad.

Una vulnerabilidad en un smart contract puede resultar en la pérdida de fondos o en comportamientos no deseados que no pueden corregirse sin desplegar una nueva versión del contrato. A diferencia del software tradicional, los errores en estos sistemas no pueden ser solucionados mediante parches o actualizaciones, lo que hace que el costo de un fallo pueda ser extremadamente alto. Por esta razón, la comunidad ha puesto especial énfasis en el uso de auditorías, pruebas exhaustivas y técnicas formales de verificación para asegurar su correcto funcionamiento.

Entre las técnicas de verificación, la verificación estática ha tomado relevancia por su capacidad para explorar exhaustivamente los posibles comportamientos del contrato, y detectar errores lógicos, violaciones de invariantes y condiciones no deseadas. Esta forma de análisis permite aumentar la confianza en contratos complejos, y responde a la necesidad crítica de garantías sólidas en un entorno donde los errores no son tolerables.

2.3. La blockchain Sui

Sui es una blockchain de Capa 1 diseñada para ofrecer baja latencia, con un enfoque particular en la gestión de activos digitales. Para lograr estos objetivos, su arquitectura introduce varias particularidades, como por ejemplo su enfoque del consenso. La red opera bajo un modelo de Prueba de Participación Delegada (Delegated Proof-of-Stake - DPoS), donde los poseedores de la moneda nativa delegan sus tokens a un conjunto de validadores. La particularidad más significativa de Sui es su capacidad para procesar transacciones en paralelo, para lo cual distingue entre dos tipos de activos: los objetos poseídos (controlados por una única dirección) y los objetos compartidos (accesibles por múltiples usuarios). Las transacciones que involucran solo a los primeros no requieren un consenso global y se procesan casi instantáneamente, mientras que únicamente aquellas que modifican objetos compartidos son ordenadas a través de un motor de consenso de alto rendimiento llamado Mysticeti [20].

La seguridad de este mecanismo y la operación de la red se sustentan en su criptomoneda nativa, SUI. Este token cumple dos funciones principales: participar en el mecanismo de DPoS mediante el staking y pagar las comisiones de gas, las tarifas requeridas para ejecutar transacciones y almacenar datos en la blockchain. El modelo económico de Sui es notable por su fondo de almacenamiento, un mecanismo donde una parte de las tarifas pagadas por el guardado de datos se reserva para compensar a los futuros validadores[1].

Finalmente, la lógica de los contratos inteligentes se ejecuta en la Sui Move VM, una máquina virtual adaptada del lenguaje de programación Move. Esta versión de la VM ha sido modificada significativamente para ser compatible con el modelo de datos centrado en objetos de Sui[2], donde cada activo digital es una entidad independiente en la red, a diferencia de los modelos basados en cuentas de otras blockchains[3].

[1] https://docs.sui.io/concepts/tokenomics
[2] https://docs.sui.io/concepts/object-model
[3] https://ethereum.org/en/developers/docs/accounts/

3. HERRAMIENTAS

En esta sección se explicará sobre las herramientas preexistentes utilizadas, es decir Move, Move Prover y Kani Verifier, y la contribución de este trabajo, el traductor de Move a Rust y el resto del proceso necesario para verificar Move con Kani.

3.1. Move on Sui

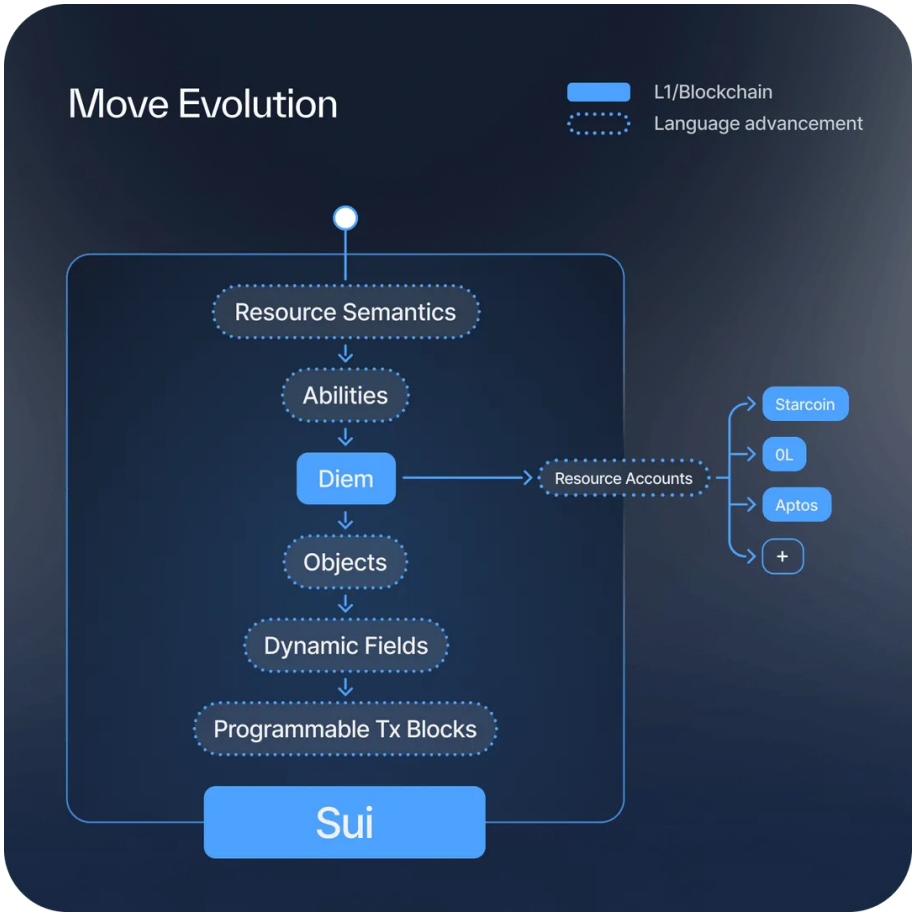

Fig. 2: Diagrama mostrando lo construido por Sui sobre Move. Sui difiere principalmente en su sistema de objetos de otras blockchain que salieron de Diem, como Aptos.

En este trabajo estaremos experimentando con contratos escritos en Move on Sui, es decir, en lenguaje Move para la blockchain Sui. Si bien al hacer análisis estático nos limitaremos a cuestiones del lenguaje y no nos concierne dónde va a ejecutarse, vale la pena aclarar la blockchain en consideración, dado que el lenguaje Move ha mutado según la blockchain que lo adoptó.

En el caso particular de Sui, se introdujeron muchos cambios respecto al Move original. Podemos ver en la figura 2, extraída del sitio oficial de Sui, una representación de las diferencias. Uno de los más importantes es la incorporación de un modelo centrado en

objetos en lugar del modelo tradicional de cuentas con recursos, como en Diem o Aptos. En este modelo, los objetos son identificables de forma única y global, poseen un dueño explícito y pueden ser transferidos, modificados o compartidos siguiendo reglas definidas por el sistema. Esta estructura está diseñada para representar activos digitales de manera directa, con soporte nativo para operaciones como mover, destruir o mutar objetos.

Además del modelo de objetos, Sui introduce campos dinámicos (Dynamic Fields) y bloques de transacciones programables (Programmable Transaction Blocks), que permiten encadenar múltiples operaciones en una misma transacción de forma más flexible. Estas incorporaciones se integran dentro del Sui Framework, una colección de módulos escritos en Move que encapsula las reglas del sistema y provee primitivas reutilizables.

El lenguaje Move fue diseñado para tener una forma segura de programar Smart Contracts. Sus particularidades estan orientadas a limitar el comportamiento de los objetos, manipular estructuras encapsuladas, y mantener las entidades de la ejecución vinculadas con entidades reales certificadas en la blockchain.

3.1.1. Análisis de un Contrato en Move on Sui

Para ilustrar en detalle los elementos del lenguaje Move en su versión para la blockchain Sui, se presenta el contrato de ejemplo en el Listing 1. Este módulo, `trophy_shop`, define un objeto digital único (`Trophy`) que puede ser adquirido por los usuarios a través de una "tienda" (`TrophyShop`).

```
1  /// Módulo de ejemplo que define un objeto "Trophy" que puede ser comprado.
2  module my_trophy::trophy_shop {
3      use sui::coin::{Self, Coin}; // Modela una moneda y su cantidad.
4      use sui::object::{Self, UID}; // Maneja la creación y destrucción de los
       ↪ objetos.
5      use sui::tx_context::{Self, TxContext}; // Trae el contexto del block en
       ↪ ejecución
6      use sui::transfer; // Necesario para modificar la propiedad de objetos
7      use sui::sui;
8
9      /// El objeto Trophy que se puede poseer.
10     /// Tiene la habilidad 'key' para poder tener un Id, que le permite ser un
       ↪ objeto único en el sistema de objetos de Sui.
11     /// Tiene la habilidad 'store' para poder ser guardado dentro de otros
       ↪ objetos.
12     public struct Trophy has key, store {
13         id: UID,
14         mint_number: u64,
15         magical_power: u64,
16     }
17
18     /// Objeto compartido que representa la tienda.
19     /// Contiene el precio y el contador de trofeos vendidos.
20     public struct TrophyShop has key {
21         id: UID,
22         price: u64,
23         trophies_minted: u64,
24     }
25
```

```
26      /// Se ejecuta una única vez cuando el módulo es publicado.
27      /// Crea el objeto 'TrophyShop' y lo comparte para que todos puedan
        ↪ interactuar con él.
28      fun init(ctx: &mut TxContext) {
29          let shop = TrophyShop {
30              id: object::new(ctx),
31              price: 1000, // Precio de 1000 MIST
32              trophies_minted: 0,
33          };
34          transfer::share_object(shop);
35      }
36
37      /// Permite a un usuario comprar un nuevo trofeo.
38      /// La función es 'public entry' para poder ser llamada directamente en una
        ↪ transacción.
39      public entry fun buy_trophy(
40          shop: &mut TrophyShop,
41          payment: Coin<SUI>,
42          ctx: &mut TxContext
43      ) {
44          // Se asegura de que el pago sea suficiente.
45          assert!(coin::value(&payment) >= shop.price, 'Pago insuficiente');
46
47          // Incrementa el contador de trofeos.
48          shop.trophies_minted = shop.trophies_minted + 1;
49
50          // Crea el nuevo trofeo.
51          let new_trophy = Trophy {
52              id: object::new(ctx),
53              mint_number: shop.trophies_minted,
54              magical_power: 9001,
55          };
56
57          // Transfiere el trofeo al comprador (dueño del pago).
58          transfer::public_transfer(new_trophy, tx_context::sender(ctx));
59
60          // Transfiere el pago al dueño de la tienda (en este caso, por
            ↪ simplicidad se quema/destruye).
61          coin::burn_for_testing(payment);
62      }
63
64      // --- Tests ---
65
66      #[test]
67      /// Prueba que la compra de un trofeo funciona correctamente si el pago es
        ↪ exacto.
68      fun test_buy_trophy_succeeds() {
69          use sui::test_scenario;
70
71          // 1. Iniciar un escenario de prueba con una dirección de usuario.
72          let admin = @0xAD;
73          let user = @0xUSER;
74          let mut scenario = test_scenario::begin(admin);
```

```
75
76          // 2. Ejecutar el inicializador del módulo para crear la tienda.
77          {
78              init(test_scenario::ctx(&mut scenario));
79          };
80
81          // 3. Pasar al siguiente "epoch" de la transacción para que el objeto
            ↪  compartido (la tienda) esté disponible.
82          test_scenario::next_tx(&mut scenario, user);
83          {
84              // 4. Obtener la tienda y crear una moneda para el pago.
85              let mut shop = test_scenario::take_shared<TrophyShop>(&scenario);
86              let payment = coin::mint_for_testing<SUI>(shop.price,
                ↪  test_scenario::ctx(&mut scenario));
87
88              // 5. Llamar a la función a probar.
89              buy_trophy(&mut shop, payment, test_scenario::ctx(&mut scenario));
90
91              // 6. Verificar que el estado final es el esperado.
92              assert!(shop.trophies_minted == 1, 'El contador de trofeos debería
                ↪  ser 1');
93
94              // 7. Devolver los objetos mutados al escenario.
95              test_scenario::return_shared(shop);
96          };
97
98          // 8. Finalizar el escenario.
99          test_scenario::end(scenario);
100     }
101
102     #[test]
103     #[expected_failure]
104     /// Prueba que la compra falla si el pago es insuficiente.
105     fun test_buy_trophy_fails_if_payment_is_low() {
106         use sui::test_scenario;
107
108         let admin = @0xAD;
109         let user = @0xUSER;
110         let mut scenario = test_scenario::begin(admin);
111         {
112             init(test_scenario::ctx(&mut scenario));
113         };
114
115         test_scenario::next_tx(&mut scenario, user);
116         {
117             let mut shop = test_scenario::take_shared<TrophyShop>(&scenario);
118             // El pago es menor que el precio.
119             let payment = coin::mint_for_testing<SUI>(shop.price - 1,
                ↪  test_scenario::ctx(&mut scenario));
120
121             buy_trophy(&mut shop, payment, test_scenario::ctx(&mut scenario));
122
123             test_scenario::return_shared(shop);
```

```
124         };
125         test_scenario::end(scenario);
126     }
127 }
```

Listing 1: Código Move on Sui ejemplo, donde se tiene una tienda de trofeos que pueden ser comprados. El módulo utliliza las *abilities*, el Sui Framework, la función de inicialización del módulo, y funcionalidades de testing.

3.1.1.1. Sintaxis básica

La sintaxis de Move es fuertemente inspirada en la de Rust. Muchos de los tipos nativos son los mismos, se comparte el concepto de *Ownership* de las variables, las referencias pueden o no ser mutables, puntos y comas al final de lineas, dos puntos para el tipado, llaves para los scopes, entre otros. Por ejemplo, en el Listing 1 podemos observar el uso de dos puntos para el tipado de parámetros y campos (e.g., `price: u64`, línea 22), referencias mutables (`&mut`, línea 40), el punto y coma para finalizar sentencias y el uso de llaves para delimitar los scopes de módulos y funciones. Estas similitudes serán útiles para la sección 4.1.

3.1.1.2. Abilities

Una de las filosofías principales de Move es manejar activos digitales de forma "realista" es decir, los objetos deben ser de la propiedad de alguien, no se deberían copiar sin esfuerzo, ni tampoco descartar [21].

A los objetos se les puede dar *abilities*: *copy*, *drop*, *key*, y *store*. Cada una de estas abilities personalizan los tipos y habilitan comportamiento en sus instancias que comúnmente se da por hecho en los objetos de otros lenguajes:

- Copy: Permite al objeto ser copiado.
- Drop: Permite al objeto ser descartado, eliminado.
- Key: Permite al objeto ser utilizado como una Key en un storage. Internamente deben tener un campo UID, lo cual lo hace único y no copiable ni "dropeable".
- Store: Permite al objeto ser guardado en structs que tengan la habilidad de key.

En el ejemplo del Listing 1, esto se observa en la línea 12, donde se declara el struct `Trophy` con las habilidades key y store.

La habilidad key es fundamental en Sui, ya que designa a `Trophy` como un objeto con un ID único que puede ser poseído y transferido en la red, requiriendo que su primer campo sea un UID. Por otro lado, la habilidad store permite que este objeto sea contenido o "almacenado" dentro de otro struct. Es igualmente notable la ausencia de las habilidades copy y drop, una decisión de diseño intencional para asegurar que cada `Trophy` sea un activo único que no pueda ser duplicado ni destruido accidentalmente.

3.1.1.3. Inicializador de Módulo (init)

Move on Sui permite definir una función especial init (línea 28), la cual se ejecuta una única vez al momento de publicar el contrato. Su propósito es establecer el estado inicial del módulo. En nuestro ejemplo, la función init crea la instancia única de `TrophyShop` y la convierte en un objeto compartido mediante la llamada a `transfer::share_object` en la línea 34. Esto es un patrón de diseño clave en Sui, ya que permite que cualquier usuario en la red pueda interactuar con la tienda.

3.1.1.4. Uso de Objetos del Framework

El desarrollo en Sui depende en gran medida del Sui Framework, una librería estándar que provee las herramientas para las operaciones más comunes. La función `buy_trophy` (línea 39) ilustra el uso de varios de sus componentes:

El argumento `payment` (línea 41) es de tipo Coin<SUI>, un objeto del framework que representa la moneda nativa de la red.

Para verificar el monto del pago, se utiliza la función `coin::value` en la línea 45.

La creación de nuevos objetos únicos se realiza a través de la función canónica `object::new`, que requiere el contexto de la transacción (ctx), como se ve en la línea 52.

3.1.1.5. Pruebas Unitarias

Move incluye un framework de pruebas integrado para verificar la lógica de un contrato de forma aislada. En el Listing 1 se muestran dos tests:

La anotación `#[test]` (líneas 66 y 102) marca una función como una prueba unitaria.

El módulo `test_scenario` se utiliza para simular un entorno de ejecución, permitiendo crear usuarios, un valor en moneda, y llamar a las funciones del contrato.

La anotación `#[expected_failure]` (línea 103) declara que el test es exitoso solo si la ejecución falla, lo que permite verificar que las aserciones de seguridad (como la que valida el pago en la línea 45) funcionan correctamente.

3.2. Move Prover

El Move Prover es un verificador automático diseñado específicamente para el lenguaje Move, orientado a garantizar propiedades de seguridad en programas que gestionan activos digitales. Su desarrollo inicial fue impulsado por el equipo de Diem Blockchain (anteriormente conocido como Libra), y continuó evolucionando aún luego de la adopción por otras plataformas como Sui y Aptos.

Move Prover es un verificador deductivo. Como tal, el proceso de verificación se basa en anotaciones declarativas escritas junto al código fuente, conocidas como especificaciones. Estas incluyen precondiciones, postcondiciones, invariantes de estado, y otras propiedades que el programa debe cumplir. A partir de este módulo anotado, el Move Prover genera una representación intermedia a partir del bytecode compilado, y transforma tanto el código como las especificaciones en un modelo interno basado en objetos y operaciones del lenguaje.

Una vez construido el modelo, el Move Prover lo traduce al lenguaje intermedio **Boogie**[22], ampliamente utilizado en el campo de la verificación deductiva. Boogie actúa como

una capa de abstracción que facilita la traducción a fórmulas lógicas que luego son evaluadas por un SMT solver (típicamente Z3 [23]), encargado de determinar si las propiedades pueden ser violadas o no. En caso de que alguna propiedad no se cumpla, la herramienta provee contraejemplos que permiten entender qué condiciones llevaron a la violación.

Una limitación importante es que el Move Prover depende del modelo de ejecución del Move original, lo que complica su uso directo en plataformas con extensiones o modificaciones significativas del lenguaje, como Sui. En ese sentido, su aplicabilidad fuera del contexto Diem/Aptos es restringida, y motivó la búsqueda de alternativas para este trabajo.

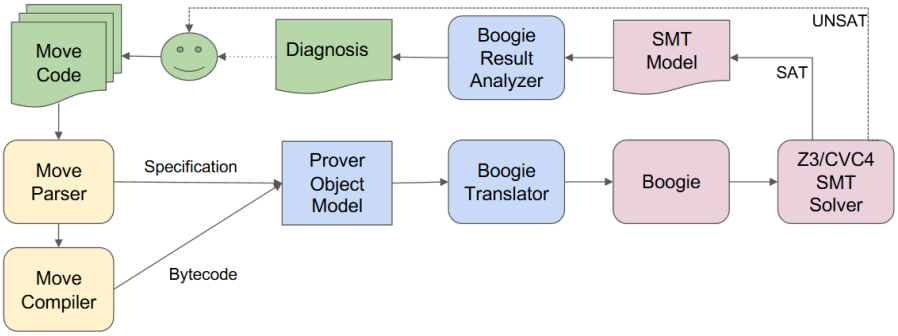

Fig. 3: Arquitectura de Move Prover

```
public fun pay_from_sender(payee: address, amount: u64) acquires T
{
    Transaction::assert(payee != Transaction::sender(), 1); // new!
    if (!exists<T>(payee)) {
        Self::create_account(payee);
    };
    Self::deposit(
        payee,
        Self::withdraw_from_sender(amount),
    );
}

spec fun pay_from_sender {
    // ... omitted aborts_ifs ...
    aborts_if amount == 0;

    aborts_if global<T>(sender()).balance.value < amount;

    ensures exists<T>(payee);

    ensures global<T>(sender()).balance.value ==
    old(global<T>(sender()).balance.value) - amount;
}
```

Listing 2: Código ejemplo de un método en Move con su especificación.

El código presentado en Listing 2 es un ejemplo de código Move en Diem, no es código de Sui reciente. Como se ha mencionado anteriormente, la variante de Sui de Move ha ido iterando con el tiempo y, entre otros cambios, se han removido las especificaciones del lenguaje. Entre otros, las diferencias que vemos en el Listing 2 con código de Sui actual son:

- El uso de `global<t>(address)`. Accede al guardado de recursos bajo direcciones y en Sui los objetos son administrados directamente, teniendo en su definición su propiedad.
- `create_account`. Ahora en la blockchain todos los objetos únicos generan su ID con `object::new(ctx)`.
- Los bloques *spec* fueron eliminados de la sintaxis de Sui.

A la fecha no está *oficialmente* soportado el Move Prover, no está incluído en el repositorio de Sui y su modelo de objetos no está siendo actualizado con los nuevos objetos que va sumando Sui al framework.

3.3. Kani Verifier

Kani Verifier es un Bounded Model Checker para el lenguaje Rust.

A diferencia de los verificadores deductivos que se basan en especificaciones formales escritas como anotaciones (como lo es el Move Prover, seccion 3.2), Kani utiliza harnesses de prueba. Un harness es una función que invoca el código a verificar y sobre la cual Kani explora exhaustivamente las posibles rutas de ejecución para encontrar fallos.

Para entender su funcionamiento, estudiaremos un ejemplo básico.

```
fn sum_of_first_and_last(slice: &[u8]) -> u8 {
    // PRECONDICIÓN: El 'slice' no debe estar vacío.
    // Si slice.len() == 0, la siguiente línea causará un panic.
    let first = slice[0];
    let last = slice[slice.len() - 1];

    // Usamos saturating_add para evitar desbordamientos y enfocarnos en la precondición.
    first.saturating_add(last)
}


#[kani::proof]
fn check_sum_harness() {
    // 1. Entrada no determinística.
    // Necesita un máximo de longitud para el vector, se setea el 5 suficiente para este ejemplo.
    let v: Vec<u8> = kani::vec::any_vec::<u8, 5>();


    // 2. Establecimiento de la precondición
    kani::assume(!v.is_empty());

    // 3. Llamada a la función bajo la precondición
    sum_of_first_and_last(&v);
}
```

Listing 3: Ejemplo de función verificada con un harness de Kani.

Consideremos la función en el Listing 3. Su objetivo es sumar el primer y último elemento de un slice de números. Esta función tiene una precondición clara, solo es válida si el slice no está vacío, de otra forma el acceso `slice[0]` provocaría un error (panic, en Rust).

El harness introduce los conceptos centrales de Kani:

- La anotación `#[kani::proof]` (línea 12) marca la función como un punto de entrada para la verificación. Kani analizará todas las rutas de ejecución posibles dentro de esta función.
- Se genera la entrada no determinística con `kani::vec::any_vec()` (línea 15), lo cual es un helper que construye la estructura (vector) con una longitud máxima (cinco, en este ejemplo), donde cada elemento del vector es generado no determinísticamente con `kani::any()`. Esta es la primitiva fundamental del model checking. En lugar de darle un valor concreto a la variable `v`, `kani::vec::any_vec()` le indica al verificador que debe considerar todos los `Vec<u8>` posibles.
- Se establecen precondiciones con `kani::assume()` (línea 20), definimos el "contrato" de nuestra función. La línea `kani::assume(!v.is_empty())` le ordena a Kani que ignore todos los casos en los que el vector `v` esté vacío y analice únicamente aquellos que cumplen con la precondición.

Al ejecutar el harness del Listing 3, Kani explora todas las posibles entradas que no son vacías. Como la función es segura para estos casos, la salida del verificador presentada en el Listing 4, es exitosa.

```
VERIFICATION:- SUCCESSFUL
Verification Time: 3.4872322s

WARNING: Kani could not produce a concrete playback for
↪  `harnesses::example_kani::check_sum_harness` because there were no failing panic
↪  checks or satisfiable cover statements.
INFO: The concrete playback feature never generated unit tests because there were no
↪  failing harnesses.
Complete - 1 successfully verified harnesses, 0 failures, 1 total.
```

Listing 4: Salida de Kani Verifier al correr la verificación ejemplo del Listing 3.

Este resultado es una prueba formal de que, para cualquier vector no vacío, la función se ejecuta sin errores. Si omitimos la línea con `kani::assume`, Kani explora el caso del vector vacío, detectando el pánico y reportando lo que vemos en el Listing 5, proveyendo además un contraejemplo que nos mostraría que la falla ocurre cuando la entrada es [].

```
VERIFICATION:- FAILED
Verification Time: 1.7666851s
```

````
Concrete playback unit test for `harnesses::example_kani::check_sum_harness`:
```
#[test]
fn kani_concrete_playback_check_sum_harness_11986038205070805422() {
    let concrete_vals: Vec<Vec<u8>> = vec![
        // 0ul
        vec![0, 0, 0, 0, 0, 0, 0, 0],
    ];
    kani::concrete_playback_run(concrete_vals, check_sum_harness);
}
```
````

Listing 5: Salida de Kani Verifier al correr la verificación ejemplo del Listing 3, eliminando la linea de `kani::assume()`. El caso de falla es un input que rompe la precondición de la función a verificar.

La salida de Kani muestra los valores de entrada que causan el fallo. En este caso, la única entrada no determinística es la longitud del vector, que es de tipo usize. El valor vec![0, 0, ...] es la representación en 8 bytes del número 0. Por lo tanto, Kani nos informa que el fallo ocurre cuando la longitud del vector es 0, es decir, cuando el vector está vacío.

4. TRADUCIENDO MOVE A RUST

Una de las decisiones de diseño fundamentales detrás del lenguaje Move en Sui fue adoptar conceptos clave de Rust, en particular el modelo de ownership y borrowing, como base para su sistema de tipos. Esta similitud no solo favorece la seguridad en la manipulación de recursos digitales, sino que también facilita la interpretabilidad del lenguaje, haciéndolo similar a uno como Rust, ya utilizado en otras aplicaciones vinculadas a blockchains en general y a Sui en particular.

Además de compartir principios de seguridad de memoria, Move hereda parcialmente la sintaxis de Rust. Esto significa que muchas construcciones en Move pueden trasladarse a Rust con modificaciones mínimas, sobre todo en lo que respecta a estructuras, funciones y control de flujo. Si bien algunas abstracciones de Move, como los objetos o abilities, no tienen un equivalente directo en Rust, es posible modelarlas mediante tipos personalizados o patrones específicos.

Esta cercanía sintáctica abre la posibilidad de llevar contratos escritos en Move hacia módulos en Rust, con el objetivo de aplicar sobre ellos herramientas maduras de verificación formal disponibles en el ecosistema de Rust. Tal es el caso de Kani Verifier (estudiado en sección 3.3).

Se plantea la arquitectura presentada en la figura 4. Los contratos en Move serán procesados por un traductor de Move a Rust, luego se implementará un harness para decidir los estados a explorar en el contrato, y finalmente se enviarán a Kani el contrato y su harness.

4.1. Módulo de traducción

Para la traducción de Move a Rust se usó un script en Python [24] que con una lista de reemplazos basados en reglas Regex, más unas funciones para reestructurar el código, convierten un subconjunto satisfactorio de contratos Move a módulos funcionales en Rust.

A continuación, en la figura 5, se detallan los pasos que atraviesa el código Move para ser transformado en código funcional en lenguaje Rust, mientras que podemos ver el algoritmo 1 escrito en pseudocódigo.

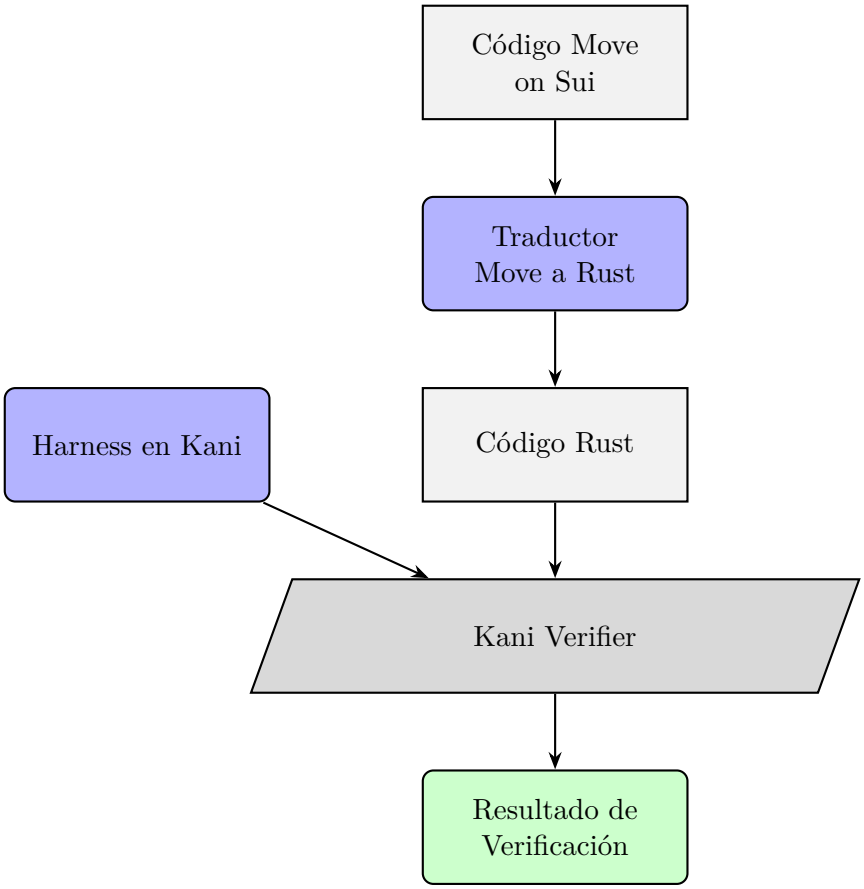

Fig. 4: Arquitectura del flujo de verificación de Move on Sui utilizando Kani. En azul se encuentran el traductor y el harness, ambos componentes desarrollados en este trabajo.

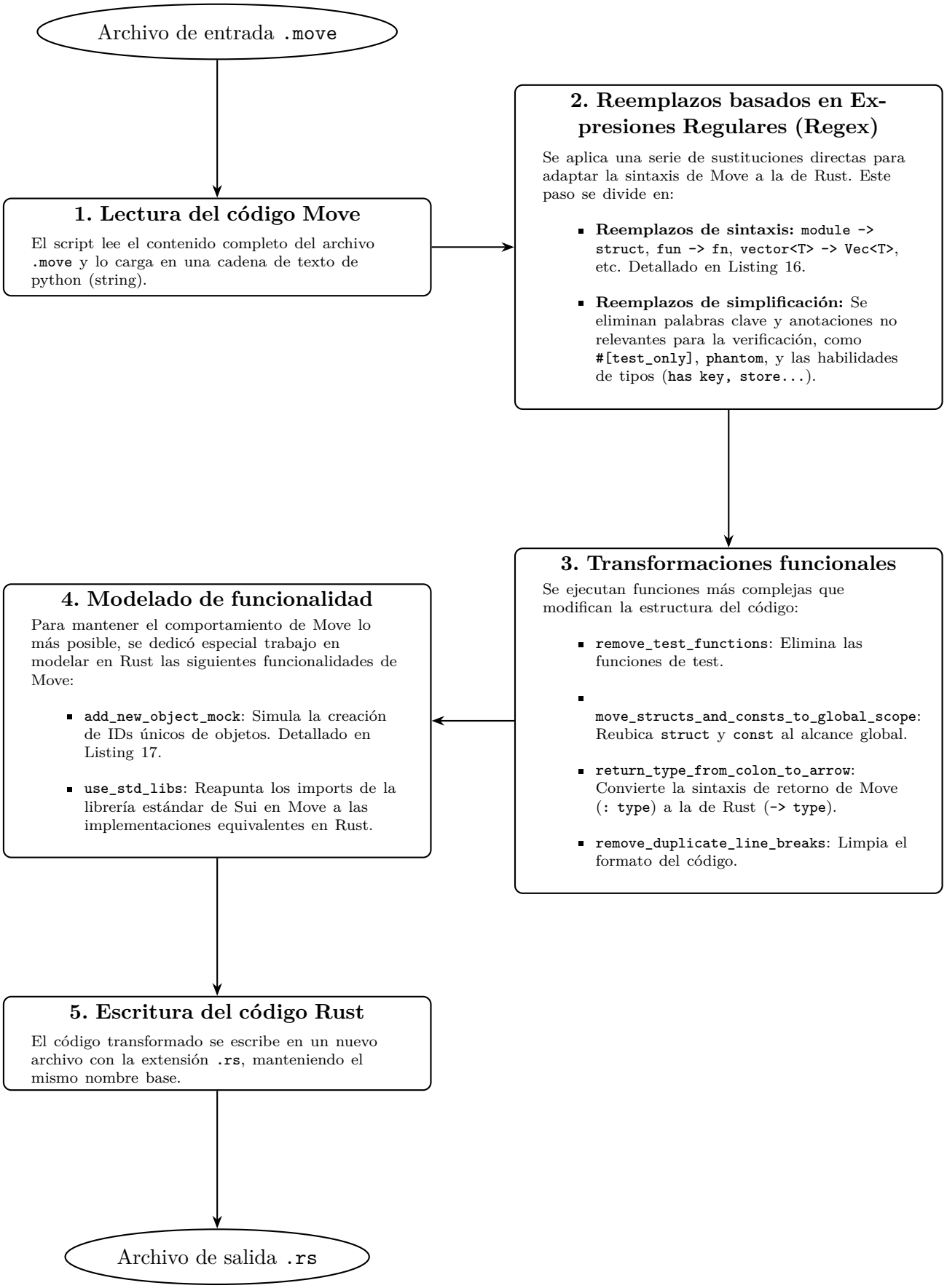

Fig. 5: Diagrama de flujo que ilustra las fases del script de traducción de Move a Rust.

```
function TraducirArchivo(ruta_archivo_move)
    string_move ← LeerArchivo(ruta_archivo_move)

    string_rust ← TransformarCodigo(string_move)

    ruta_archivo_rust ← Reemplazar(ruta_archivo_move, ".move", ".rs")
    EscribirArchivo(ruta_archivo_rust, string_rust)
end function

function TransformarCodigo(string_de_codigo_entrada)
    Let codigo ← string_de_codigo_entrada

    codigo ← MapearSintaxis(codigo)

    codigo ← SimplificarCodigo(codigo)
    codigo ← ElevarDefiniciones(codigo)
    codigo ← CorregirTiposDeRetorno(codigo)
    codigo ← AdaptarImports(codigo)

    codigo ← ModelarFuncionalidadSui(codigo)

    return codigo
end function

function MapearSintaxis(codigo)
    // Aplica un conjunto de reglas de reemplazo Regex sobre el string de código.
    for all regla en ReglasSintacticas do
        codigo ← ReemplazarRegex(codigo, regla.patron, regla.reemplazo)
    end for
    // Ejemplos de reglas: 'fun' -> 'fn', 'vector<T>' -> 'Vec<T>'.
    return codigo
end function

function SimplificarCodigo(codigo)
    // Esta función elimina varias construcciones de Move que no tienen un
    // equivalente directo en Rust o no son necesarias para la verificación.
    Let codigo_modificado ← codigo

    // 1. Eliminar las habilidades de los structs (e.g., 'has key, store...')
    patron_abilities ← has\s+(key|store|copy|drop)(,\s*(key|store|copy|drop))*\s*
    codigo_modificado ← ReemplazarRegex(codigo_modificado, patron_abilities, "")

    // 2. Eliminar la palabra clave 'phantom'
    codigo_modificado ← ReemplazarTexto(codigo_modificado, "phantom", "")

    // 3. Eliminar anotaciones de solo-test
    patron_test_only ← "#test_only.*"
```

```
    codigo_modificado ← ReemplazarRegex(codigo_modificado, patron_test_only, "")

    // 4. Eliminar funciones de test completas (bloques anotados con #[test])
    Let lineas_entrada ← SepararEnLineas(codigo_modificado)
    Let lineas_salida ← []
    Let i ← 0
    while i < Longitud(lineas_entrada) do
        if lineas_entrada[i] contiene '#[test]' then
            fin_bloque ← EncontrarFinDeBloque(lineas_entrada, i)
            i ← fin_bloque + 1
        else
            Agregar lineas_entrada[i] a lineas_salida
            i ← i + 1
        end if
    end while
    codigo_modificado ← UnirLineas(lineas_salida)

    return codigo_modificado
end function
function ElevarDefiniciones(codigo)
    // Mueve las definiciones de structs y constantes al inicio del archivo.
    Let lineas ← SepararEnLineas(codigo)
    Let lineas_elevadas ← [], lineas_normales ← []
    for all linea en lineas do
        if linea es una definición anidada de 'struct' o 'const' then
            Agregar QuitarIndentacion(linea) a lineas_elevadas
        else
            Agregar linea a lineas_normales
        end if
    end for
    return UnirLineas(lineas_elevadas) + UnirLineas(lineas_normales)
end function

function CorregirTiposDeRetorno(codigo)
    // Cambia la sintaxis de retorno de funciones de ':' a '->'.
    Let lineas ← SepararEnLineas(codigo)
    for all linea en lineas do
        if linea es una signatura de función y contiene ':' después de los paréntesis then
            indice ← EncontrarUltimo(linea, " : ")
            ReemplazarEnIndice(linea, indice, "− > ")
        end if
    end for
    return UnirLineas(lineas)
end function

function AdaptarImports(codigo)
    // Reemplaza los 'use' de Sui por implementaciones locales en Rust.
```

```
    Let lineas ← SepararEnLineas(codigo)
    Let nuevos_imports ← []
    Let lineas_sin_import ← []
    for all linea en lineas do
        if linea contiene 'use sui::coin' then
            Agregar 'use crate::sui_std::coin::Coin;' a nuevos_imports
        else if linea contiene 'use sui::transfer' then
            Agregar 'use crate::sui_std::transfer::transfer;' a nuevos_imports
            // ... y así sucesivamente para otros imports de Sui
        else
            Agregar linea a lineas_sin_import
        end if
    end for
    return UnirLineas(nuevos_imports) + UnirLineas(lineas_sin_import)
end function

function ModelarFuncionalidadSui(codigo)
    // Inyecta código Rust para simular la creación de IDs de Sui.
    Let mock_id_getter ← "código Rust del struct IdGetter..."
    codigo ← mock_id_getter + codigo
    codigo ← ReemplazarRegex(codigo, "object::new(...)", "ID_GETTER.get_new_id()")
    return codigo
end function
```

Algorithm 1: Algoritmo Detallado del Traductor de Move a Rust

En un principio el script se encarga de hacer pequeños reemplazos en la sintaxis, que demuestran la cercanía que hay entre Move y Rust. Regex fue más que suficiente para la mayoría de las diferencias entre los lenguajes (función MapearSintaxis del algoritmo 1). Algunos reemplazos fueron `public` por `pub`, `option::none/some` por `None/Some`, `vector<...>` por `Vec<...>`, como se ve, en su mayoría eran muy similares sintácticamente. La totalidad de las reglas regex pueden encontrarse en el Listing 16. En particular, requirió un proceso especial el reemplazo del tipado de retorno de las funciones "`: type`" en Move y "`-> type`" en Rust. Con un iterador sobre las lineas de código y un contador de paréntesis y llaves, fue posible identificar, con una consistencia razonable, cuáles ":" reemplazar, ya que los dos puntos son también usados para tipar los parámetros de una función tanto en Move como Rust, por lo que en ese caso no debían ser modificados (función CorregirTiposDeRetorno).
Luego, se removieron todas las funcionalidades de Move que no contribuían a la verificación del código o bien su traslado a Rust no era trivial (función SimplificarCodigo). Por ejemplo, las anotaciones `#[test only]`, `phantom`, y las habilidades de tipos (`has key, store`...) que no existen en Rust. Por último, otra de las simplificaciones fue remover las funciones de test.
A continuación se trabajó en adaptar la estructura de un módulo en Move a un script de Rust. En principio los `structs` y `const` debieron ser movidos al alcance global del archivo, de forma que el código Rust tuviera acceso (función ElevarDefiniciones). Se modeló la función de IDs globalmente únicos para los objetos de Move con un contador global, insertado en el archivo Rust, de forma que cada `object::new(...)` utilice

`ID_GETTER.get_new_id()`, en este caso preservando la correctitud (o la falta de ella) (función ModelarFuncionalidadDeSui).
Dado que los módulos de Move usan fuertemente la librería estándar de Sui, en vez de modelar/mockear sus llamadas, se tradujeron sus implementaciones. Fueron traducidas a Rust los siguientes módulos de `Sui Framework` [25]:

- `Balance`: Estructura interna de `Coin`. Mantiene un valor monetario y maneja las operaciones sobre el mismo.
- `Coin`: Modela la moneda, es un tipo paramétrico porque puede modelar tanto Sui (`Coin<SUI>`) como otras monedas. Al tener la Ability `Key`, es identificada con un ID global único.
- `Table`: Estructura similar a un `map`, donde el guardado se da en el sistema de objetos global de Sui, en vez de en la estructura en sí.
- `Transfer`: Objeto para transferir propiedad de objetos a una dirección en particular. Sirve también para "compartir" objetos, lo que provoca que cualquiera pueda accederlo y mutarlo, o para "congelarlos", que deja a un objeto irreversiblemente inmutable.

Una vez que se tuvo Sui Framework en Rust, en cada archivo Move traducido se reemplazaron los imports de Move por imports de Rust al framework en Rust localmente generado (función AdaptarImports).
Una vez convertidos los contratos a lenguaje Rust y armados los harnesses, se puede proceder a usar el verificador Kani corriendo: `cargo kani -harness <harness_function_name> -Z concrete-playback -concrete-playback=print` . Las opciones extra son para que, en caso de encontrar un input que hace fallar al programa, este sea imprimido en consola.
Se intentó verificar tres contratos Move traducidos a Rust con Kani, `simple_warrior`, `inorder` y `DSChief`, cada uno más complejo que el anterior.

4.2. Limitaciones del traductor

El proceso de traducción de contratos Move a Rust introduce ciertas simplificaciones que limitan la capacidad de verificar correctamente los contratos resultantes. En particular, existen casos en los que un contrato traducido puede ser verificado en Rust, aun cuando el contrato original en Move presente errores:

$$\exists \text{ código} \quad : \quad \text{verificado}(T_{Rust}(\text{código})) \not\Rightarrow \text{verificado}(\text{código})$$

Una de las principales simplificaciones fue la remoción del sistema de *Abilities* de Move (explicado en la sección 3.1.1.2). Esto implica que la traducción no conserva errores que dependan exclusivamente de dichas *Abilities*, como, por ejemplo, un intento de copiar un objeto que no es copiable.
El módulo `Transfer`, encargado de manipular el *ownership* de objetos en la blockchain, fue traducido a Rust con muchas de sus funciones vacías, ya que su comportamiento no puede replicarse de forma directa en Rust, cuyo sistema de *ownership* es conceptualmente distinto. Aunque esta decisión permitió verificar la lógica general de contratos que usan `Transfer`, no preserva la validez de las operaciones de transferencia en sí.

Asimismo, los valores obtenidos a partir de `TxContext` fueron sustituidos por variables inventadas. Si bien esto posibilitó continuar con la ejecución del contrato traducido, se pierden potenciales errores que dependen de un estado particular de la blockchain.
Estas limitaciones significan que, incluso con una traducción funcional, no es posible garantizar la equivalencia total entre la verificación en Rust y la verificación en el entorno original de Sui. Sin acceso a la testnet ni a un modelo realista de transacciones y estados de blockchain, la verificación en Rust siempre será parcial y no sustituye la validación en el ecosistema original.

5. EVALUACIÓN

5.1. Move Prover

Se realizó la prueba de, aunque no este soportado, utilizar la última versión de MoveProver (Sui v1.15 ~Enero 2024) en contratos actuales (Sui v1.31 ~Agosto 2024).
El siguiente contrato fue evaluado con la CLI de Sui ejecutando "`sui move prove`":

```
// Copyright (c) Mysten Labs, Inc.
// SPDX-License-Identifier: Apache-2.0

/// Demonstrates wrapping objects using the `Option` type.
module simple_warrior::example {
    public struct Sword has key, store {
        id: UID,
        strength: u8,
    }

    public struct Warrior has key, store {
        id: UID,
        sword: Option<Sword>,
    }

    /// Warrior already has a Sword equipped.
    const EAlreadyEquipped: u64 = 0;

    /// Warrior does not have a sword equipped.
    const ENotEquipped: u64 = 1;

    public fun new_sword(strength: u8, ctx: &mut TxContext): Sword {
        Sword { id: object::new(ctx), strength }
    }

    public fun new_warrior(ctx: &mut TxContext): Warrior {
        Warrior { id: object::new(ctx), sword: option::none() }
    }

    public fun equip(warrior: &mut Warrior, sword: Sword) {
        assert!(option::is_none(&warrior.sword), EAlreadyEquipped);
        option::fill(&mut warrior.sword, sword);
    }

    public fun unequip(warrior: &mut Warrior): Sword {
        assert!(option::is_some(&warrior.sword), ENotEquipped);
        option::extract(&mut warrior.sword)
    }
}
```

Listing 6: Contrato "Simple Warrior" ejemplo proveído por el repositorio de Sui.

El contrato es simple, un objeto `warrior` puede equipar o desequipar otro objeto `sword`. Si equipa cuando ya tiene una espada equipada, falla, y si desequipa sin tener una espada

equipada también.
El move prover no pudo interpretar el código de Sui y devolvió el siguiente error:

```
WARNING: the level of Move Prover support for Sui is incomplete; use at your own risk as
↪  not everything is guaranteed to work (please file an issue if an update breaks
↪  existing usage but the level of current support is limited)
error: unbound module
   ./sources/simple_warrior.move:28:21

28         Sword { id: object::new(ctx), strength }
                       ^^^^^^ Unbound module alias 'object'

error: unbound module
   ./sources/simple_warrior.move:32:23

32         Warrior { id: object::new(ctx), sword: option::none() }
                         ^^^^^^ Unbound module alias 'object'

error: unbound type
   ./sources/simple_warrior.move:18:16

18         sword: Option<Sword>,
                  ^^^^^^ Unbound type 'Option' in current scope
```

Listing 7: Error al intentar correr el prover en código de Sui actual.

Los errores se repitieron en una variedad de contratos y están relacionados con cambios no retrocompatibles en el lenguaje Move. El prover y compilador de Move de la versión v1.15 no soporta el tipo `Option` ni la invocación a `object`.
Se intentó simplificar los contratos para evadir los cambios, como por ejemplo implementar parcialmente el Sui framework con tipos como `Option`, pero la estructura de especificación que el verificador requiere, es decir, el bloque `spec`, fue removida del lenguaje base por decisiones de diseño de Sui. Esto impide escribir propiedades formales directamente en el contrato, tal como requiere el flujo de trabajo del Move Prover.
A pesar de intentar adaptar manualmente estas estructuras, la falta de soporte oficial y documentación específica para Sui llevó a abandonar la idea de implementar verificación automática utilizando el Move Prover en código Sui.

5.2. Verificación con Kani

Descartado el Move Prover y sin otra opción dentro del ecosistema Sui/Move, se consideró utilizar herramientas de otro stack. Aprovechando las cercanía entre Move y Rust, en esta sección se estudia la factibilidad de Kani Verifier aplicado a contratos de Sui.

5.2.1. Simple Warrior: Verificación con “Option”

El código de `simple_warrior` es el previamente evaluado con el Move Prover, en la sección 5.1. Para un primer contrato verificado, se buscó uno suficientemente simple. Sin embargo, el código utiliza el tipo `option`, utilizado para encapsular valores que pueden ser o bien de

un tipo en particular (`option` es un tipo paramétrico) o bien `null` [1]. Podremos ver si luego de pasar por la traducción a Rust y la verificación con Kani se mantiene el comportamiento de esta funcionalidad, también presente en Rust.
Veremos entonces los harnesses probados en el contrato `simple_warrior`, mostrado en el Listing 6.

```
#[kani::proof]
fn try_warrior_succeeds() {
    let mut w = simple_warrior__example::new_warrior();
    let mut s = simple_warrior__example::new_sword(1);
    simple_warrior__example::equip(&mut w, s);
    simple_warrior__example::unequip(&mut w);
}

#[kani::proof]
#[kani::should_panic]
fn try_warrior_unequips_empty() {
    let mut w = simple_warrior__example::new_warrior();
    let mut s = simple_warrior__example::new_sword(1);
    simple_warrior__example::unequip(&mut w);
}

#[kani::proof]
#[kani::should_panic]
fn try_warrior_equips_twice() {
    let mut w = simple_warrior__example::new_warrior();
    let mut s1 = simple_warrior__example::new_sword(1);
    let mut s2 = simple_warrior__example::new_sword(1);
    simple_warrior__example::equip(&mut w, s1);
    simple_warrior__example::equip(&mut w, s2);
}
```

Listing 8: Harness donde se prueban usos comunes del módulo Simple Warrior.

El primer harness (`try_warrior_succeeds`) representa el uso correcto del guerrero (`warrior`), equipar y desequipar la espada (`sword`). Este no presenta errores en el verificador, como esperado.
Luego en el segundo (`try_warrior_unequips_empty`) se desequipa un guerrero sin espada. El verificador detecta correctamente la falla, provocada por la siguiente linea en la implementación Move: `assert!(option::is_some(&warrior.sword), ENotEquipped);`
Finalmente, en el último harness (`try_warrior_equips_twice`) se prueba el error levantado al equipar dos veces una espada, sin desequipar en el medio. Una vez más el verificador encuentra correctamente esta falla. Podemos ver en el Listing 9 el detalle de Kani, en este caso los `concrete_vals` son vacíos ya que no generamos ninguna variable no deterministica para la verificación.

```
VERIFICATION:- SUCCESSFUL (encountered one or more panics as expected)
Verification Time: 5.9072447s
```

[1] Documentación de `option` en Move: https://move-book.com/move-basics/option.html y en Rust: https://doc.rust-lang.org/std/option/

````
Concrete playback unit test for
↪ `harnesses::simple_warrior_harnesses::try_warrior_equips_twice`:
```
#[test]
fn kani_concrete_playback_try_warrior_equips_twice_13502110605057177274() {
    let concrete_vals: Vec<Vec<u8>> = vec![
    ];
    kani::concrete_playback_run(concrete_vals, try_warrior_equips_twice);
}
```
````

Listing 9: Salida de Kani Verifier al correr los harnesses de `simple_warrior`.

5.2.2. Inorder: Llamadas no determinísticas

El siguiente contrato es un objeto con tres métodos (`a`, `b`, y `c`) que no hacen más que registrar su propia ejecución para mantener el contrato del objeto: no se puede llamar a `b` antes que `a`, ni a `c` antes que `b`. El invariante es representado en la figura 6, donde se puede ver un grafo con los estados del objeto en cada nodo, y las llamadas a métodos permitidas para cada estado en cada eje. Este contrato fue extraido del trabajo previamente mencionado de Vera Bogdanich [9].

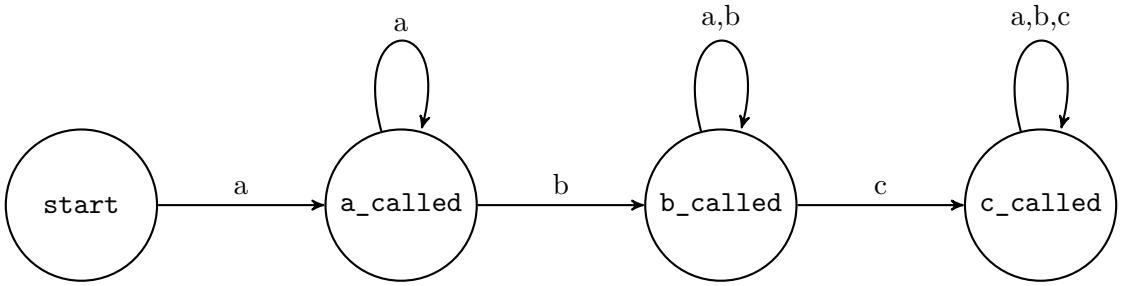

Fig. 6: Grafo de estados que representa la secuencia de llamadas permitidas en `inorder`.

Probaremos Kani en este caso, esperando que detecte correctamente las secuencias inválidas de llamadas a métodos.

```
1  module inorder::inorder {
2
3      const EACallRequired: u64 = 0;
4      const EBCallRequired: u64 = 1;
5      const ECCallRequired: u64 = 2;
6
7      public struct CallRegistry has key {
8          id: UID,
9          num_calls: u64,
10         a_called: bool,
11         b_called: bool,
12         c_called: bool,
13     }
14
15     fun init(ctx: &mut TxContext) {
```

```
16          let call_registry = CallRegistry {
17              id: object::new(ctx),
18              num_calls: 0,
19              a_called: false,
20              b_called: false,
21              c_called: false,
22          };
23
24          transfer::share_object(call_registry);
25      }
26
27      public fun a(call_registry: &mut CallRegistry) {
28          call_registry.a_called = true;
29          call_registry.num_calls++;
30      }
31
32      public fun b(call_registry: &mut CallRegistry) {
33          assert!(call_registry.a_called, EACallRequired);
34
35          if call_registry.b_called {
36              call_registry.num_calls = 0;
37              return;
38          }
39
40          call_registry.b_called = true;
41          call_registry.num_calls++;
42
43      }
44
45      public fun c(call_registry: &mut CallRegistry): u64 {
46          assert!(call_registry.a_called, EACallRequired);
47          assert!(call_registry.b_called, EBCallRequired);
48
49          call_registry.c_called = true;
50          call_registry.num_calls++;
51
52          return 3;
53      }
54  }
```

Listing 10: Código de inorder.

Si lo traducimos a Rust, podemos ver el resultado en el listing 11.

```
1   pub struct CallRegistry {
2       id: u8,
3       num_calls: u64,
4       a_called: bool,
5       b_called: bool,
6       c_called: bool,
7   }
8
9   const EInvariantBroken: u64 = 3;
10
11  const ECCallRequired: u64 = 2;
12
```

```
13 const EBCallRequired: u64 = 1;
14
15 const EACallRequired: u64 = 0;
16
17 use std::sync::LazyLock;
18
19 pub struct IdGetter {
20     current_id: std::sync::Mutex<u8>,
21 }
22
23 impl IdGetter {
24     pub fn new() -> Self {
25         IdGetter {
26             current_id: std::sync::Mutex::new(0),
27         }
28     }
29
30     pub fn get_new_id(&self) -> u8 {
31         let mut id = self.current_id.lock().unwrap();
32         *id += 1;
33         *id
34     }
35 }
36
37 // Use LazyLock to initialize ID_GETTER
38 pub static ID_GETTER: LazyLock<IdGetter> = LazyLock::new(|| IdGetter::new());
39
40 pub struct inorder__inorder {}
41 impl inorder__inorder {
42
43     pub fn init() -> CallRegistry {
44         let call_registry = CallRegistry {
45             id: ID_GETTER.get_new_id(),
46             num_calls: 0,
47             a_called: false,
48             b_called: false,
49             c_called: false,
50         };
51
52         call_registry
53     }
54
55     pub fn a(call_registry: &mut CallRegistry) {
56         call_registry.a_called = true;
57         call_registry.num_calls+=1;
58     }
59
60     pub fn b(call_registry: &mut CallRegistry) {
61         assert!(call_registry.a_called, "{}", EACallRequired);
62
63         if call_registry.b_called {
64             call_registry.num_calls = 0;
65             return;
66         }
67
68         call_registry.b_called = true;
69         call_registry.num_calls+=1;
70     }
```

```
71
72      pub fn c(call_registry: &mut CallRegistry) -> u64 {
73          assert!(call_registry.a_called, "{}", EACallRequired);
74          assert!(call_registry.b_called, "{}", EBCallRequired);
75
76          call_registry.c_called = true;
77          call_registry.num_calls+=1;
78
79          0
80      }
81  }
```

Listing 11: Código de inorder traducido a Rust.

Para este caso se generó un harness que, de forma no determinística, ejecute los métodos de inorder en cualquier orden (Listing 12). Es posible hacer esto aprovechando la forma en la que se generan variables no determinísticas en el model checking de Kani.
Kani explora todos los valores posibles para `x`, la variable generada con `kani::any()`. Para los usos de este harness se limita, también mediante Kani, los valores de `x` a aquellos menores a 3. Como `x` es un *unsigned integer*, esto permite tres valores, 0, 1 o 2, que mediante la estructura `match` se lleva a los casos posibles de llamadas a métodos de `inorder`.
Por último, se agregó `#[kani::unwind(5)]` para limitar las ejecuciones del loop, ya que no interesan las secuencias muy largas de llamadas de métodos.

```
1   #[kani::proof]
2   #[kani::unwind(5)]
3   pub fn try_generic_inorder() {
4       let mut call_registry = inorder__inorder::init();
5       while true {
6           let x: u8 = kani::any();
7           kani::assume(x < 3);
8           match x {
9               0=> inorder__inorder::a(&mut call_registry),
10              1=> inorder__inorder::b(&mut call_registry),
11              2=> {inorder__inorder::c(&mut call_registry);},
12              _=>{}
13          }
14      }
15
16  }
```

Listing 12: Código de inorder traducido a Rust.

La salida del verificador nos permite ver las secuencias de llamadas a métodos que hicieron fallar el harness. El vector `concrete_vals` son las `x`s generadas con `kani::any()`, por lo que un 0 es una llamada al método `a`, 1 es `b` y 2 es `c`:

```
VERIFICATION:- FAILED
[Kani] info: Verification output shows one or more unwinding failures.
[Kani] tip: Consider increasing the unwinding value or disabling
↪  `--unwinding-assertions`.
Verification Time: 17.46555s
```

````
Concrete playback unit test for `harnesses::inorder_harnesses::try_generic_inorder`:
```
#[test]
fn kani_concrete_playback_try_generic_inorder_337113188998539698() {
    let concrete_vals: Vec<Vec<u8>> = vec![
        // 2
        vec![2],
    ];
    kani::concrete_playback_run(concrete_vals, try_generic_inorder);
}
```
Concrete playback unit test for `harnesses::inorder_harnesses::try_generic_inorder`:
```
#[test]
fn kani_concrete_playback_try_generic_inorder_12045961763212337033() {
    let concrete_vals: Vec<Vec<u8>> = vec![
        // 0
        vec![0],
        // 2
        vec![2],
    ];
    kani::concrete_playback_run(concrete_vals, try_generic_inorder);
}
```
Concrete playback unit test for `harnesses::inorder_harnesses::try_generic_inorder`:
```
#[test]
fn kani_concrete_playback_try_generic_inorder_16718444418884853822() {
    let concrete_vals: Vec<Vec<u8>> = vec![
        // 1
        vec![1],
    ];
    kani::concrete_playback_run(concrete_vals, try_generic_inorder);
}
```
````

Listing 13: Salida de Kani Verifier al correr los harnesses de `inorder`.

Los casos de error, expresados con los valores concretos de cada `x` generada, son: [2], [0, 2], [1]. Esto significa que `inorder` falló al llamar primero a `c`, primero a `b`, o en la secuencia [`a`, `c`]. De aquí podemos deducir, calculando el complemento, que todas las secuencias que no hacen fallar a `inorder` tienen al menos una llamada a `a` antes que a `b`, y al menos una llamada a `b` antes que a `c`, lo cual coincide con el invariante conocido.

5.2.3. DSChief: Adaptación de objetos a Kani

Finalmente se probó con el contrato `DSChief`, que modela una votación. Como el anterior contrato en la sección 5.2.2, este también proviene del trabajo de Bogdanich [9]. La idea de esta última prueba es estudiar el comportamiento de Kani con estructuras dinámicas como los mappings, usados en este contrato, que en Move son representados con `Table`s, traducidos a Rust (internamente utilizando `HashMap`s de la librería estándar).

```
module SimpleDSChief::SimpleDSChief {

    const EAddShouldBeGreater: u64 = 0;
    const ESubShouldBeSmaller: u64 = 1;

    use sui::table::Table;

    public struct Address {
        id: u64,
    }


    public struct DSChief has key {
        id: UID,
        slates: Table<u64, Address>,
        votes: Table<Address, u64>,
        approvals: Table<Address, u64>,
        deposits: Table<Address, u64>,
    }

    public fun lock(chief: &mut DSChief, sender: Address, wad: u64) {
        chief.deposits[sender] = add(chief.deposits[sender], wad);
        addWeight(chief, wad, chief.votes[sender]);
    }

    public fun free(chief: &mut DSChief, sender: Address, wad: u64) {
        chief.deposits[sender] = sub(chief.deposits[sender], wad);
        subWeight(chief, wad, chief.votes[sender]);
    }

    public fun voteYays(chief: &mut DSChief, sender: Address, yay: Address) -> u64 {
        let slate: u64 = etch(chief, yay);
        voteSlate(chief, slate);

        return slate
    }

    public fun etch(chief: &mut DSChief, yay: Address) -> u64 {
        let slate = yay.id; // way around hashing
        chief.slates[slate] = yay;
        return slate
    }

    public fun voteSlate(chief: &mut DSChief, sender: Address, slate: u64) {
        let weight: u64 = chief.deposits[sender];
        subWeight(chief, weighyt, chief.votes[sender]);
        chief.votes[sender] = slate;
        addWeight(chief, weight, chief.votes[sender]);
    }

    public fun addWeight(chief: &mut DSChief, weight: u64, slate: u64) {
        let yay: Address = chief.slates[slate];
        chief.approvals[yay] = add(chief.approvals[yay], weight);
    }

    public fun subWeight(chief: &mut DSChief, weight: u64, slate: u64) {
        let yay: Address = chief.slates[slate];
        chief.approvals[yay] = sub(chief.approvals[yay], weight);
```

```
59      }
60
61      public fun add(x: u64, y: u64) -> u64 {
62          let z: u64 = x + y;
63          assert!(z >= x, EAddShouldBeGreater);
64
65          return z;
66      }
67
68      public fun sub(x: u64, y: u64) -> u64 {
69          let z: u64 = x - y;
70          assert!(z <= x, ESubShouldBeSmaller);
71
72          return z;
73      }
74  }
```

Listing 14: Código de DSChief.

Para verificar un contrato con tantos objetos particulares al archivo, es necesario darle a Kani instrucciones para construir una instancia genérica del objeto, es decir, especificar el método `kani::any()` para estos objetos no nativos [2]. Además, en intentos de reducir el tiempo de ejecución se utilizaron llamadas a `kani::assume()` para reducir el espacio de búsqueda. El código resultante se encuentra en el Listing 18.
No fue posible utilizar Kani en este contrato, la verificación no terminaba y se continuaba escribiendo en consola un mensaje repetido. Para intentar resolver esto, se trabajó en simplificar el esfuerzo hecho por el verificador. Los intentos contemplaron:

- Mayor especificación para los objetos modelados en Kani. Se restringieron los valores de variables no deterministicas con `kani::assume(value < limit)`.
- Menor complejidad en el contrato. Se probó removiendo partes no críticas del contrato.
- Modelado más simple del `HashMap`. Lo nuevo en este contrato fue esta estructura. Se intentó implementando un `HashMap` que dentro funcionara con arreglos acotados o con listas enlazadas.

Estos esfuerzos no posibilitaron la verificación con Kani. Si bien podemos atribuir la falla a causas como un error en la traducción a Rust, o que Kani tiene mucha funcionalidad aún en etapa experimental, no podemos justificar una razón concreta. El código fue exitosamente compilado y probado manualmente en Rust, pero no se descarta que en la traducción se haya introducido un bug que provoque el error en Kani.

```
Unwinding loop _RNvXs3_NtNtCs3LsjwiATwoX_4core4hash3sipINtB5_6HasherNtB5_11Sip13RoundsE
↪  NtB7_6Hasher5writeCsb7Lo72Qdc8S_12move_to_rust.0 iteration 1 file
↪  /github/home/.rustup/toolchains/nightly-2024-07-01-x86_64-unknown-linux-gnu/lib/rus
↪  tlib/src/rust/library/core/src/hash/sip.rs line 286 column 9 function
↪  <core::hash::sip::Hasher<core::hash::sip::Sip13Rounds> as std::hash::Hasher>::write
↪  thread 0
```

[2] Descripción de este proceso en la documentación de Kani: https://model-checking.github.io/kani/tutorial-nondeterministic-variables.html#custom-nondeterministic-types

Listing 15: Salida de Kani Verifier al correr los harnesses de `DSChief`.

6. CONCLUSIONES Y TRABAJO FUTURO

En el presente trabajo se planteó como objetivo determinar la factibilidad del uso de herramientas de verificación automática en Smart Contracts escritos en Move para la blockchain Sui.
Al momento de la investigación, el apartado de verificación en el ecosistema Sui no se encontraba priorizado por los mantenedores del lenguaje, con lo cual no es factible la utilización del verificador anteriormente oficial de Move, el Move Prover (ni tampoco son simples los ajustes necesarios para su funcionamiento como se discutió en la sección 5.1).
Sin embargo, el camino alternativo de traducir Move a Rust y aprovechar las herramientas en este otro lenguaje tuvo un éxito parcial pero considerable. Dada una limitación sobre los objetos que usa el Smart Contract, y utilizando una traducción de la librería estándar de Move on Sui a Rust (sección 4.1), es posible la verificación estática de un subconjunto de contratos.
Resta determinar si es posible ajustar el verificador automático Kani para resolver los problemas de no terminación de la sección 5.2.3, o bien si existe una parte del código que, si hubiera sido especificada y su ejecución salteada, se hubiera resuelto el problema que impedía la verificación.
Por otro lado, lo discutido aquí abre la puerta a otros estudios vinculados, como la aplicación de la técnica de análisis EPA (Enabledness Program Abstractions)[11] a los contratos en Sui, el estudio del reciente Sui Prover[26] inspirado en Move Prover, o un análisis sobre las diferencias entre la verificación en lenguajes como Move comparado a Solidity.

7. ANEXO

```
syntax_replacements = [
    (r'module\s+([a-zA-Z0-9_]+)::([a-zA-Z0-9_]+)', r'pub struct \1__\2 {}\nimpl \1__\2'), # "Module address::name"
    (r'resource\s+struct', r'struct'), # Structs
    (r'public', r'pub'), # Public to pub
    (r'(pub struct (\w+)(<[^>]+>)?(?: has (copy|key|store|drop)(, (copy|key|store|drop))*)?\s*{)', r'pub struct \2 {'), # Remove type abilities
    (r'\(package\)', r''), # Remove 'package' scope
    (r'entry fun', r'fun'), # Remove 'Entry'
    (r'fun', r'fn'), # Fun to fn
    (r'\+\+', r'+=1'), # ++ not in rust syntaxis
    (r'ascii', r'string'), # No need for ascii
    (r'string::String', r'String'), # Rename of string type
    (r'option::is_some\(&(\w+\.\w+)\)', r'\1.is_some()'), # Option is_some
    (r'option::is_none\(&(\w+\.\w+)\)', r'\1.is_none()'), # Option is_none
    (r'option::fill\(&mut (\w+\.\w+), (\w+)\)', r'assert!(\1.replace(\2).is_none())'), # Option fill (assignment if is None, otherwise fail)
    (r'option::extract\(&mut (\w+\.\w+)\)', r'\1.take().unwrap()'), # Option extract (take)
    (r'option::none\(\)', r'None'), # Option None
    (r'option::some\(\)', r'Some'), # Option Some
    (r'assert!\((.+?),\s*(.+?)\)', r'assert!(\1, "{}", \2)'), # Assert with string literal
    (r'ctx: &mut TxContext(,?)', r''), # Remove TxContext. TODO: Might need to model this.
    (r'phantom ', r''), # Remove phantom
    (r'Balance<[^>]+>', r'Balance'), # Balance type not parametric.
    (r'Coin<[^>]+>', r'Coin'), # Coin type not parametric.
    (r'Supply<[^>]+>', r'Supply'), # Supply type not parametric.
    (r'TreasuryCap<[^>]+>', r'TreasuryCap'), # TreasuryCap type not parametric.
    (r'CoinMetadata<[^>]+>', r'CoinMetadata'), # CoinMetadata type not parametric.
    (r'Url', r'String'), # Use strings for URLs.
    (r'UID', r'ID'), # Transform all to ID an then...
    (r'ID', r'u8'), # Use u8 for ID.
    (r'address', r'String'), # Use string for address type.
    (r'vector<([^>]+)>', r'Vec<\1>'), # Rename to rust vector type.
    (r'return (.+?)(;)?', r'\1'), # Return in rust
    (r'transfer::share_object\((.+?)\)(;)?', r'\1'), # Instead of move transfer, return
    (r'fn init', r'pub fn init'), # Set init as public
    (r'VecMap', r'Map'), # Move map
]
```

Listing 16: Los patrones regex y los reemplazos correspondientes.

```
def add_new_object_mock(code):
    """Replaces calls to object::new(ctx) which assigns a specific UID in the blockchain
    with calls to a local counter."""
    id_getter_code = '''
use std::sync::LazyLock;

```

```
pub struct IdGetter {
    current_id: std::sync::Mutex<u8>,
}

impl IdGetter {
    pub fn new() -> Self {
        IdGetter {
            current_id: std::sync::Mutex::new(0),
        }
    }

    pub fn get_new_id(&self) -> u8 {
        let mut id = self.current_id.lock().unwrap();
        *id += 1;
        *id
    }
}

// Use LazyLock to initialize ID_GETTER
pub static ID_GETTER: LazyLock<IdGetter> = LazyLock::new(|| IdGetter::new());

'''
    code = id_getter_code + code
    code = re.sub(r'object::new\(\w*\)', 'ID_GETTER.get_new_id()', code,
    ↪  flags=re.MULTILINE)
    return code
```

Listing 17: Workaround de la funcionalidad `object::new(...)` en Move on Sui.

```
fn bounded_any() -> u64 {
    kani::any_where(|x: &u64| *x == 0 || *x == 1 || *x == 2)
}

impl Arbitrary for Address {
    // Custom method to generate arbitrary `Address`
    fn any() -> Self {
        // Generate arbitrary `u64` value for `id`
        let id: u64 = bounded_any();
        Address {
            id: id,
        }
    }
}

pub fn arbitrary_hashmap<K, V>() -> HashMap<K, V>
where
    K: Arbitrary + Eq + std::hash::Hash + Clone,
    V: Arbitrary + Clone,
{
    let mut map = HashMap::new();
    let size: u8 = bounded_any() as u8;
    kani::assume(size < 3); // Limit the map size

    for _ in 0..size {
        let key: K = kani::any();
        let value: V = kani::any();
```

```
28          map.insert(key, value);
29      }
30
31      map
32  }
33
34  impl<K: Key + Arbitrary + Clone, V: Arbitrary + Clone> Arbitrary for Table<K, V> {
35      // Custom method to generate arbitrary `Address`
36      fn any() -> Self {
37          // Generate arbitrary `u64` value for `id`
38          let map = arbitrary_hashmap();
39          let size = map.len();
40          let id = bounded_any() as u8;
41          Table {
42              id: id,
43              map: map,
44              size: size as u8, // Cast is safe as size is less than 10
45          }
46      }
47  }
48
49  impl Arbitrary for DSChief {
50      // Custom method to generate arbitrary `Address`
51      fn any() -> Self {
52          let mut common_keys = HashSet::new();
53          let size: u8 = kani::any_where(|x| *x > 0 && *x <= 5); // Prevent excessive keys
54
55          for _ in 1..size { // No empty cases
56              common_keys.insert(Address::any());
57          }
58
59          let mut votes = table::new();
60          let mut approvals = table::new();
61          let mut deposits = table::new();
62
63          for key in &common_keys {
64              table::add(&mut votes, key.clone(), bounded_any());
65              table::add(&mut approvals, key.clone(), bounded_any());
66              table::add(&mut deposits, key.clone(), bounded_any());
67          }
68
69          let id: u8 = bounded_any() as u8;
70
71          // Generate arbitrary `u64` value for `id`
72          let dschief = DSChief {
73              id: id,
74              slates: kani::any(),
75              votes: votes,
76              approvals: approvals,
77              deposits: deposits,
78          };
79          for key in dschief.slates.map.keys() {
80              kani::assume(*key == 0 || *key == 1 || *key == 2);
81          };
82
83
84          dschief
85      }
```

```
}

pub fn get_random_key<K, V>(table: &Table<K, V>) -> K
where
    K: Key + Arbitrary + Clone, V: Arbitrary + Clone
{
    let keys: Vec<K> = table.map.keys().cloned().collect();
    let idx: usize = kani::any_where(|x| *x < keys.len()); // Ensure the index is valid

    keys[idx].clone()
}


//#[kani::unwind(3)]
#[kani::proof]
pub fn try_generic_dschief() {
    let mut dschief: DSChief = kani::any();

    run_loop_iteration(&mut dschief);
}


pub fn run_loop_iteration(dschief: &mut DSChief) {
    let address: Address = get_random_key(&dschief.deposits);
    let address2: Address = get_random_key(&dschief.deposits);
    let x: u8 = bounded_any() as u8;
    let wad: u64 = bounded_any();

    match x {
        0=> SimpleDSChief__SimpleDSChief::lock(dschief, &address, wad),
        1=> SimpleDSChief__SimpleDSChief::free(dschief, &address, wad),
        2=> {SimpleDSChief__SimpleDSChief::voteYays(dschief, &address, address2);},
        _=>{}
    }
}
```

Listing 18: Harness de dschief.

BIBLIOGRAFíA

[1] OWASP Foundation. *OWASP Smart Contract Top 10 (2025)*. 2025. URL: https://owasp.org/www-project-smart-contract-top-10/.

[2] Thomas Bourveau, Janja Brendel y Jordan Schoenfeld. "Decentralized Finance (DeFi) assurance: early evidence". En: *Review of Accounting Studies* 29.3 (2024), págs. 2209-2253. ISSN: 1573-7136. DOI: 10.1007/s11142-024-09834-8. URL: https://doi.org/10.1007/s11142-024-09834-8.

[3] Diem Association. *The Move Programming Language*. 2020. URL: https://github.com/move-language/move/.

[4] Mysten Labs. *Sui Blockchain*. 2024. URL: https://sui.io/.

[5] Diem Association. *Move Prover*. 2020. URL: https://github.com/move-language/move/tree/main/language/move-prover.

[6] Kani Verifier Contributors. *Kani Verifier*. Ver. 0.55. Sep. de 2024. URL: https://github.com/model-checking/kani.

[7] Yuepeng Wang et al. "Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain". En: *Verified Software. Theories, Tools, and Experiments*. Ed. por Supratik Chakraborty y Jorge A. Navas. Cham: Springer International Publishing, 2020, págs. 87-106. ISBN: 978-3-030-41600-3.

[8] Leonardo Alt et al. "SolCMC: Solidity Compiler's Model Checker". En: *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*. Haifa, Israel: Springer-Verlag, 2022, págs. 325-338. ISBN: 978-3-031-13184-4. DOI: 10.1007/978-3-031-13185-1_16. URL: https://doi.org/10.1007/978-3-031-13185-1_16.

[9] Vera Bogdanich Espina. "Ethereum Smart Contracts Verification: A Survey and a Prototype Tool". Tesis de Grado. Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales, 2019.

[10] Mark Mossberg et al. "Manticore: a user-friendly symbolic execution framework for binaries and smart contracts". En: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ASE '19. San Diego, California: IEEE Press, 2020, págs. 1186-1189. ISBN: 9781728125084. DOI: 10.1109/ASE.2019.00133. URL: https://doi.org/10.1109/ASE.2019.00133.

[11] Guido De Caso et al. "Enabledness-based program abstractions for behavior validation". En: *ACM Trans. Softw. Eng. Methodol.* 22.3 (jul. de 2013). ISSN: 1049-331X. DOI: 10.1145/2491509.2491519. URL: https://doi.org/10.1145/2491509.2491519.

[12] Javier Godoy et al. "Predicate abstractions for smart contract validation". En: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, págs. 289-299. ISBN: 9781450394666. DOI: 10.1145/3550355.3552462. URL: https://doi.org/10.1145/3550355.3552462.

[13] Edén Torres. "Generador de Abstracciones para Smart Contracts". Tesis de Grado. Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales, 2023.

[14] Daniel Wappner. "Construcción de Abstracciones de comportamiento para contratos inteligentes mediante ejecución simbólica". Tesis de Grado. Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales, 2024.

[15] Ian Grinspan. "Generador de Abstracciones de Comportamiento para Contratos Inteligentes mediante Fuzzing". Tesis de Grado. Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales, 2025.

[16] Gustavo Grieco et al. "Echidna: effective, usable, and fast fuzzing for smart contracts". En: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2020, págs. 557-560. ISBN: 9781450380089. DOI: 10.1145/3395363.3404366. URL: https://doi.org/10.1145/3395363.3404366.

[17] David Dill et al. *Fast and Reliable Formal Verification of Smart Contracts with the Move Prover*. 2022. arXiv: 2110.08362 [cs.PL]. URL: https://arxiv.org/abs/2110.08362.

[18] Doron Peled Edmund M. Clarke Orna Grumberg. *Model Checking*. MIT Press, 1999. ISBN: 9780262032704.

[19] C. A. R. Hoare. "An axiomatic basis for computer programming". En: 12.10 (1969), págs. 576-580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[20] George Danezis et al. *Mysticeti: Low-Latency DAG Consensus with Fast Commit Path*. Inf. téc. Mysten Labs, 2023. URL: https://docs.sui.io/assets/files/mysticeti-b7bf6bc1e0df4e363fd971dc51e11ac0.pdf.

[21] The Move Book Contributors. *Language for Digital Assets*. 2024. URL: https://move-book.com/object/digital-assets.html.

[22] Boogie Project. *Boogie: An Intermediate Verification Language*. 2024. URL: https://github.com/boogie-org/boogie.

[23] Leonardo De Moura y Nikolaj Bjørner. "Z3: An efficient SMT solver". En: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, págs. 337-340.

[24] Leo Mansini. *move_to_rust: Transpiler Script*. URL: https://github.com/LeoMansini/move_to_rust/blob/master/src/transpiler.py.

[25] Mysten Labs. *Sui Framework (GitHub Repository)*. https://github.com/MystenLabs/sui/tree/main/crates/sui-framework/packages/sui-framework. 2024.

[26] Asymptotic Technologies. *Sui Formal Verification*. 2024. URL: https://info.asymptotic.tech/sui-prover.