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

Generador de abstracciones para smart contracts

Tesis de Licenciatura en Ciencias de la Computación

Edén Torres

Director: Sebastian Uchitiel

Codirector: Javier Godoy

Buenos Aires, 2023

RESUMEN

Vamos a abordar la problemática de validar y verificar la corrección de los contratos inteligentes, programas que se ejecutan en una blockchain y pueden manejar recursos de alto valor. Debido a la falta de especificaciones claras del comportamiento esperado y al entorno de ejecución concurrente en el que funcionan, validar la corrección de estos contratos es una tarea compleja. Una manera de facilitar la comprensión del comportamiento de los contratos es construyendo máquinas de estado, pero esta técnica se realiza de manera manual y requiere la traducción del código a un lenguaje de modelado. Por lo tanto, se propone desarrollar una herramienta capaz de construir automáticamente abstracciones para contratos inteligentes para la red Ethereum utilizando ideas similares a las de un estudio anterior y utilizando un analizador estático de programas Solidity. Para esto, se utilizará VeriSol, una herramienta de análisis estático desarrollada por Microsoft Research que verifica la correctitud de las aserciones en un contrato dado.

Palabras clave: contratos inteligentes, blockchain, validación, verificación, máquinas de estado, análisis estático, Solidity, Verisol.

RESUMEN

We will address the challenge of validating and verifying the correctness of smart contracts, which are programs that run on a blockchain and can handle high-value resources. Due to the lack of clear specifications of the expected behavior and the concurrent execution environment in which they operate, validating the correctness of these contracts is a complex task. One way to facilitate the understanding of contract behavior is by building state machines, but this technique is done manually and requires translation of the code into a modeling language. Therefore, we propose to develop a tool capable of automatically building abstractions for smart contracts on the Ethereum network using ideas similar to a previous study and using a Solidity static program analyzer. For this, we will use VeriSol, a static analysis tool developed by Microsoft Research that verifies the correctness of assertions in a given contract.

Keywords: smart contracts, blockchain, validation, verification, state machines, static analysis, Solidity, VeriSol.

Índice general

1. Introducción 1
2. Preliminares 2
   - 2.1. Enabledness-presserving abstractions (EPAs) 2
   - 2.2. Blockchain 3
   - 2.3. VeriSol 5
   - 2.4. Trabajos relacionados 6
3. Desarrollo 8
   - 3.1. EPA 8
   - 3.2. States 10
   - 3.3. Creación de la tool 11
   - 3.4. Limitaciones 15
   - 3.5. Optimización 16
4. Validación 19
   - 4.1. Hello Blockchain 19
   - 4.2. Asset Transfer 23
   - 4.3. Auction 29
   - 4.4. CrowdFunding 34
   - 4.5. Rock Papper Scissors 38
   - 4.6. EPXCrowdSale 41
5. Experimentación 45
   - 5.1. Benchmark Azure 45
   - 5.2. Benchmak Smartpulse 52
   - 5.3. Conclusiones de las optimizaciones 52
6. Trabajo Futuro 55
7. Conclusiones 56

1. Introducción

Los contratos inteligentes son programas que se ejecutan en una blockchain y pueden manejar recursos de alto valor. Como cualquier otro programa, pueden contener errores, ya sea porque el comportamiento del código no coincide con lo que el diseñador del contrato especificó o porque el comportamiento especificado por el diseñador es diferente al que espera el dueño del contrato.

La validación y verificación de estos contratos es complicada por varias razones. En primer lugar, normalmente no hay especificaciones claras (formales o informales) del comportamiento esperado. Además, estos contratos funcionan en entornos de ejecución concurrente, lo que significa que no solo es importante verificar la corrección de cada función del contrato, sino también todas las posibles secuencias de llamados a estas mismas funciones. En segundo lugar, es importante destacar que los contratos son inmutables una vez desplegados. Esta característica es un factor de gran importancia a tener en cuenta ya que estos contratos manejan activos de gran valor económico. Cualquier error o fallo podría resultar en la pérdida de millones de dólares.

Una manera de facilitar la comprensión del comportamiento de los contratos es construir máquinas de estado que proporcionen una visión abstracta pero compacta del espacio de estados del contrato y las posibles operaciones que generan cambios de estado. Sin embargo, estas abstracciones pueden ser imprecisas, lo que significa que el modelo resultante puede tener secuencias de operaciones de más o de menos.

En un estudio anterior [1], se demostró la utilidad de estas abstracciones para auditores de contratos que deben realizar tareas de entendimiento, verificación y validación. Sin embargo, la construcción de estas abstracciones se realiza de manera manual, lo que requiere la traducción del código a un lenguaje de modelado que luego puede ser procesado por una herramienta llamada `Alloy` [2]

En otro estudio [3], se presenta una técnica automática que genera abstracciones similares a las utilizadas en [1] a partir de código C, utilizando un analizador automático (estático) de código C.

El objetivo de este estudio es desarrollar una herramienta capaz de construir automáticamente abstracciones como las descritas en [3] para smart contracts, utilizando ideas como las de [1] y utilizando un analizador estático de programas `Solidity` (un lenguaje para smart contracts).

Como analizador de programas estático de smart contracts se va a usar `VeriSol` citada en [4]. Esta herramienta es un proyecto de Microsoft Research para la creación de prototipos de un sistema formal de verificación y análisis para contratos inteligentes desarrollado en el popular lenguaje de programación `Solidity`. Se basa en la traducción de programas en el lenguaje de `Solidity` a programas en el lenguaje de verificación intermedio de Boogie y luego aprovechando y ampliando la cadena de herramientas de verificación para los programas de Boogie. En resumen, la herramienta `VeriSol` va a verificar la correctitud de los assertions en un contrato dado.

Para lograr este objetivo se propone generar abstracciones correctas del Benchmark de `Microsoft Azure` [5] y el Benchmark de `Smartpulse` [6]. El benchmark de `Azure` se utilizó en el estudio de `VeriSol` [4] y el de Smartpulse en el estudio de `Smartpulse`. Ambos contienen implementaciones de contratos inteligentes con una descripción detallada de su comportamiento previsto. El primer Benchmark contiene un diagrama de transición de estado producido manualmente que describe el comportamiento esperado del contrato.

2. Preliminares

Antes de presentar el proceso de construcción automática de abstracciones, es fundamental tener un conocimiento profundo de los aspectos técnicos que las sustentan. Esto implica establecer una notación rigurosa y precisa, y por tanto, se sugiere hacer referencia al trabajo [3]. Es imprescindible comprender estos aspectos técnicos para tener una comprensión completa de la técnica de los algoritmos de abstracción automática.

2.1. Enabledness-presserving abstractions (EPAs)

Se puede representar el protocolo de una clase mediante una máquina de estados. La máquina de estados tiene un estado inicial y a partir de este, podemos realizar acciones que nos llevan a otros estados. Por ejemplo, si pensamos en una pila, el estado inicial es cuando la pila está vacía. Luego, se puede llenarla con datos y vaciarla de a un elemento. El espacio de estados para representar este protocolo es potencialmente infinito.

Para resolver el problema de la representación de un espacio potencialmente infinito de estados y transiciones en un conjunto finito de clases de equivalencia, utilizamos las enabledness-preserving abstractions (EPAs). Estas abstracciones permiten agrupar estados según las operaciones que se pueden realizar a partir de ellos y pueden ser caracterizados con un invariante de estado. Por ejemplo, en el caso de la pila mencionada anteriormente, los estados donde la pila tiene uno o dos elementos son equivalentes porque ambos permiten las mismas operaciones: apilar y des-apilar. En cambio, el estado vació solo permite la operación de apilar elementos. Las EPAs se describen como un labeled transition system (LTS) finito y potencialmente no determinístico.

El invariante de un estado abstracto de una clase $C = \langle M, F, R, inv, init \rangle$ se da por un conjunto de métodos $ms \subseteq M$ y se expresa como el predicado $inv_{ms} : C \to \{verdadero, falso\}$, tal que:

$inv_{ms}(c) \leftrightarrow inv(c) \wedge \bigwedge_{m \in ms} Rm(c) \wedge \bigwedge_{m \notin ms} \neg Rm(c)$

De acuerdo a esta definición, un estado abstracto ms es válido si y solo si existe una instancia $c \in C$ en la que se encuentran habilitados los métodos del conjunto ms y ningún otro método lo está, es decir, todos los otros métodos que no pertenecen a este conjunto se encuentran deshabilitados en esa misma instancia c.

Una `EPA` (Enabledness preserving abstraction) $\langle \Sigma, S, S_0, \delta \rangle$ se define a partir de una clase $C = \langle M, F, R, inv, init \rangle$ y consta de cuatro elementos: un conjunto de etiquetas de transición Σ que corresponde a los métodos de la clase C, un conjunto de estados S que es el conjunto de todos los posibles subconjuntos de M, un estado inicial S_0 que consiste en los subconjuntos de métodos válidos para una instancia c en la que se satisface la invariante de estado y la condición inicial, y una función de transición de estado δ que establece cómo se conectan los estados mediante las etiquetas de transición.
Las condiciones que debe cumplir una `EPA` son las siguientes:

- El conjunto de etiquetas de transición Σ es igual al conjunto de métodos M de la clase C.
- El conjunto de estados S es el conjunto de todos los posibles subconjuntos de M.
- El estado inicial S_0 es el conjunto de subconjuntos de métodos que son válidos para una instancia c en la que se satisface la invariante de estado y la condición inicial.

- Para cada estado ms y cada método $m \in \Sigma$, si el subconjunto de métodos ms no contiene al método m, entonces la función de transición de estado δ devuelve un conjunto vacío. Si ms contiene al método m, entonces δ devuelve un conjunto de estados ns que corresponden a los conjuntos de métodos que son válidos para una instancia c en la que se satisface la invariante de estado de ms y se satisface la post condición del método m en la instancia c.

En términos formales, se puede afirmar que cualquier ejecución válida de código que utiliza correctamente un objeto de una clase en términos de su protocolo, representa un camino en la `EPA` correspondiente a dicha clase.

Un enfoque para la generación de EPAs consiste en enumerar todos los posibles estados abstractos que la `EPA` podría tener, basándose en los métodos públicos que tiene la clase. Es decir, si una clase tiene n métodos públicos, entonces el número máximo de estados abstractos que podría tener la `EPA` correspondiente es 2^n.

Para generar una `EPA`, se examinan todos los posibles estados abstractos que podría tener la clase, se seleccionan aquellos que cumplen ciertas condiciones y se eliminan aquellos que no son alcanzables.

Cabe mencionar que existen técnicas de generación de EPAs más efectivas que no se abordarán en este trabajo.

2.2. Blockchain

Una blockchain es un registro distribuido que mapea cuentas a cantidades de una criptomoneda (como Bitcoin o Ether). La historia completa de las transacciones entre estas cuentas se almacena en la blockchain y está disponible públicamente. El estado actual de cada cuenta se puede calcular al recorrer la historia de todas las transacciones. La blockchain se implementa como una secuencia de bloques que se comparten a través de una red peer-to-peer.

Ethereum [7] va más allá de la infraestructura estándar de la blockchain al proporcionar un estado de programa distribuido. Los participantes pueden modificar el estado del programa ejecutando contratos inteligentes. Un contrato inteligente es un programa que se almacena en una dirección de blockchain de la misma manera que una cuenta estándar. Por lo tanto, los contratos inteligentes son cuentas que pueden recibir y transferir fondos. Al igual que las transacciones entre cuentas en la blockchain no se pueden revertir, un programa implementado en un contrato inteligente no se puede modificar.

Las funciones de los contratos inteligentes se llaman desde cuentas (direcciones en la blockchain) que tienen fondos. Para ejecutarse, el contrato inteligente consume una cantidad de gas (es decir, una cantidad de la criptomoneda) que el caller del contrato inteligente debe gastar. La cantidad de gas que cuesta la ejecución del contrato inteligente dependerá de las acciones específicas que realice el contrato inteligente y la carga de la red.

Ethereum proporciona una máquina virtual (Ethereum Virtual Machine o EVM) para ejecutar las transacciones entre cuentas y actualizar el estado interno de la blockchain. Además, se proporciona un lenguaje de programación llamado `Solidity` para permitir a los programadores escribir cómo se debe actualizar el estado de la blockchain.

```
1
2 pragma \verb|Solidity| ^0.5.0;
3
4 contract Crowdfunding {
```

```
    address payable owner;
    uint max_block;
    uint goal;
    mapping(address => uint) backers;
    bool funded = false;

    constructor(address payable _owner, uint _max_block, uint _goal) public {
        owner = _owner;
        max_block = _max_block;
        goal = _goal;
    }

    function Donate() public payable {
        require(max_block > block.number);
        require(backers[msg.sender] == 0);
        backers[msg.sender] = msg.value;
    }

    function GetFunds() public {
       require(max_block < block.number);
        require(msg.sender == owner);
        require(goal <= address(this).balance);
        funded = true;
        owner.transfer(address(this).balance);
    }

    function Claim() public {
        require(max_block < block.number);
        require(backers[msg.sender] > 0 && !funded);
        require(goal > address(this).balance);
        uint val = backers[msg.sender];
        backers[msg.sender] = 0;
        msg.sender.transfer(val);
    }

}
```

Listing 1: Contrato Crowdfunding

Ejemplo de Crowdfunding. Es un contrato simple que admite crowdfunding tomado del Benchmark de Azure [5]. Este contrato tiene un constructor que establece un objetivo de recaudación, el propietario del contrato y una fecha límite para las donaciones. Una vez implementado en la blockchain, terceros pueden donar fondos. Una vez que se haya pasado la fecha límite, si se ha alcanzado el objetivo de recaudación, el propietario puede obtener los fondos donados. De lo contrario, cuando se haya pasado la fecha límite, los donantes pueden reclamar sus donaciones.

El contrato inteligente se compone de un estado local definido en las primeras líneas del listado, y funciones que son llamadas por cualquier participante en la cadena de bloques. La variable de estado *owner* es la cuenta que recibirá los fondos que se recojan. La variable *goal* contiene la cantidad total que el propietario espera recolectar. La variable *max_block* codifica la fecha límite de donación. Se puede notar que en lugar de usar una variable de tiempo, se utiliza un número de bloque para referirse al instante en que se extiende la cadena de bloques para incluir el bloque *max_block*. Este es un mecanismo común para establecer la expiración del contrato utilizado para evitar ataques en los que el tiempo local se altera maliciosamente [8]. La variable *backers* es un mapeo que almacena cuánto se comprometen diferentes contribuyentes en la recaudación de crowdfunding. Inicialmente todas las direcciones se asignan a cero. Finalmente, *funded* es un booleano que indica si la iniciativa alcanzó el objetivo de financiación y los fondos fueron retirados por el propietario.

El contrato tiene cuatro funciones públicas que están disponibles para cualquier cuenta que lo invoque. El constructor, que solo se ejecuta cuando el smart contract se deploya en la cadena de bloques y simplemente almacena los argumentos en variables de estado. Además, el constructor configura el modificador `payable` para el parámetro *owner*. Una dirección `payable` es un tipo de dirección que puede recibir Ether. Esto contrasta con una dirección regular, que no puede recibir esta moneda.

La función *Donate*() no tiene argumentos explícitos, pero utiliza argumentos implícitos como *msg* y *block* que contienen información como la dirección del caller de la función

(msg.sender), la cantidad de Ether comprometida (msg.value) y el número de bloque actual (block.number). La función incluye dos cláusulas requires. Si estas cláusulas no se cumplen cuando se invoca la función, falla y se consume una pequeña cantidad de gas. La primera cláusula requiere que el número de bloque actual de la cadena de bloques (block.number) sea menor que el valor almacenado en max_block, es decir, que la fecha límite de donación no haya pasado. La segunda cláusula requiere que el caller no haya donado antes.

Es importante destacar el modificador payable en la declaración de la función $Donate()$. El smart contract tiene su propia dirección y el modificador payable permite que esta función reciba Ether del llamador. Después de la terminación exitosa de $Donate()$, la cantidad comprometida por el caller(msg.value) se transferirá desde el mismo msg.sender a la dirección del contracto address(this). Si msg.value es mayor que el saldo del caller, la transacción fallará y se restaurará el estado del smart contract.

El contracto tiene otras dos funciones que utilizan las expresiones balance y transfer(). La primera lee el saldo actual del smart contract, mientras que la segunda permite la transferencia entre cuentas. $GetFunds()$ requiere que el período de financiamiento no haya terminado y que se haya alcanzado la meta de financiamiento para permitir al propietario transferir las donaciones totales. $Claim()$ requiere que el objetivo de financiamiento colectivo haya terminado sin éxito, es decir, con menos donaciones que la meta, para permitir que los backers reclamen su donación.

2.3. VeriSol

VeriSol es una herramienta de verificación formal para contratos inteligentes escritos en Solidity. Es desarrollado por Microsoft Research [4] y está disponible como parte de las Herramientas de Blockchain de Azure. Este es un verificador de propósito general y no está vinculado a un Workbench. La herramienta codifica la semántica de los programas Solidity en un lenguaje de verificación intermedio de bajo nivel llamado Boogie [9] aprovechando su pipeline de verificación, tanto para la verificación como para la generación de contraejemplos. En particular, VeriSol aprovecha la herramienta existente de comprobación de model checking Corral [10] para Boogie para encontrar violaciones de afirmaciones, y aprovecha los generadores prácticos de condiciones de verificación para Boogie para automatizar pruebas de corrección. Además, VeriSol utiliza la abstracción de predicados monomiales para inferir automáticamente los llamados invariantes de contrato, los cuales hemos encontrado que son cruciales para la verificación automática de la conformidad semántica.

Más en detalle, los pasos que sigue VeriSol para analizar un contrato son:

- Traducción a Boogie: El primer paso consiste en la traducción del código fuente del contrato Solidity a una representación intermedia en el lenguaje Boogie, que es un lenguaje de programación intermedio utilizado en la verificación formal. Esta traducción se realiza mediante el módulo de traducción de Boogie en VeriSol.

- Generación de invariantes: A continuación, VeriSol utiliza el módulo de generación de invariantes para inferir invariantes del contrato, así como invariantes de bucle y resúmenes de procedimientos. Los invariantes son propiedades que deben mantenerse verdaderas en todo momento durante la ejecución del contrato. Estos invariantes son

utilizados posteriormente por el verificador para demostrar la corrección del contrato o encontrar errores.

- Model checking acotado: El siguiente paso es el model checking acotado. En este paso, `VeriSol` explora todas las posibles ejecuciones del contrato dentro de un límite acotado de transacciones (definido por el usuario) y verifica si se cumplen los invariantes inferidos en el paso anterior. Si se encuentra una ejecución que viola uno o más invariantes, el verificador produce un contraejemplo que muestra cómo se viola la propiedad.

Una vez completado el model checking acotado, `VeriSol` produce uno de los siguientes tres resultados para el contrato:

- Completamente verificado: Si `VeriSol` es capaz de demostrar que todas las afirmaciones del contrato son verdaderas en todas las posibles ejecuciones del contrato dentro del límite acotado de transacciones, entonces el contrato se considera completamente verificado.
- Refutado: Si `VeriSol` encuentra una ejecución del contrato que viola una o más afirmaciones, entonces el contrato se considera refutado. En este caso, `VeriSol` también produce un contraejemplo que muestra cómo se viola la propiedad.
- Parcialmente verificado: Si `VeriSol` no puede verificar ni refutar completamente la corrección del contrato dentro del límite acotado de transacciones, entonces se considera que el contrato está parcialmente verificado. En este caso, `VeriSol` realiza una verificación limitada para establecer que el contrato es seguro hasta un número de transacciones definido por el usuario.

En caso de que `VeriSol` detecte una violación en el contrato durante su ejecución, detendrá el análisis y no continuará evaluando posibles refutaciones adicionales.

2.4. Trabajos relacionados

En esta sección, analizamos investigaciones previas sobre la creación de abstracciones para smart contracts. Para apoyar nuestra propia investigación nos basamos en un estudio anterior [1] que demostró la utilidad de estas abstracciones para auditores de contratos, quienes deben realizar tareas como comprensión, verificación y validación. Sin embargo, la creación de estas abstracciones se lleva a cabo de forma manual, lo que implica la traducción del código a un lenguaje de modelado que luego puede procesarse con una herramienta llamada `Alloy`.

Para generar una abstracción con `Alloy`, primero se debe definir los predicados que describen el comportamiento del contrato inteligente. Estos predicados se pueden derivar del código del contrato inteligente. Luego se debe definir los invariantes que deben cumplirse para todos los estados del contrato inteligente. Finalmente, se usan las capacidades de verificación de modelos de `Alloy` para analizar el modelo y verificar que los invariantes siempre se cumplan.

En ingeniería de software, un invariante se utiliza para especificar una propiedad que se mantiene verdadera en un programa en un punto determinado de su ejecución. Es una restricción que se aplica en una clase, método o estructura de datos que se supone que no

cambia durante la ejecución del programa. Los invariantes son útiles para asegurarse de que ciertas condiciones se mantengan en todo momento, incluso en situaciones imprevistas o excepcionales. Se pueden utilizar para garantizar la corrección de un programa y prevenir errores que puedan resultar en comportamientos no deseados o en fallas.

En este trabajo se va a usar la herramienta `VeriSol` a diferencia del lenguaje `Alloy`.

En `Alloy` el usuario debe traducir manualmente el código del contrato a un modelo `Alloy`, lo que requiere un conocimiento profundo del lenguaje de modelado y del contrato en sí. Luego, se puede utilizar el analizador de `Alloy` para verificar propiedades específicas del contrato.

`VeriSol` utiliza técnicas de inferencia para generar automáticamente abstracciones del contrato. Esto significa que el usuario no necesita tener un conocimiento profundo de cómo construir un modelo de contrato en un lenguaje de modelado específico, ni cuáles son sus invariantes. `VeriSol` es capaz de inferir automáticamente el modelo a partir del código fuente del contrato, lo que reduce significativamente el tiempo y el esfuerzo necesarios para generar las abstracciones.

En este trabajo vamos a usar los mismos Benchmarks en las validaciones que en [1] para poder comparar los resultados obtenidos.

3. Desarrollo

El objetivo del estudio es desarrollar una herramienta para construir abstracciones automáticas de smart contracts utilizando un analizador estático de programas Solidity, llamado VeriSol. Se quiere generar dos tipos de abstracciones a partir de esta herramienta: las abstracciones EPA y las abstracciones de estados previamente definidos. Se va a analizar el proceso de desarrollo de esta herramienta en detalle a continuación.

3.1. EPA

Para iniciar, se utilizará la definición de EPA que fue vista en la sección anterior. Se partirá de un contrato Ethereum para construir una abstracción empleando la herramienta `VeriSol`. Concretamente, los estados posibles del grafo serán todas las combinaciones posibles de las funciones disponibles en dicho contrato, lo que equivale a 2^{CF}, donde CF representa la cantidad de funciones del contrato C.

La precondición de un estado va a ser igual a la:

$$pre_s = \bigwedge_{f \in s} pre_f \wedge \bigwedge_{f \notin s} \neg pre_f$$

Dos estados $S1$ y $S2$ estarán conectados mediante una función f con parámetros $param$ si se cumplen las siguientes condiciones:

- La precondición de f debe ser parte de la precondición de $S1$: $pre(f) \in pre(S1)$.
- La precondición de $param$ debe ser verdadera: $pre(param)$ es verdadera.
- Luego de la ejecución de f se debe llegar al estado $S2$, es decir, la precondición de $S2$ debe ser verdadera: $pre(S2)$ es verdadera.

En otras palabras:

- Voy a suponer que estoy en un estado X
- Voy a querer llamar a una función con sus parámetros correctos
- Voy a querer ver si llego a un estado Y

Esta idea se puede utilizar en `Solidity` de la siguiente manera:

```
1
2 function doesTransictionExists(params) public {
3     require(S1_precondtion);
4     require(func_params_precondition);
5     func(params);
6     assert(!S2_precondition):
7 }
```

Listing 2: Función en Solidity para validar transición entre dos estados

Cuando se desea ejecutar una función en un contrato utilizando la herramienta `VeriSol`, se generará una instancia del contrato que cumpla con ambas precondiciones, es decir, la del estado actual y la de la función que se desea llamar. Si `VeriSol` no logra construir dicha instancia, la ejecución de la función se detendrá. En cambio, si se logra construir el contrato, se ejecuta la función y se verifica que se cumple la precondición negada del

estado al que se desea llegar. Si la ejecución de `VeriSol` devuelve un error $assertfalse$ en el contrato, esto indica que se alcanzó el estado deseado y, por lo tanto, la transición entre los estados existe. Aquí se muestra un ejemplo de la respuesta obtenida:

```
*** Found a counterexample (see corral.txt)
-----Transaction Sequence for Defect ------
DigitalLocker::Constructor (this = address!0, msg.sender = address!3,
msg.value = 39, bankAgent = address!4)
DigitalLocker::BeginReviewProcess (this = address!0, msg.sender = address!2,
msg.value = 43)
DigitalLocker::UploadDocuments (this = address!0, msg.sender = address!2,
msg.value = 44, lockerIdentifier = 6724, imageCode = 533)
DigitalLocker::vc0x5x3 (this = address!0, msg.sender = address!3,
msg.value = 4461, rejectionReason = 45, lockerIdentifier = 46,
image = 47, thirdPartyRequestor = address!5, expirationDate = 505)
OutputTemp3.sol(135,1): : ASSERTION FAILS!
```

Es relevante resaltar que para garantizar la precisión del análisis, el contrato de estudio no debe utilizar la palabra clave *assert* para verificar condiciones que puedan no cumplirse, sino que debe emplear la función *require*. Aunque ambas funciones (require y assert) detienen la ejecución en caso de fallo, el lenguaje de programación `Solidity` recomienda el uso de *assert* solamente para detectar violaciones de invariantes internos que no se espera que fallen durante la ejecución del contrato. En consecuencia, el uso de *require* permite al sistema detectar errores en las entradas o condiciones externas al contrato, en lugar de violaciones internas del contrato.

Se procederá a verificar si una transición específica existe entre dos estados considerando exclusivamente a las funciones presentes en la precondición del estado actual $S1$. Las transiciones a través de funciones que no están en la precondición de $S1$ no se explorarán, porque por definición no se va a cumplir nunca su precondición. Esta elección se realiza con el objetivo de reducir la cantidad de posibles combinaciones de estados y transiciones que deben ser exploradas durante el proceso de construcción de la abstracción. De esta forma, se busca mejorar la eficiencia y reducir el tiempo necesario para realizar el análisis.

Una vez construído este conjunto de transiciones y estados, se puede afirmar que solo incluyen estados alcanzables, ya que se ha podido cumplir su precondición a través de una construcción del contrato y aplicándole cierto número de acciones. Es decir, se descartan aquellos estados que no pueden ser alcanzados desde un estado inicial *init*.

La cantidad de posibilidades a explorar en el peor caso para generar la abstracción es $2^{CF} \times CF \times 2^{CF}$ donde CF es el número de funciones en el contrato.

Con estos resultados, se pueden graficar y casi tenemos una EPA. Sin embargo, es necesario agregar un estado init que se conecte con transiciones al grafo. Para esto, se debe determinar a qué estados se debe conectar init a través de la función constructora. Estas transiciones estarán conectadas a estados donde la precondición se puede cumplir sin necesidad de realizar alguna acción del contrato.

Esta idea se puede expresar en `Solidity` de la siguiente manera:

```
function doesTransictionExists(params) public {
    assert(!state_end_precondition):
}
```

El problema que surge al ejecutar `VeriSol` con esta modificación en el contrato, es que se va a informar para todos los estados válidos que existe una instancia donde se puede llegar a cumplir la precondición final. Sin embargo, lo que se desea es obtener información únicamente sobre los estados alcanzables partiendo desde el estado inicial del contrato y a través de las funciones constructoras.

Para resolver este problema, se puede ajustar la configuración de `VeriSol` txBound, que determina la cantidad de acciones que se ejecutan para construir una instancia de un contrato. Por defecto, este valor es 4, pero en este caso se desea que sea 1.

De esta manera, se puede agregar a la abstracción previa un estado inicial y la función constructora que se conectan a los primeros estados válidos a los que se puede llegar cumpliendo la precondición mediante ninguna acción. De esta forma, se obtiene una EPA que representa de forma precisa y eficiente los estados y transiciones alcanzables del contrato.

El nombre de los estados generados por la herramienta de análisis, es la lista de funciones habilitadas en cada estado. En caso de que no existan funciones habilitadas en un estado, este se va a llamar Vacío para facilitar su identificación y comprensión en el análisis de la abstracción generada.

3.2. States

En este método, se busca generar una abstracción del contrato Ethereum. A diferencia del método anterior, se permite seleccionar específicamente los estados que se desean analizar, junto con sus correspondientes precondiciones. Además, se pueden introducir predicados arbitrarios, lo cual tiene ventajas y desventajas. Por un lado, permite generar diferentes abstracciones con distintos niveles de detalle, en función del enfoque deseado por el auditor. Sin embargo, también implica la necesidad de introducir información de forma manual.

Para lograr esto, se seguirá utilizando la herramienta `VeriSol` de manera similar al método anterior. La principal diferencia radica en que en este caso no es necesario generar todos los posibles estados a través de las funciones con sus respectivas precondiciones.

La cantidad de posibilidades que se deben explorar para generar esta abstracción es igual a:

$CE * CF * CE$

donde CE es la cantidad de estados dados y CF es igual a la cantidad de funciones públicas. Esto se debe a que se debe considerar cada posible combinación de estados y funciones para determinar si existe una transición válida entre ellos.

Una vez definidos los estados y precondiciones a analizar, se puede utilizar `VeriSol` para explorar las posibles transiciones entre estos estados. Esto permitirá generar un grafo de transiciones que conecta los estados seleccionados. Cabe destacar que estos estados solo incluyen aquellos que se hayan cumplido a través de construcciones de `VeriSol`, por lo que son estados alcanzables.

Finalmente, se debe agregar un estado inicial a la abstracción y conectarlo con las transiciones que cumplen su precondición sin necesidad de ejecutar ninguna función del contrato. De esta manera, se obtiene una abstracción que representa el contrato Ethereum seleccionado y los estados y transiciones alcanzables partiendo desde el estado inicial a través de las funciones constructoras.

3.3. Creación de la tool

Hasta ahora se ha realizado un proceso manual para crear abstracciones de contratos. Esto ha implicado la tarea de escribir los inputs necesarios para validar cada estado y cada transición del contrato en Solidity, para luego utilizar la herramienta `VeriSol`. Sin embargo, el propósito de este trabajo es automatizar este proceso con el fin de generar abstracciones de nuevos contratos.

Se exploró la posibilidad de automatizar la creación de abstracciones utilizando el lenguaje `Solidity`. Sin embargo se llegó rápidamente a la conclusión de que no es un lenguaje práctico para lograr este objetivo. La complejidad y limitaciones del lenguaje hacen que sea difícil escribir código para generar las abstracciones de manera automática, además esto requería modificar los contratos. En lugar de esto, se buscó una alternativa más viable para generar abstracciones de contratos de manera eficiente y automatizada.

Por este motivo se optó por desarrollar un programa en Python que fuera capaz de inyectar inputs en el código `Solidity` del contrato, ejecutar el análisis de `VeriSol` con estos inputs, interpretar los resultados obtenidos y, finalmente, generar el grafo de la abstracción. De esta manera, se busca optimizar el proceso de generación de abstracciones, al automatizar gran parte del trabajo manual previo que se requería para ello.

Para que el programa en Python pueda comprender la estructura del contrato, se requerirá de un archivo de configuración que contendrá información clave. Esta configuración incluirá detalles de las funciones públicas del contrato y las precondiciones asociadas a cada función.

En el futuro, es posible que la información necesaria para la generación de la abstracción se pueda obtener directamente del contrato mediante técnicas de análisis estático. Sin embargo, en la actualidad, la creación de un archivo de configuración detallado es necesaria para permitir que el programa en Python comprenda la estructura del contrato y genere la abstracción de manera eficiente.

En los archivos de configuración se incluyen:

- Nombre del contrato y del archivo
- txBound de `VeriSol`
- Las funciones que se van a analizar del contrato, con sus precondiciones y de sus parámetros

Para las abstracciones States además se necesita:

- Los nombres de los estados que se desean observar, con un identificador
- Las precondiciones de estos estados.

El archivo de configuración correspondiente al Listing 1 de la sección de preliminares se puede observar en el Listing 3.

En este archivo de configuración, se encuentran definidas diversas variables y parámetros que son relevantes para el análisis del contrato. En primer lugar, se especifica el nombre del archivo de contrato y su respectivo nombre.

A continuación, se detallan todas las funciones públicas del contrato, junto con sus precondiciones y los parámetros correspondientes. Estas funciones deben ser escritas en el lenguaje Solidity, ya que los strings serán utilizados en los contratos.

La variable $functionVariables$ declara los parámetros de las funciones que no están ligados y que $VeriSol$ deberá explorar.

Las variables *statesModeState* y *statesNamesModeState* describen los estados del modo State en las abstracciones. La primera variable representa el *id* del estado, el cual es generado automáticamente en el modo EPA, y la segunda variable indica el nombre del estado en el grafo. A futuro, se puede implementar la generación automática del *id* en el modo de estados.

A continuación, la variable *statePreconditionsModeState* especifica las precondiciones de los estados previamente definidos. Estas precondiciones deben seguir el mismo orden establecido.

Por último, la variable *txBound* define el límite máximo que Verisol usará de transacciones que se exploran durante la búsqueda de contraejemplos.

```
1  fileName = "Crowdfunding.sol"
2  contractName = "Crowdfunding"
3  functions = [
4  "Donate();",
5  "GetFunds();",
6  "Claim();",
7  ]
8  statePreconditions = [
9  "(max_block > blockNumber)",
10 "(max_block < blockNumber && goal <= balance)",
11 "(blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)",
12 ]
13 functionPreconditions = [
14 "backers[msg.sender] == 0",
15 "msg.sender == owner",
16 "backers[msg.sender] != 0",
17 ]
18 functionVariables = ""
19 tool_output = "Found a counterexample"
20
21 statesModeState = [[1,0,0,0], [0,2,0,0], [0,0,3,0], [0,0,0,4] ]
22 statesNamesModeState = ["Vacio", "Donate", "Funds", "Claim"]
23 statePreconditionsModeState = [
24 "(!(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance) && !(blockNumber >
        max_block && !funded && goal > balance && backersArray.length != 0))",
25 "(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance) && !(blockNumber >
      max_block && !funded && goal > balance && backersArray.length != 0)",
26 "!(max_block > blockNumber) && (max_block < blockNumber && goal <= balance) && !(blockNumber >
      max_block && !funded && goal > balance && backersArray.length != 0)",
27 "!(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance) && (blockNumber >
      max_block && !funded && goal > balance && backersArray.length != 0)",
28 ]
29
30 txBound = 4
```

Listing 3: Archivo de config Crowdfunding

En la sección 4 de Validación se proporcionará una explicación más detallada de cómo son los archivos de configuración para este programa.

En la estructura raíz del proyecto, se encuentra una carpeta denominada *Contracts* que alberga todos los contratos a los que se desea generar abstracciones. Además existe la carpeta $Configs$ que contiene las configuraciones de los contratos con las observaciones a analizar. En la carpeta *Graph* , se generarán los gráficos correspondientes al análisis. El programa de este estudio se encuentra en el directoria raíz con el nombre $Tesis.py$

Para utilizar este programa, es necesario ejecutarlo siguiendo el siguiente formato:

```
python3 Tesis.py [Config] [Print Mode] [Mode] [Reduce Mode]
```

Aquí, $Config$ es el archivo de configuración que se desea ejecutar. Este archivo contendrá la información del contrato y las funciones a observar. Este contrato se buscará dentro de la carpeta *Contracts* en el directorio raíz.

El parámetro $PrintMode$ indica cuánta información se desea imprimir en pantalla durante la ejecución del programa. Puede ser en modo verbose (-v), donde se imprime

la máxima cantidad de información posible, incluyendo los comandos de `VeriSol` que se ejecutan y sus resultados. O bien, puede ser en modo simple (-t), donde se imprime la menor cantidad posible de información en pantalla. Este parámetro es obligatorio.

Luego se encuentra la configuración del modo, donde se puede indicar si se desea una abstracción de estados (-s) o una abstracción de EPA (-e). Este parámetro es obligatorio.

Por último, se encuentra la configuración de $ReduceMode$, que se utilizará en la sección de experimentación, la cual se detallará en la subsección de *Optimizaciones* 3.5. Por defecto, todas las optimizaciones son usadas. Se agrega la opción de apagar estas optimizaciones para que sea posible medir su eficacia en la sección de Experimentación. Las diferentes opciones para este modo son:

- $ReduceTrue()$: Elimina la optimización de las precondiciones verdaderas.
- $ReduceEqual()$: Elimina la optimización de las precondiciones iguales.
- $ReduceStates()$: Elimina la optimización de encontrar estados válidos.
- $ReduceAll()$: Incluye todas las configuraciones anteriores.

La ejecución de este programa se puede desglosar en distintos pasos importantes, que se describen detalladamente a continuación:

- Obtención de los estados posibles a explorar.
- Reducción de los estados posibles mediante técnicas de optimización.
- Identificación de las transiciones entre estados válidos.
- Identificación de las transiciones iniciales.
- Representación gráfica de los resultados obtenidos.

Obtención de los estados posibles a explorar

El primer paso es obtener los posibles estados de la abstracción. En el modo de estados, esto es sencillo, ya que estos están definidos en la configuración.

En cambio, si se utiliza el modo EPA, se genera un array con todas las combinaciones posibles de funciones que se pueden ejecutar en un estado y se almacena en otro arreglo la precondición correspondiente a cada combinación. La precondición se construye a partir de las precondiciones de las funciones especificadas en la configuración, como se explicó previamente en la subsección 2.1.

Por ejemplo, se tienen las funciones a, b y c con las precondiciones A, B y C respectivamente. El arreglo de combinaciones que se va a generar es: [[], [1], [1, 2], [1, 3], [2], [2, 3], [3], [1, 2, 3]] donde cada número corresponde a una función. El arreglo con las precondiciones de cada función va a ser: [[$true$], [A], [$A\&\&B$], [$A\&\&C$], [B], [$B\&\&C$], [C], [$A\&\&B\&\&C$]]

Reducción de los estados posibles mediante técnicas de optimización

Una vez que se tienen todas las combinaciones de estados posibles, se realiza una reducción utilizando `VeriSol` para obtener solo los estados válidos. Un estado es considerado válido si es alcanzable, es decir, se puede llegar a él desde el estado inicial del contrato mediante la ejecución de un conjunto de acciones permitidas. Se explica más adelante como se puede utilizar `VeriSol` con este fin en la subsección 3.5.

Para esta reducción de estados válidos utilizando `VeriSol`, se utilizarán threads para hacer el proceso más eficiente como se explica en la subsección 3.5.

El número de veces que se ejecutará `VeriSol` en el contrato será igual al número de estados a verificar. Durante el proceso de ejecución, se marcarán los estados que se consideren válidos para su posterior utilización. Una vez que se hayan ejecutado todas las verificaciones, se tendrá una lista de estados válidos y se descartarán los que no cumplan con los criterios previamente establecidos. Este proceso de reducción de estados permitirá simplificar la complejidad del contrato original y, por lo tanto, facilitar su análisis. Además, al utilizar threads, se podrá realizar la verificación de múltiples estados de manera más eficiente.

Identificación de las transiciones entre estados válidos

Una vez obtenido el nuevo conjunto de estados válidos S con sus precondiciones asociadas, el siguiente paso es obtener las transiciones que conectan estos estados. Este proceso también se realiza utilizando threads como se explica en la subsección 3.5. Para cada thread, se crea una copia del contrato en una nueva carpeta, ya que `VeriSol` utiliza archivos temporales en el directorio para su ejecución. Ejecutar varios threads en el mismo directorio puede producir resultados esporádicos y no confiables.

En cada copia del contrato, se suman las funciones que verifican si existe una transición entre dos estados, tal como se explicó anteriormente en el Listing 2. Esto se escribe como un string en la copia del contrato de `Solidity` usando su semántica correspondiente. La cantidad de veces que se ejecuta `VeriSol` en el contrato es igual a la cantidad de transiciones a probar.

Recordemos el Listing 2:

```
1
2 function doesTransictionExists(params) public {
3     require(S1_precondtion);
4     require(func_params_precondition);
5     func(params);
6     assert(!S2_precondition):
7 }
```

El conjunto de estados válidos S se divide entre los threads, generando subconjuntos T_i.

Luego $\forall S1 \in T_i, \forall func \in S1, \forall S2 \in S$ se va a probar la transición. Las transiciones válidas se guardan para luego graficarlas.

Identificación de las transiciones iniciales

En este paso, se procederá a obtener los estados que se encuentran conectados con el estado *init*. Este proceso se realizará utilizando nuevamente el conjunto S obtenido en pasos anteriores y dividiéndolo en la cantidad de threads a utilizar. Para cada thread, se creará una copia del contrato en una nueva carpeta, en la cual se sumarán las funciones

necesarias para verificar si se puede cumplir un estado válido. Estas funciones se escribirán en la copia del contrato de `Solidity` correspondiente utilizando su semántica adecuada.

Cada función agregada en el contrato necesitará ser ejecutada en `VeriSol` utilizando el parámetro txBound en 1 para cumplir con nuestro requerimiento. Las transiciones que se encuentren válidas se guardarán para luego ser utilizadas en la visualización del grafo.

Representación gráfica de los resultados obtenidos

Para finalizar, se utilizará la librería graphviz para graficar las transiciones obtenidas en los pasos anteriores. Esto permitirá visualizar el grafo que representa la abstracción del sistema en cuestión, lo que será de gran ayuda para entender su comportamiento y analizar posibles problemas en su funcionamiento.

3.4. Limitaciones

Al utilizar la herramienta `VeriSol` para analizar contratos en `Solidity`, se identificaron varias limitaciones que se describen a continuación.

En primer lugar, se ha encontrado que `VeriSol` no es compatible con las versiones más recientes de `Solidity`, ya que solo admite hasta la versión 0.6. Esta limitación puede ser un problema para aquellos contratos que requieren características avanzadas que solo están disponibles en las versiones más nuevas de `Solidity`.

Otra limitación encontrada es que los contratos que usan herencias y utilizan la palabra clave *super* no pueden ser compilados con `VeriSol`. Esto se debe a que la herramienta no tiene soporte para la palabra clave *super*, lo que hace que la ejecución se detenga inesperadamente antes de devolver un resultado. Para superar esta limitación, se ha optado por eliminar la herencia y pasar las variables y funciones necesarias al contrato padre.

Además, se ha identificado que `VeriSol` no infiere automáticamente las direcciones como tipo *address*, lo que puede generar errores al analizar contratos donde se utilizan. Para solucionar este problema, se ha implementado una conversión explícita de las direcciones a *address*, aunque sería posible crear un script que realice esta tarea de manera automatizada en casos similares. Por ejemplo, esta declaración:

```
address A = 0x0;
```

Se convierte en:

```
address A = address(0x0);
```

En el benchmark de Smartpulse, se ha identificado un problema relacionado con la imposibilidad de recorrer los mappings en Solidity para determinar si están vacíos o conocer la cantidad de elementos que contienen. Esta limitación ha llevado a la pérdida de transiciones en las abstracciones generadas por VeriSol, ya que no se puede contabilizar la cantidad de elementos guardados en un mapping para determinar si una función puede ejecutarse o no.

Para ilustrar esta problemática, consideremos un contrato en el cual distintas personas pueden depositar y extraer dinero. Sin la capacidad de contar los elementos guardados en el mapping, no sería posible saber si una persona puede realizar una extracción hasta que haya realizado un depósito previo. Esta limitación afecta la precisión de las precondiciones utilizadas en la interpretación de VeriSol.

Finalmente, en el benchmark de Smartpulse también hay contratos que utilizan *block.number* y *now* en las precondiciones. Estos valores cambian a lo largo de la ejecución del contrato, ya que constantemente se agregan bloques a la blockchain, pero `VeriSol` no tiene en cuenta este cambio. Para solucionar este problema, se han creado variables paralelas que se inicializan en el constructor y se ha implementado una función tau que aumenta en uno estas variables. Esta sirve para modelar la adición de nuevos bloques en la blockchain.

Se puede modelar a tau de la siguiente manera:

```
1 function tau() public {
2   blockNumber = blockNumber + 1;
3   time = time + 1;
4 }
```

Un cambio a futuro posible es modelar tau como se observa a continuación:

```
1 function tau(int n) public {
2   blockNumber = blockNumber + n;
3   time = time + 1;
4 }
```

Esto va a permitir encontrar más transiciones ya que se asemeja mayormente a la realidad de una blockchain.

Luego, dependiendo de lo que se quiera analizar, se puede llamar a tau al final de cada función pública o como una función adicional a analizar en la abstracción de `VeriSol`

Otra limitación que se encontró es la configuración de *txBound* de $Verisol$. El parámetro es una opción que se utiliza para limitar el número de transacciones que se exploran durante la búsqueda de contraejemplos.

El valor predeterminado es 4. Si se desea explorar contraejemplos con más de 4 transacciones, se puede especificar un valor mayor. Es importante tener en cuenta que aumentar el valor puede hacer que el proceso de verificación sea más lento. Por lo tanto se debe aumentar con precaución.

3.5. Optimización

En esta sección se describen las optimizaciones implementadas en el programa para mejorar su rendimiento. A continuación, se describe cada una de ellas en detalle y luego en la sección de experimentación vamos a ver si son útiles en el rendimiento del programa.

Threads

Una de estas optimizaciones es la utilización de múltiples threads, lo que permite una ejecución en paralelo del procesamiento y una reducción en el tiempo total de ejecución. Para ello, se divide la cantidad de llamadas a `VeriSol` por la cantidad de threads disponibles, de manera que cada thread procesa un subconjunto de llamadas en paralelo con los demás. Esto permite que el procesamiento se realice en paralelo y, por lo tanto, se reduzca el tiempo total de ejecución.

En la etapa de ejecución se creará una copia del contrato en una carpeta nueva para cada thread. Es importante que cada copia esté en una carpeta distinta debido a que `VeriSol` utiliza archivos temporarios en el directorio para su ejecución. Al ejecutar varios threads en el mismo directorio, pueden surgir resultados esporádicos y no deseables. En cada copia del contrato se sumarán las funciones que verifican los estados que le corresponden a cada thread. Para ello, se escribirá un string en la copia del archivo de `Solidity` usando la semántica correspondiente.

Reducción de la combinación de estados

La combinación de estados posibles en un programa para su análisis puede ser exponencialmente grande, de 2^n, donde n es la cantidad de funciones disponibles. Sin embargo, es posible reducir esta cantidad mediante la aplicación de dos restricciones:

- Reducción si una precondición es verdadera.
- Reducción si hay dos precondiciones idénticas.

La primera restricción consiste en reducir los estados donde la precondición de una función es true. Si una función tiene una precondición true, no tiene sentido explorar los estados donde esta función no se pueda ejecutar. Es decir, estos estados son inválidos ya que la precondición siempre se cumplirá.

La segunda restricción es la eliminación de estados redundantes donde dos funciones tienen la misma precondición pero tienen diferente comportamiento. Por ejemplo, si dos funciones a y b, tienen la misma precondición P, no es posible tener un estado válido donde se ejecuta solo la función a o solo la función b. a y b tienen que ambas estar habilitadas o deshabilitas de un estado. Por lo tanto, se pueden reducir los estados válidos mediante la eliminación de una de estas funciones y sus precondiciones.

Es importante destacar que estas restricciones reducen la cantidad de estados válidos sin afectar la precisión del análisis del programa. Al aplicar estas restricciones, se obtiene una reducción significativa en la cantidad de estados que deben ser analizados, lo que a su vez mejora el rendimiento del análisis del programa.

Eliminación de estados no válidos

Esta optimización se lleva a cabo para reducir el espacio de búsqueda y mejorar el rendimiento del programa. Se va a aprovechar del funcionamiento de la herramienta `VeriSol`. Antes de generar el conjunto de transiciones de una abstracción, se comprueba si los estados creados son válidos. Esta validación se realiza mediante la ejecución de una función específica en el contrato, como se muestra a continuación:

```
function doesTransitionExist(params) public {
require(state_precondition);
assert(false);
}
```

Si la función `doesTransitionExist` devuelve un valor de `assertFalse`, `VeriSol` será capaz de generar una instancia válida del contrato mediante la aplicación de acciones públicas permitidas por el contrato. De lo contrario, el estado generado será descartado por ser inalcanzable y no participará en ninguna abstracción. Esto se debe a que, como se ha explicado previamente, utilizar `VeriSol` para abstraer estados inalcanzables no es viable según su especificación. La eliminación de estos estados inválidos ayuda a reducir el espacio de búsqueda y, por ende, debería mejorar el rendimiento del programa.

Es importante señalar que las optimizaciones de reducción de estados solo se aplican a las abstracciones EPA, ya que en las abstracciones de estados, asumimos que la configuración ya proporciona los estados válidos y, por lo tanto, no es necesario reducirlos. Si esto no se puede asumir, se pueden hacer modificaciones para optimizar los estados dados.

Evitar escritura en disco excesiva

Existe una limitación en el análisis realizado por $VeriSol$ que interrumpe la evaluación de consultas adicionales si se encuentra un $assertFalse$ durante la ejecución del contrato. Esta restricción impide la posibilidad de agrupar todas las consultas en el contrato y ejecutar $VeriSol$ de manera simplificada. Como consecuencia, se debe escribir en disco una consulta a la vez en el contrato, lo que puede resultar en un aumento significativo en los tiempos de ejecución.

Para superar esta limitación y evitar la escritura repetitiva e innecesaria en disco, se ha adoptado una estrategia alternativa. En lugar de escribir cada consulta por separado, se opta por escribir toda la información requerida de una sola vez en disco. Posteriormente, se aprovecha la capacidad de configurar el parámetro $ignoreMethod$ de $VeriSol$, el cual permite excluir del análisis aquellas funciones del contrato que no están siendo utilizadas en ese momento.

4. Validación

El objetivo de esta sección es generar abstracciones precisas y confiables de los contratos inteligentes presentes en el Benchmark de Microsoft `Azure` y el Benchmark de `Smartpulse`. Ambos benchmarks se utilizaron en este estudio para evaluar la herramienta propuesta [1]. Estos contienen implementaciones de contratos inteligentes con una descripción detallada de su comportamiento previsto. Para cada uno de ellos, se generó un archivo de configuración que corresponde a sus especificaciones para poder generar la abstracción. Todos los resultados fueron comparados con los resultados obtenidos en [1].

Durante el análisis del Benchmark de `Azure`, se identificaron varios errores en las precondiciones de los métodos del contrato. Por lo tanto, se decidió crear una versión corregida de los mismos para observar el comportamiento de nuestra herramienta de abstracción en un contrato con precondiciones correctas.

En esta sección, se presentarán los resultados más significativos obtenidos de los resultados de nuestra herramienta de abstracción. El resto de los resultados se pueden observar en el repositorio fuente [11].

4.1. Hello Blockchain

Se puede observar a continuación, el caso más básico de nuestro set de ejemplos, el contrato *HelloBlockchain*. La aplicación expresa un flujo de trabajo entre una persona que envía una solicitud y otra persona que responde a la solicitud.

En la figura 1 se puede ver el requerimiento dado por `Azure`:

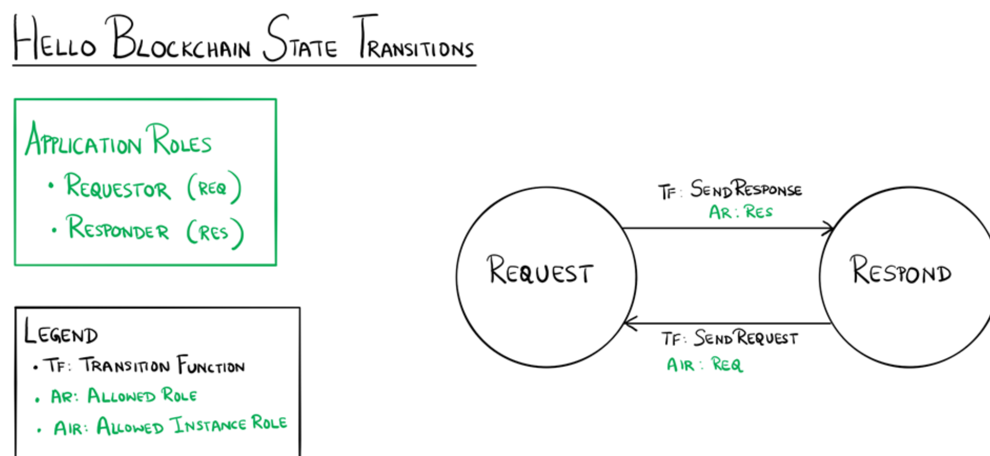

Fig. 1: Explicación de Azure de Hello Blockchain

Una instancia del flujo de trabajo de la aplicación Hello Blockchain comienza en el estado de Solicitud cuando un Solicitante hace una solicitud. La instancia pasa al estado de Respuesta cuando un Responder envía una respuesta. La instancia vuelve al estado de Solicitud cuando el Solicitante hace otra solicitud. Estas transiciones continúan siempre y cuando el Solicitante envíe una solicitud y un Responder envíe una respuesta.

Al analizar el código con el prototipo de la herramienta implementada en este trabajo, se obtuvo una abstracción diferente a la esperada. Observando la figura 2 se ve claramente

que se cumple nuestra hipótesis inicial.

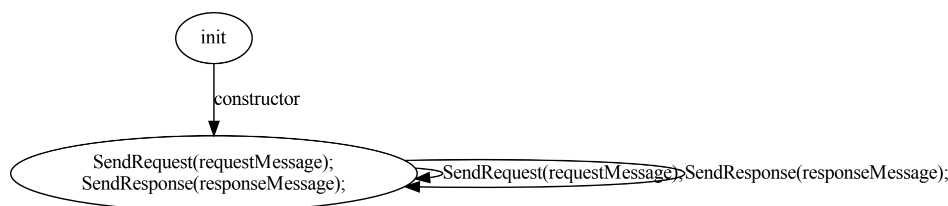

Fig. 2: Hello Blockchain Abstracción EPA

Se va a realizar una modificación al contrato actual con el fin de generar una abstracción que cumpla con la especificación deseada. Esto se puede observar en el Listing 4. Para ello, se utilizará la información del estado actual del contrato para definir las condiciones previas que se deben cumplir antes de ejecutar una acción determinada. De esta manera, se espera obtener una abstracción que contemple únicamente los estados y transiciones relevantes para el cumplimiento de la especificación, descartando aquellos que no sean necesarios.

```
1
2  contract HelloBlockchain {
3       //Set of States
4      enum StateType { Request, Respond}
5
6      //List of properties
7      StateType public  State;
8      address public  Requestor;
9      address public  Responder;
10     string public RequestMessage;
11     string public ResponseMessage;
12
13     // constructor function
14     constructor(string memory message) public{
15         Requestor = msg.sender;
16         RequestMessage = message;
17         State = StateType.Request;
18     }
19
20     // call this function to send a request
21     function SendRequest(string memory requestMessage) public {
22         // FIX: Add precondition
23         if (State != StateType.Respond) { revert();}
24         if (Requestor != msg.sender) {revert();}
25
26         RequestMessage = requestMessage;
27         State = StateType.Request;
28     }
29
30     function SendResponse(string memory responseMessage) public {
31         // FIX: Add precondition
32         if (State != StateType.Request) { revert();}
33         Responder = msg.sender;
34         ResponseMessage = responseMessage;
35         State = StateType.Respond;
36     }
37 }
```

Listing 4: Contrato Hello Blockchain

Se puede observar en la figura 3 que el resultado coincide con lo esperado.

Ya que se tiene una variable de estado en el contrato, se va a utilizar para generar una abstracción con estados previamente definidos. Se establecerá que los estados sean iguales a los valores posibles de esta variable de tipo *enum*.

Se procederá a generar el gráfico 4 correspondiente a la abstracción sin correcciones, donde se pueden distinguir claramente los diferentes estados definidos y se observa que cada

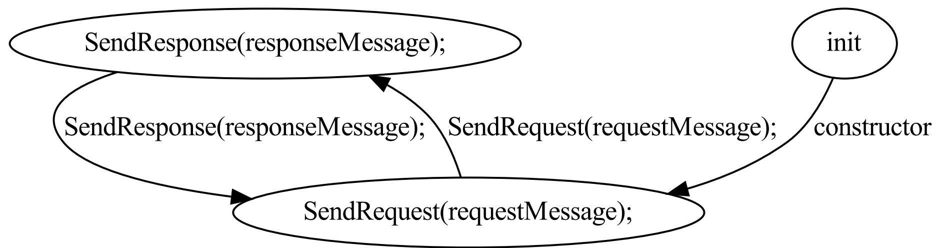

Fig. 3: Hello Blockchain con correcciones Abstracción EPA

uno está conectado consigo mismo. Esto es similar a lo que se observó en la abstracción EPA sin correcciones.

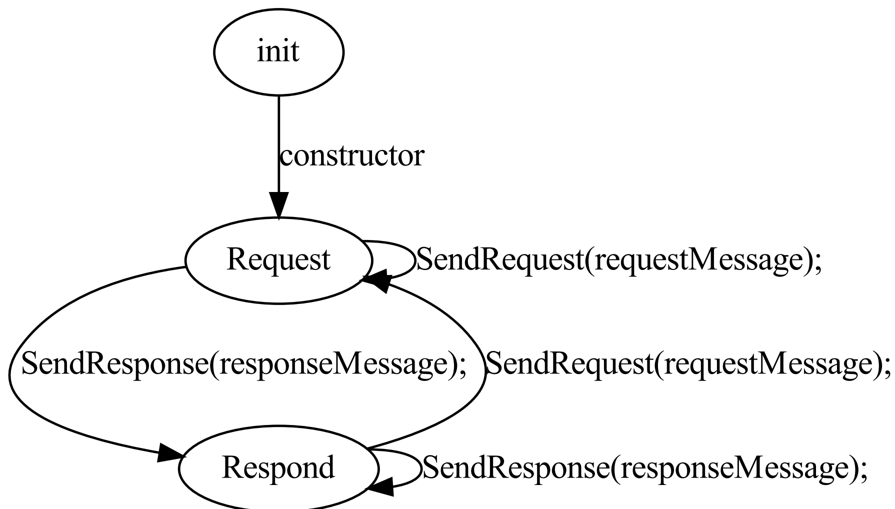

Fig. 4: Hello Blockchain Abstracción Estados

Se procede a realizar un análisis de la abstracción del contrato utilizando las nuevas precondiciones. Se puede observar en la figura 5 que en comparación con la abstracción anterior, los estados ya no se conectan consigo mismos.

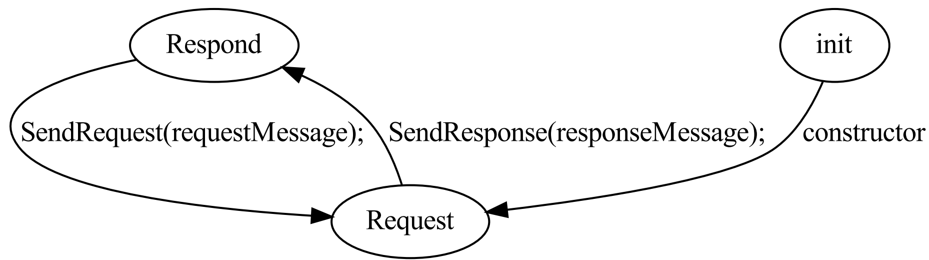

Fig. 5: Hello Blockchain con correcciones Abstracción Estados

Se presenta un ejemplo de un archivo de configuración utilizado en la versión corregida del programa en el Listing 5.

```
fileName = "HelloBlockchain_fixed.sol"
contractName = "HelloBlockchain"
functions = ["SendRequest(requestMessage);", "SendResponse(responseMessage);"]

statePreconditions = ["State == StateType.Respond",
"State == StateType.Request"]
functionPreconditions = ["msg.sender == Requestor",
"true"]

functionVariables = "uint requestMessage, uint responseMessage"
tool_output = "Found a counterexample"

statesModeState = [[1,0], [0,2]]
statesNamesModeState = ["Request", "Respond"]
statePreconditionsModeState = ["State == StateType.Request",
"State == StateType.Respond"]
txBound = 4
```

Listing 5: Configuración Hello Blockchain

4.2. Asset Transfer

Vamos a analizar el contrato más complicado de Benchmark de `Azure`, que es el AssetTransfer. Sus requerimientos están resumidos en la figura 6.

Este contrato cubre el escenario de compra y venta de productos de alto valor, que requieren un inspector y un tasador. Los vendedores pueden listar sus activos mediante la creación de una instancia. Los compradores pueden hacer ofertas haciendo una acción y otras partes pueden realizar acciones para inspeccionar o tasar el activo. Una vez que el activo se marca como inspeccionado y tasado, el comprador y el vendedor confirmarán la venta antes de que el contrato se complete. En cada punto del proceso, todos los participantes tienen visibilidad del estado del contrato a medida que se actualiza.

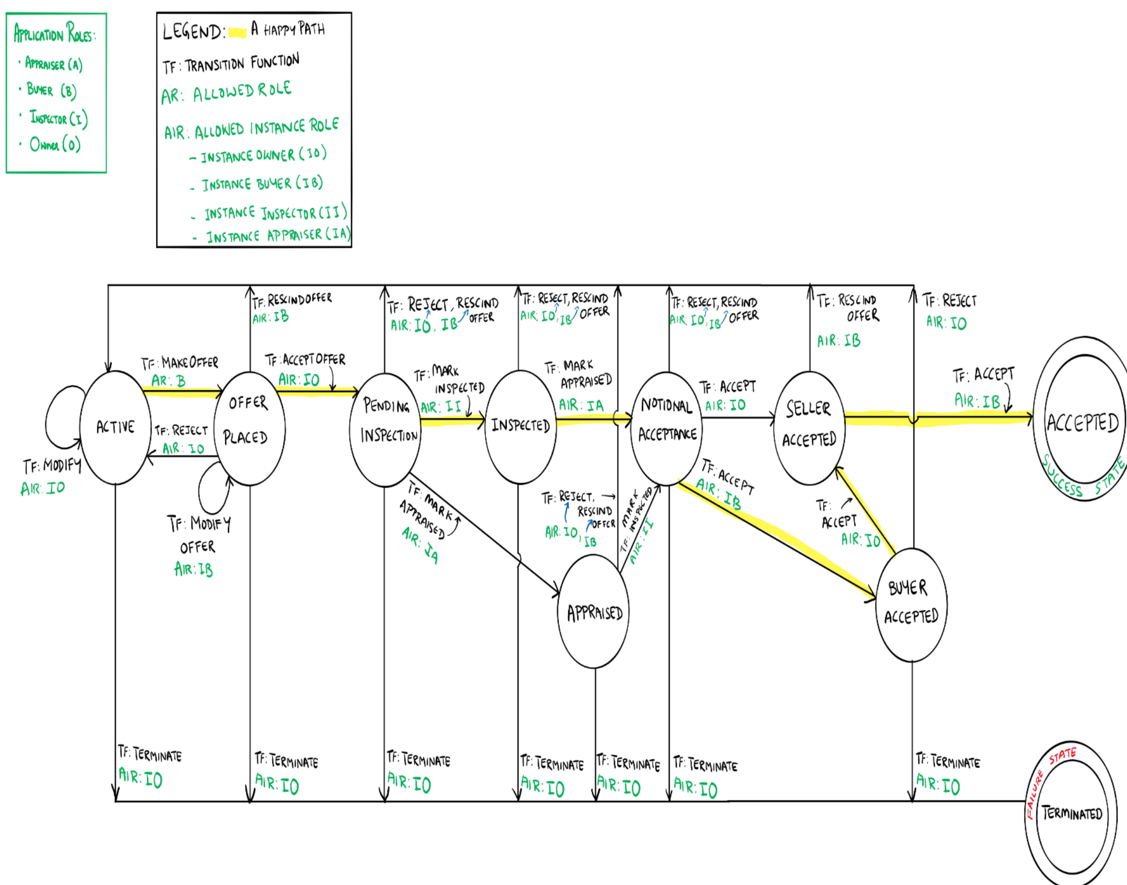

Fig. 6: Explicación de Azure de Asset Transfer

Los roles que interactúan con el contrato son los siguientes.

Nombre	Descripción
Vendedor	Una persona que posee un activo y quiere venderlo.
Comprador	Una persona que desea comprar el activo que está siendo vendido por el vendedor.
Inspector	Una persona elegida por el comprador para ser el inspector del activo que se está considerando comprar.
Tasador	Una persona elegida por el comprador para ser el tasador del activo que se está considerando comprar.

Los estados del contrato son:

Nombre	Descripción
Activo	Indica que un activo está disponible para ser comprado.
Oferta Realizada	Indica la intención de un comprador de comprar el activo.
Inspección Pendiente	Indica la solicitud de un comprador al Inspector para inspeccionar el activo que se está considerando.
Inspeccionado	Indica la aprobación del Inspector para comprar el activo que se está considerando.
Tasado	Indica la aprobación del Tasador para comprar el activo que se está considerando.
Aceptación no Oficial	Indica que tanto el Inspector como el Tasador han aprobado la compra del activo que se está considerando.
Aceptación del Vendedor	Indica la aprobación del propietario para aceptar la oferta realizada por el comprador.
Aceptación del Comprador	Indica la aprobación del comprador para la aprobación del propietario.
Aceptado	Indica que tanto el comprador como el vendedor han acordado la transferencia del activo que se está considerando.
Terminado	Indica la desaprobación del propietario para continuar vendiendo el activo que se está considerando.

```
1  pragma \verb|Solidity| >=0.4.25 <0.9.0;
2
3  contract AssetTransfer {
4
5      enum StateType { Active, OfferPlaced, PendingInspection, Inspected, Appraised,
       NotionalAcceptance, BuyerAccepted, SellerAccepted, Accepted, Terminated }
6      address public InstanceOwner;
7      string public Description;
8      uint public AskingPrice;
9      StateType public State;
10
11     address public InstanceBuyer;
12     uint public OfferPrice;
13     address public InstanceInspector;
14     address public InstanceAppraiser;
15
16     constructor(string memory description, uint256 price) public{
17         InstanceOwner = msg.sender;
18         AskingPrice = price;
19         Description = description;
20         State = StateType.Active;
21     }
22
23     function Terminate() public{
24         if (InstanceOwner != msg.sender) { revert(); }
25
26         State = StateType.Terminated;
27     }
28
29     function Modify(uint256 price) public{
```

```
30         if (State != StateType.Active) { revert(); }
31         if (InstanceOwner != msg.sender) { revert(); }
32
33         Description = description;
34         AskingPrice = price;
35     }
36
37     function MakeOffer(address inspector, address appraiser, uint256 offerPrice) public{
38         if (inspector == address(0x0) || appraiser == address(0x0) || offerPrice == 0){
39             revert();
40         }
41         if (State != StateType.Active) { revert(); }
42         if (InstanceOwner == msg.sender) { revert(); }
43
44         InstanceBuyer = msg.sender;
45         InstanceInspector = inspector;
46         InstanceAppraiser = appraiser;
47         OfferPrice = offerPrice;
48         State = StateType.OfferPlaced;
49     }
50
51     function AcceptOffer() public {
52         if (State != StateType.OfferPlaced) { revert(); }
53         if (InstanceOwner != msg.sender) { revert(); }
54         State = StateType.PendingInspection;
55     }
56
57     function Reject() public {
58         if (State != StateType.OfferPlaced && State != StateType.PendingInspection && State !=
    StateType.Inspected && State != StateType.Appraised && State != StateType.NotionalAcceptance
     && State != StateType.BuyerAccepted) {
59             revert();
60         }
61         if (InstanceOwner != msg.sender) { revert(); }
62
63         InstanceBuyer = address(0x0);
64         State = StateType.Active;
65     }
66
67     function Accept() public {
68         if (msg.sender != InstanceBuyer && msg.sender != InstanceOwner) { revert(); }
69
70         if (msg.sender == InstanceOwner &&
71             State != StateType.NotionalAcceptance &&
72             State != StateType.BuyerAccepted) { revert(); }
73
74         if (msg.sender == InstanceBuyer &&
75             State != StateType.NotionalAcceptance &&
76             State != StateType.SellerAccepted) { revert(); }
77
78         if (msg.sender == InstanceBuyer) {
79             if (State == StateType.NotionalAcceptance) {
80                 State = StateType.BuyerAccepted;
81             }
82             else if (State == StateType.SellerAccepted) {
83                 State = StateType.Accepted;
84             }
85         }
86         else {
87             if (State == StateType.NotionalAcceptance) {
88                 State = StateType.SellerAccepted;
89             }
90             else if (State == StateType.BuyerAccepted) {
91                 State = StateType.Accepted;
92             }
93         }
94     }
95
96     function ModifyOffer(uint256 offerPrice) public {
97         if (State != StateType.OfferPlaced) { revert(); }
98         if (InstanceBuyer != msg.sender || offerPrice == 0) { revert(); }
99
100        OfferPrice = offerPrice;
101    }
102
103    function RescindOffer() public {
104        if (State != StateType.OfferPlaced && State != StateType.PendingInspection && State !=
    StateType.Inspected && State != StateType.Appraised && State != StateType.NotionalAcceptance
     && State != StateType.SellerAccepted) {
105            revert();
106        }
107        if (InstanceBuyer != msg.sender) { revert(); }
108
109        InstanceBuyer = address(0x0);
110        OfferPrice = 0;
111        State = StateType.Active;
112    }
113
114    function MarkAppraised() public {
115        if (InstanceAppraiser != msg.sender) { revert(); }
116
```

```
117         if (State == StateType.PendingInspection) {
118             State = StateType.Appraised;
119         }
120         else if (State == StateType.Inspected) {
121             State = StateType.NotionalAcceptance;
122         }
123         else   { revert(); }
124     }
125
126     function MarkInspected() public {
127         if (InstanceInspector != msg.sender) { revert(); }
128
129         if (State == StateType.PendingInspection) {
130             State = StateType.Inspected;
131         }
132         else if (State == StateType.Appraised) {
133             State = StateType.NotionalAcceptance;
134         }
135         else { revert(); }
136     }
137 }
```

Listing 6: Contrato Asset Transfer

En la figura 7 a continuación se presenta el primer intento de generar la abstracción EPA. Como se puede observar, aún faltan las transiciones *Accept* entre los estados *NotionalAcceptance* y *SellerAccepted*, entre *NotionalAcceptance* y *BuyerAccepted*, entre *BuyerAccepted* y *SellerAccepted*, y entre *SellerAccepted* y *Accepted* de la figura 6.

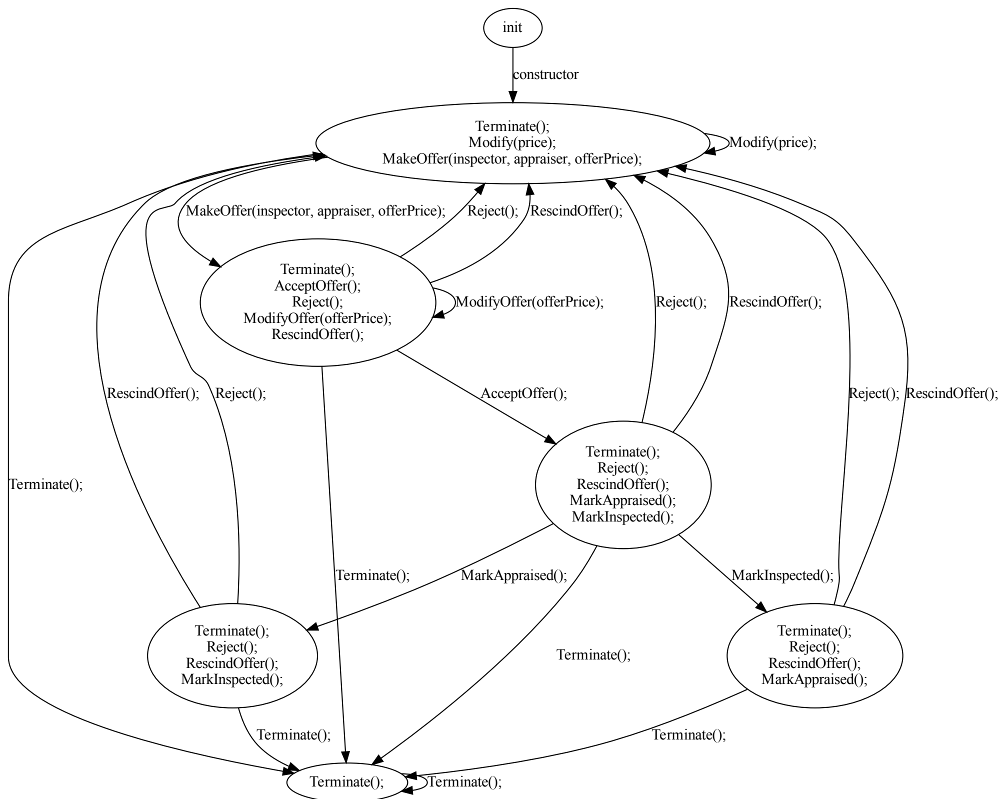

Fig. 7: Asset Transfer Abstración EPA tx4

Después de realizar una revisión exhaustiva para comprender la causa de este problema, se detectó que la limitación en la cantidad de transiciones exploradas se debe a la restricción

impuesta por el parámetro de configuración `txBound` de `VeriSol`. Se ha tomado la decisión de modificar el valor predeterminado de la variable ya que establece un límite en la cantidad de acciones que se pueden aplicar a una instancia para alcanzar el estado deseado. En los contratos que involucran múltiples acciones, esta restricción puede afectar la capacidad de explorar todas las transiciones relevantes en la abstracción generada.

Por lo tanto, se incrementó el valor de `txBound` para poder visualizar todas las transiciones posibles en la abstracción generada. De esta manera, se logró obtener una abstracción completa del contrato en cuestión como se observa en la figura 8.

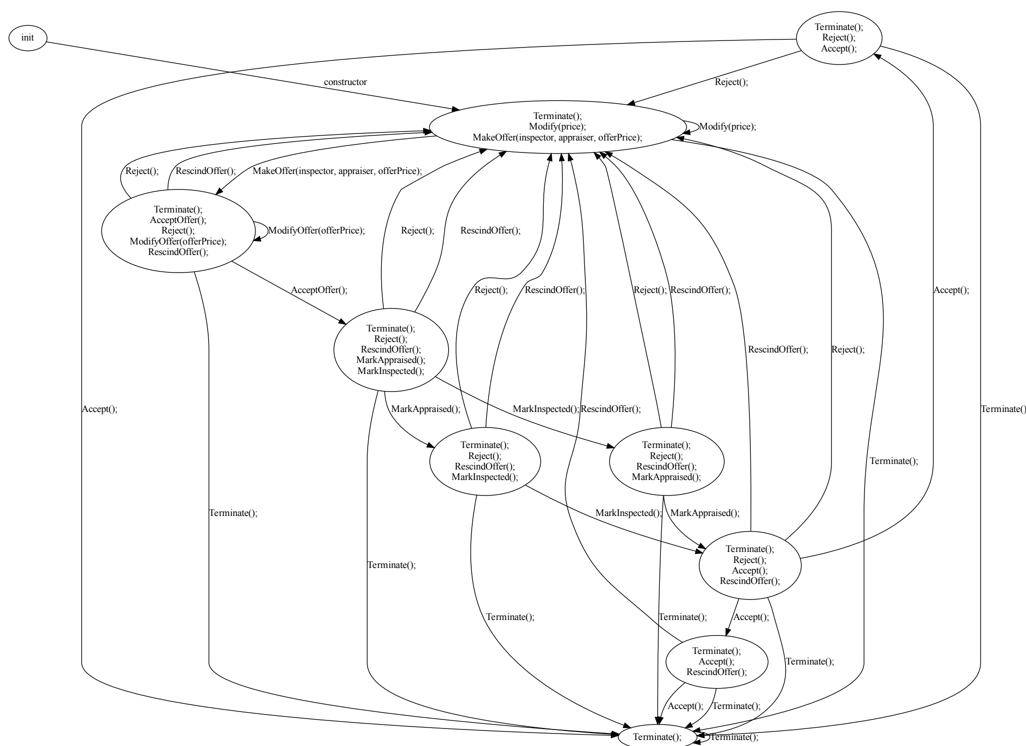

Fig. 8: Asset Transfer Abstración EPA tx8

Después de haber tenido problemas con la abstracción del contrato, se decidió cambiar el parámetro txBound utilizado por `VeriSol` en la configuraciones. Se estableció txBound igual al número de funciones públicas del contrato.

Sin embargo, en la figura 8 se observó que el estado *Terminated* no funcionaba como se esperaba ya que nunca se puede llegar a un estado vacío. Esto se debió a que la precondición de esta función es siempre verdadera. Al cambiar la precondición, se logró obtener una abstracción más precisa y completa del contrato como se nota en la figura 9.

La abstracción de estados de este contrato es la que se puede ver en la figura 10.

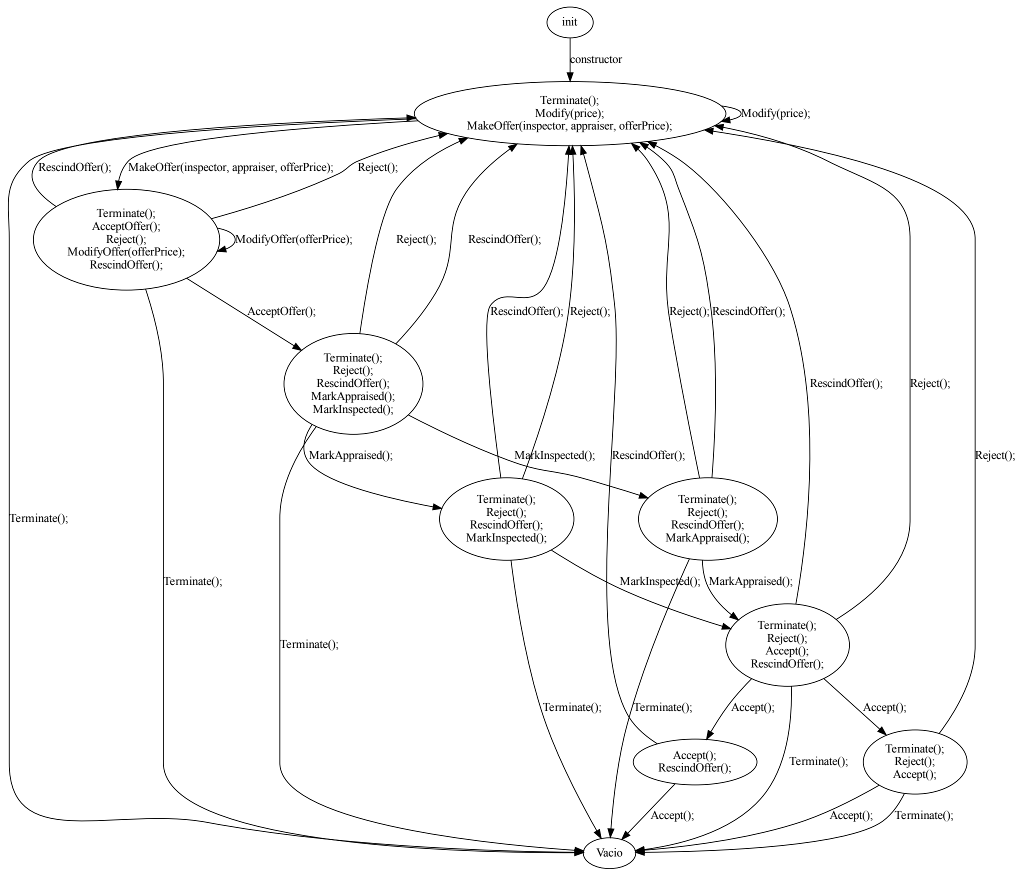

Fig. 9: Asset Transfer con Fix Abstración EPA

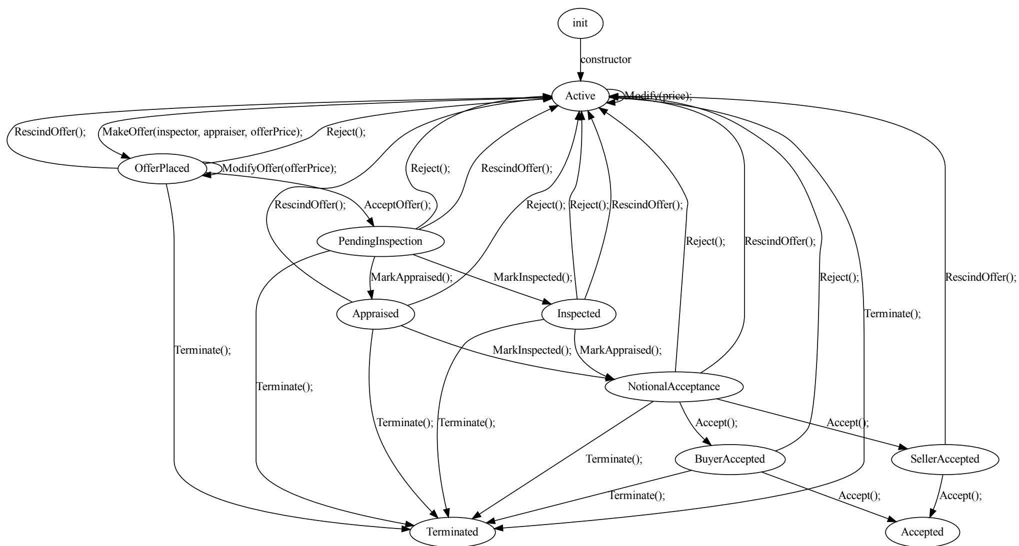

Fig. 10: Asset Transfer con Fix Abstración Estados

4.3. Auction

A continuación se presenta el análisis del contrato *Auction* del benchmark `SmartPulse`. Este contrato simula una subasta donde se pueden realizar ofertas y retirarlas. Una vez que se alcanza un cierto número de bloques, la subasta finaliza.

En la revisión del contrato se encontró un error en la función *AuctionEnded*. El problema radica en que la variable booleana *ended* nunca llega a cambiar a verdadero, lo que indica que la subasta ha finalizado.

```
1  pragma \verb|Solidity| ^0.5.0;
2
3  contract Auction {
4      uint auctionStart;
5      uint biddingTime;
6      address payable beneficiary;
7
8      bool ended = false;
9      address payable highestBidder = address(0x0);
10     address payable A = address(0x0);
11     uint highestBid = 0;
12     mapping(address => uint) pendingReturns;
13     address[] pendingReturnsArray = new address[](0);
14     uint blockNumber;
15
16     constructor(uint _auctionStart, uint _biddingTime, address payable _beneficiary,
17     address payable payable_a, uint _blockNumber) public {
18         auctionStart = _auctionStart;
19         biddingTime = _biddingTime;
20         beneficiary = _beneficiary;
21         blockNumber = _blockNumber;
22     }
23
24     function Bid() public payable {
25         uint end = auctionStart + biddingTime;
26         if(end < blockNumber || ended) { revert(); }
27         else {
28             if(msg.value <= highestBid) { revert(); }
29             else {
30                 pendingReturns[highestBidder] += highestBid;
31                 if (highestBidder != address(0x0)) {
32                     pendingReturnsArray.push(highestBidder);
33                 }
34                 highestBidder = msg.sender;
35                 highestBid = msg.value;
36             }
37         }
38         t();
39     }
40
41     function Withdraw() public {
42         if(pendingReturns[msg.sender] != 0 && pendingReturnsArray.length != 0) {
43             uint pr = pendingReturns[msg.sender];
44             pendingReturns[msg.sender] = 0;
45             pendingReturnsArray = remove(msg.sender, pendingReturnsArray);
46             msg.sender.transfer(pr);
47         }
48         else {
49             revert();
50         }
51         t();
52     }
53
54     function remove(address _valueToFindAndRemove, address[] memory _array)
55     public returns(address[] memory) {
56         auxArray = new address[](0);
57         for (uint i = 0; i < _array.length; i++) {
58             if(_array[i] != _valueToFindAndRemove) {
59                 auxArray.push(_array[i]);
60             }
61         }
62         return auxArray;
63     }
64
65     function AuctionEnd() public {
66         uint end = auctionStart + biddingTime;
67
68         //!ended is a bug
69         if(blockNumber <= end || !ended) { revert(); }
70         else {
71             ended = true;
72             beneficiary.transfer(highestBid);
73         }
74         t();
75     }
76
77     function t() public {
```

```
78          blockNumber = blockNumber + 1;
79      }
80  }
```

Listing 7: Contrato Auction

En la implementación del contrato, se han hecho las modificaciones necesarias para que sea interpretable con la herramienta `VeriSol` como fue explicado en la sección 3.4. Esto incluye la presencia de la función `t()`, la variable *blockNumber* y el uso de un array para mantener los estados del mapping.

Ahora, se desea saber cómo se vería una abstracción generada por `VeriSol` donde se sabe que hay estados inalcanzables por los resultados obtenidos en el estudio [1]. Si se compara con los resultados obtenidos en este informe [1], en la figura 12 se puede observar un resultado distinto. Se puede ver que con `VeriSol` en la figura 11 no se logra graficar la parte inalcanzable del contrato. Esto se debe a que esta herramienta, para validar los invariantes definidos automáticamente, construye instancias del contrato ejecutando sus métodos públicos. Esto imposibilita validar precondiciones de estados inalcanzables.

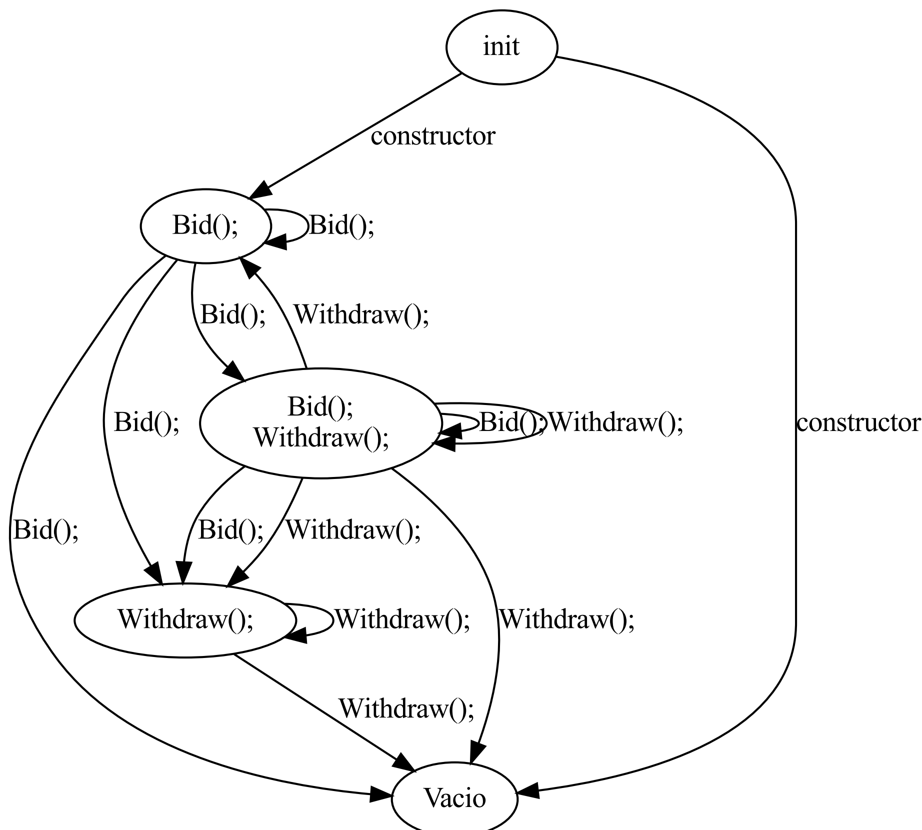

Fig. 11: Auction Abstración EPA

Se pueden realizar modificaciones interesantes con el programa presentado en este trabajo, que consiste en la modificación de la configuración del contrato para analizar un

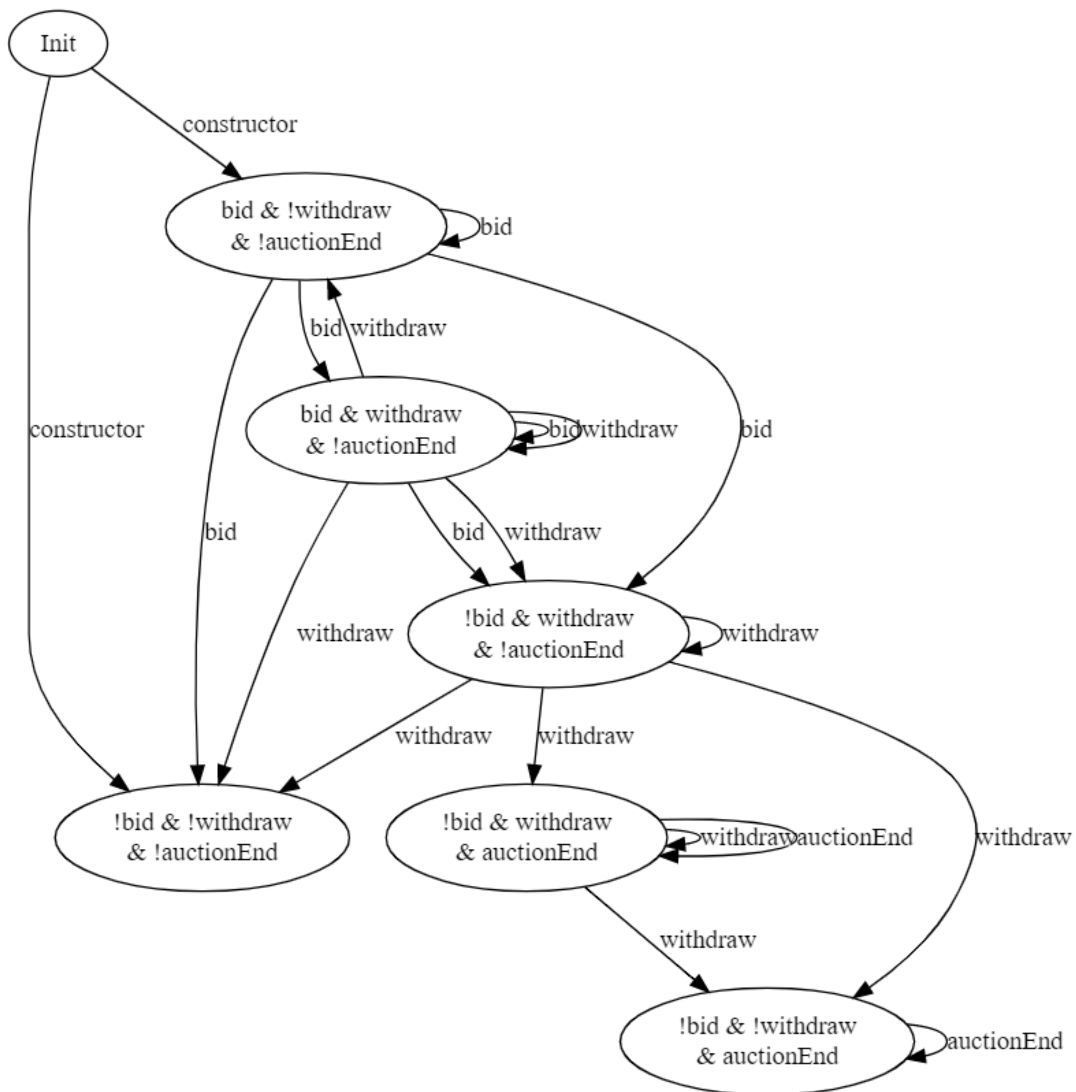

Fig. 12: Auction Abstración EPA con Alloy

conjunto más amplio de condiciones en la abstracción generada.

Supongamos que un auditor desea ver una abstracción que refleje la interacción entre diferentes usuarios del contrato. La EPA no permite generar esta abstracción refinada. Sin embargo, como se comentó en la sección anterior, es posible ingresar precondiciones para particionar el estado de diferentes formas según el interés del auditor. En este caso, el auditor podría ingresar predicados donde se establecen condiciones específicas para cada usuario, lo que permitiría analizar escenarios más detallados de interacción entre ellos.

Es importante destacar que al realizar estas modificaciones, puede ser necesario agregar nuevas funciones al contrato para representar las diferentes condiciones predefinidas. Aunque estas funciones pueden ser similares en su estructura, las precondiciones específicas permiten dividir el estado del contrato en diferentes formas, lo que proporciona una visión más precisa y detallada de las interacciones entre los usuarios en la abstracción generada.

Por ejemplo, se puede agregar más información a un contrato específico, como en el caso del contrato *Auction*. Acá se puede analizar la abstracción pero enfocándose en un bidder específico, A. Esto implica que se pueden realizar acciones como $Withdraw$ desde la dirección A o desde una dirección diferente a A. Esta modificación permite obtener una abstracción más detallada y específica del contrato, lo que resulta útil para detectar posibles bugs o vulnerabilidades.

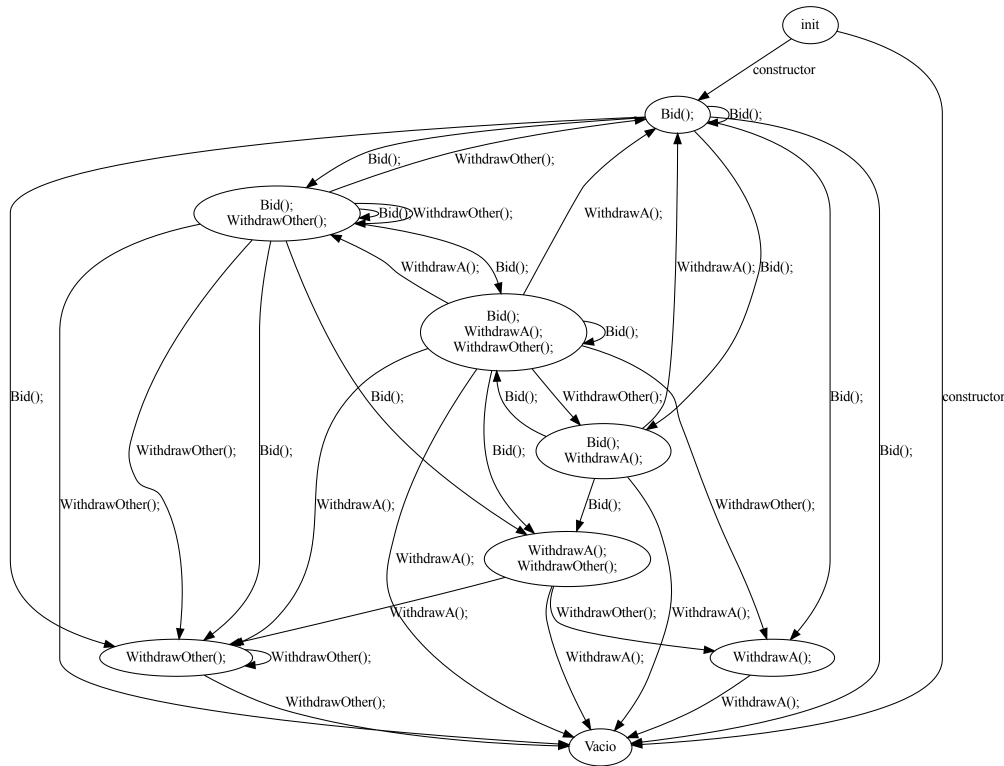

Fig. 13: Auction con Withdraw Abstración EPA

En la figura 13 se observa la abstracción al incluir las funciones $WithdrawA$ y $WithdrawNoA$ en nuestro contrato. Se establece una dirección A en el constructor. Si se ha realizado una

oferta con la dirección A, entonces se permite hacer un $withdrawA$.

Estas abstracciones resultaron diferentes al estudio [1] ya que en este, había un orden preestablecido entre las funciones $withdrawA$ y $withdrawOther$.

Además, es posible agregar observaciones a nuestro contrato utilizando abstracciones de states. Se pueden agregar variables del contrato como condiciones en los estados que se definen. Esto se puede aplicar para obtener más información sobre el comportamiento del contrato en distintas situaciones.

4.4. CrowdFunding

El contrato *Crowdfunding* del Benchmark `SmartPulse` es una implementación de una plataforma de financiamiento colectivo. Este contrato es creado mediante un constructor que recibe tres parámetros: el objetivo de financiación, el propietario del contrato y una fecha límite para las donaciones.

Una vez que el contrato es deployado en la blockchain, terceros pueden hacer donaciones de fondos. Una vez que la fecha límite establecida ha pasado, se verifica si se ha alcanzado el objetivo de financiación. Si se ha alcanzado, el propietario del contrato puede recibir todos los fondos donados. En caso contrario, los donantes pueden reclamar sus donaciones.

```

pragma \verb|Solidity| ^0.5.0;

contract Crowdfunding {
    address payable owner;
    uint max_block;
    uint goal;
    uint blockNumber;

    mapping(address => uint) backers;
    address[] backersArray = new address[](0);
    address[] auxArray;
    uint countBackers = 0;
    bool funded = false;
    uint balance = 0;

    constructor(address payable _owner, uint _max_block, uint _goal, uint _blockNumber) public {
        owner = _owner;
        max_block = _max_block;
        goal = _goal;
        balance = 0;
        blockNumber = _blockNumber;
    }

    function Donate() public payable {
        if(max_block <= blockNumber) { revert(); }
        else {
            if(backers[msg.sender] == 0) {
                backers[msg.sender] = msg.value;
                backersArray.push(msg.sender);
                balance = balance + msg.value;
            }
            else { revert(); }
        }
    }

    function GetFunds() public {
        if(max_block < blockNumber && msg.sender == owner) {
            if(goal <= balance) {
                funded = true;
                owner.transfer(balance);
                balance = 0;
            }
            else { revert(); }
        }
        else { revert(); }
    }

    function Claim() public {
        if(blockNumber <= max_block) { revert(); }
        else {
            if(backers[msg.sender] == 0 || backersArray.length == 0 || funded || goal <= balance
 ) {
                revert();
            }
            else {
                uint val = backers[msg.sender];
                backers[msg.sender] = 0;
                backersArray = remove(msg.sender, backersArray);
                msg.sender.transfer(val);
                balance = balance - val;
            }
        }
    }

    function t() public {
        blockNumber = blockNumber + 1;
    }
}
```

Listing 8: Contrato Crowdfunding

En la figura 14 se presenta la abstracción EPA que se obtiene para el contrato en cuestión.

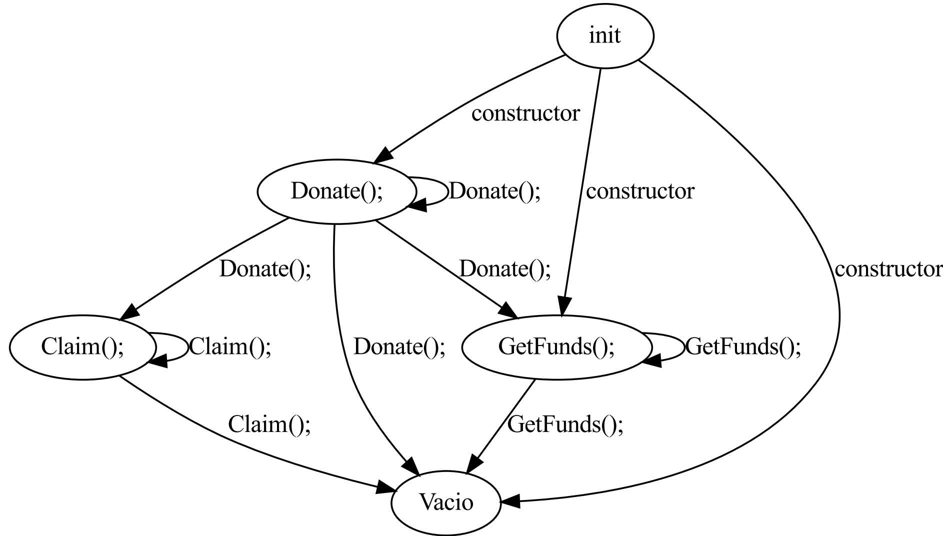

Fig. 14: Crowdfunding Abstración EPA

En este contrato se pueden realizar donaciones, claims o refunds, y es importante analizar el estado del balance al finalizar el funcionamiento del contrato. Para lograr esto, se pueden definir estados en la abstracción states que incluyan la información del balance. De esta manera, podemos agregar a la abstracción states el estado del balance al final del contrato. Se puede ver en el Listing 9 como son las precondiciones de estos estados.

```
1  ["Vacio sin balance", "Vacio con balance", "Donate", "Funds", "Claim"]
2  "(!(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance)
3  && !(blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)
4  && balance == 0)",
5  "(!(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance)
6  && !(blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)
7  && balance > 0)",
8  "(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance)
9  && !(blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)",
10 "!(max_block > blockNumber) && (max_block < blockNumber && goal <= balance)
11 && !(blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)",
12 "!(max_block > blockNumber) && !(max_block < blockNumber && goal <= balance)
13 && (blockNumber > max_block && !funded && goal > balance && backersArray.length != 0)",
```

Listing 9: Estados Crowdfunding

Se observa en la figura 15 el resultado de la abstracción states, imitando una abstracción EPA con información del balance incluida.

La abstracción de la figura 16 se obtiene al incluir la función tau en la observación.

Al analizar esta abstracción, se observa rápidamente un error en el contrato: no es posible ejecutar ninguna función cuando $block.Number = maxNumber$. En este caso, la transición lleva al estado vacío, donde se puede ejecutar la función tau y avanzar hacia otro estado.

Este hallazgo es significativo, ya que pone de manifiesto una restricción importante en el contrato que impide la ejecución de funciones cuando se alcanza el valor máximo de

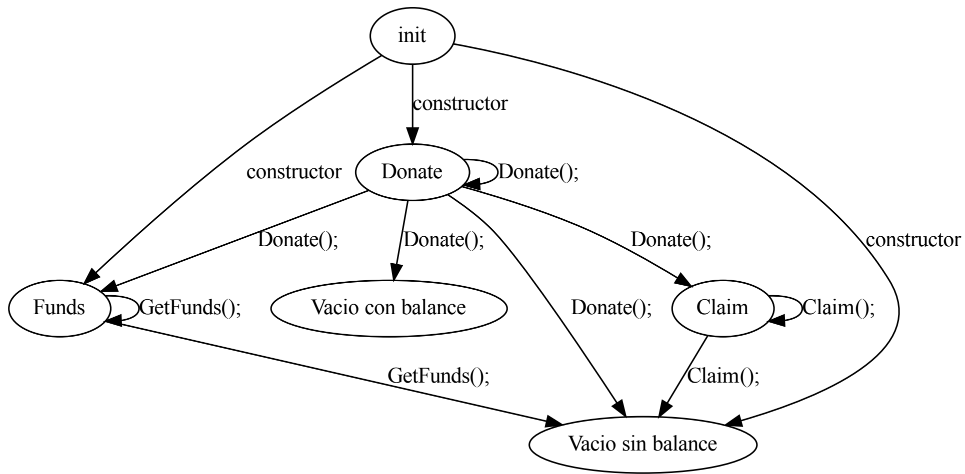

Fig. 15: Crowdfunding Abstración Estados con balance

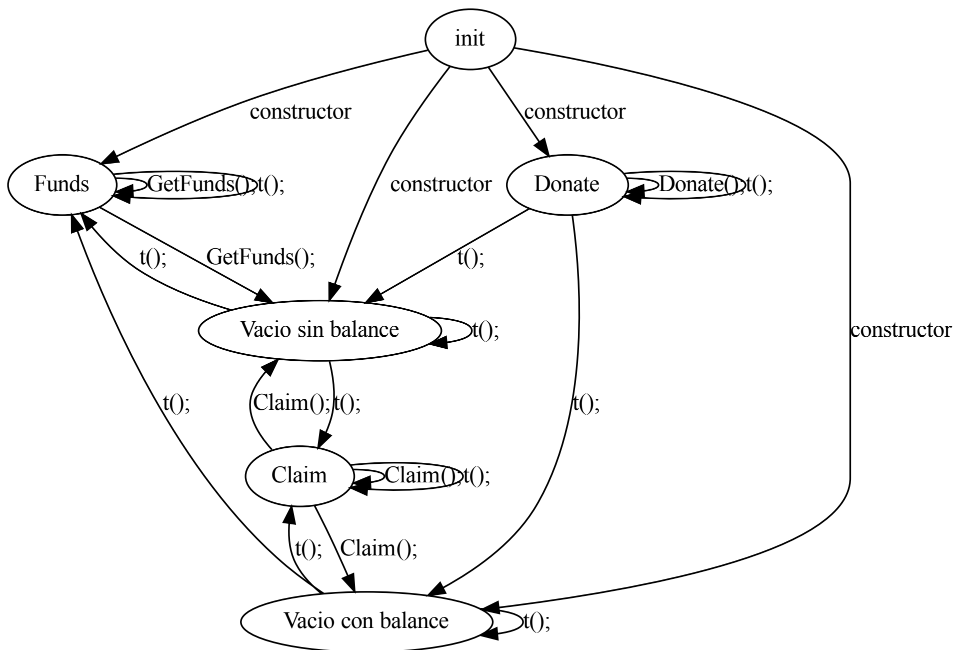

Fig. 16: Crowdfunding con tau Abstración Estados con balance

$block.Number$. La inclusión de la función tau en la abstracción permite identificar este comportamiento no deseado y proporciona una representación visual clara de la transición al estado vacío, donde se puede ejecutar la función tau y avanzar a estados subsiguientes.

4.5. Rock Papper Scissors

Es de nuestro interés observar la abstracción del contrato *RockPapperScissor*. Este es un contrato inteligente que permite a los jugadores jugar el juego de piedra, papel y tijera en la cadena de bloques *Ethereum*.

```
1      pragma \verb|Solidity| ^0.5.0;
2
3  contract RockPaperScissors {
4      address payable player1;
5      address payable player2;
6      address payable owner;
7
8      int p1Choice = -1;
9      int p2Choice = -1;
10     //mapping(uint => mapping(uint => uint)) payoffMatrix;
11     uint[3][3] public  payoffMatrix;
12
13     constructor(address payable _player1, address payable _player2, address payable _owner)
        public {
14         player1 = _player1;
15         player2 = _player2;
16         owner = _owner;
17
18         //Rock - 0
19         //Paper - 1
20         //Scissors - 2
21         payoffMatrix[0][0] = 0;
22         payoffMatrix[0][1] = 2;
23         payoffMatrix[0][2] = 1;
24         payoffMatrix[1][0] = 1;
25         payoffMatrix[1][1] = 0;
26         payoffMatrix[1][2] = 2;
27         payoffMatrix[2][0] = 2;
28         payoffMatrix[2][1] = 1;
29         payoffMatrix[2][2] = 0;
30     }
31
32     function choicePlayer1(int choice) public {
33         if(msg.sender == player1) {
34             if(p1Choice == -1 && choice >= 0 && choice <= 2) {
35                 p1Choice = choice;
36             }
37             else { revert(); }
38         }
39         else { revert(); }
40     }
41
42     function choicePlayer2(int choice) public {
43         if(msg.sender == player2) {
44             if(p2Choice == -1 && choice >= 0 && choice <= 2) {
45                 p2Choice = choice;
46             }
47             else { revert(); }
48         }
49         else { revert(); }
50     }
51
52     function determineWinner() public {
53         if(p1Choice != -1 && p2Choice != -1) {
54             uint winner = payoffMatrix[uint(p1Choice)][uint(p2Choice)];
55             if(winner == 1) {
56                 player1.transfer(address(this).balance);
57             }
58             else if(winner == 2) {
59                 player2.transfer(address(this).balance);
60             }
61             else {
62                 owner.transfer(address(this).balance);
63             }
64         }
65         else { revert(); }
66     }
67 }
```

Listing 10: Contrato RockPaperScissors

En la figura 17 se puede observar cómo es la EPA generada con nuestra herramienta.

Si generamos la abstracción de estados mirando los ganadores podemos obtener una mejor comprensión de la funcionalidad del contrato *RockPaperScissors*, como se nota en la figura 18. Al hacerlo, podemos ver los posibles estados en los que el contrato puede estar, dependiendo del resultado del juego. Por ejemplo, podemos distinguir entre un estado en el que el jugador 1 gana, el jugador 2 gana o hay un empate.

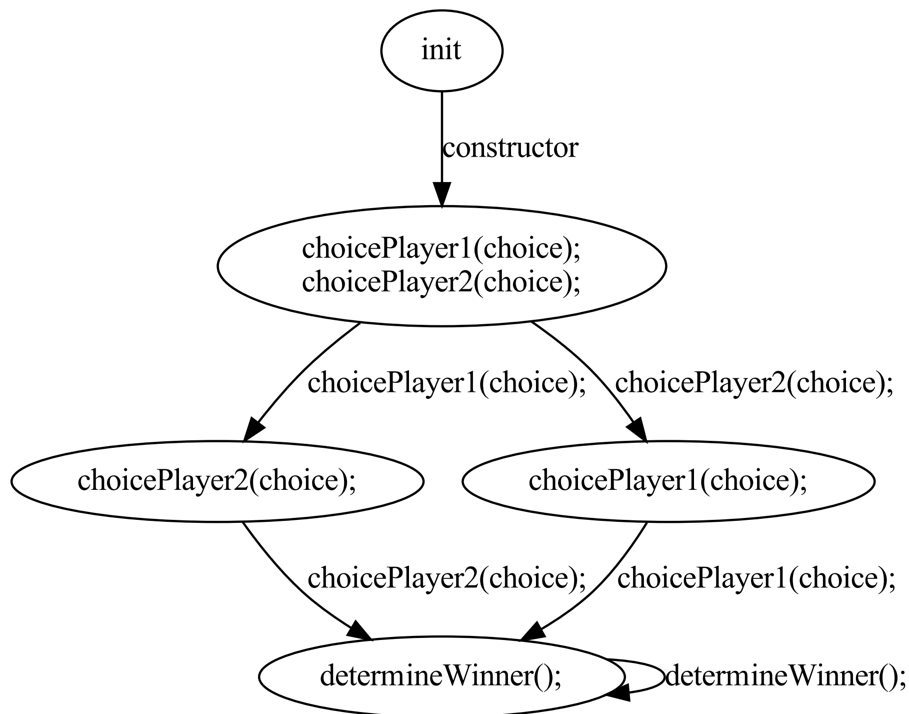

Fig. 17: RockPaperScissors Abstracción EPA

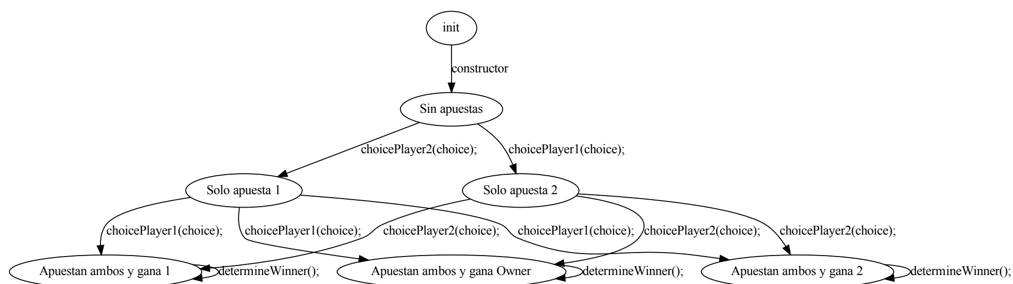

Fig. 18: RockPaperScissors Abstracción Estados

Es de utilidad esta abstracción para observar mejor la funcionalidad de un contrato. Por ejemplo, si queremos comprobar que ambos jugadores puedan ganar es una buena abstracción.

4.6. EPXCrowdSale

Es de nuestro interés analizar este último contrato del Benchmark `SmartPulse`. Es similar al crowdfunding, pero los fondos se recaudan a cambio de tokens. El contrato *EPXCrowdsale* es un contrato inteligente en la blockchain de *Ethereum* que se utiliza para realizar una venta de tokens a cambio de Ether. El contrato tiene un límite máximo de Ether que puede ser recaudado durante el crowdsale y una cantidad máxima de tokens que se pueden crear. El contrato permite a los inversores enviar Ether al contrato y recibir tokens en función de la tasa de cambio establecida.

El contrato tiene una variable de estado booleana llamada *isCrowdSaleClosed* que indica si la venta de tokens está abierta o cerrada. El contrato también tiene una función *refund* que permite a los inversores solicitar un reembolso de su Ether si la venta de tokens está cerrada.

Cuando se cierra la venta de tokens, si se alcanza el objetivo de recaudación, los Ether recaudados se transfieren al beneficiario designado. Si no se alcanza el objetivo de recaudación, los Ether se devuelven a los inversores mediante la función *refund*.

Si se emiten reembolsos, los Ether se devolverán a los inversores de manera eventual. Si el beneficiario intenta reclamar los fondos recaudados, se le enviarán todos los fondos utilizados para comprar tokens.

Se muestra una versión simplificada del contrato en la 11.

```
1
2  pragma \verb|Solidity| ^0.5.0;
3
4
5  contract EPXCrowdsale is owned, safeMath {
6      ...
7    bool     public isCrowdSaleClosed                = false;      // crowdsale completion boolean
8    bool     private areFundsReleasedToBeneficiary  = false;      // boolean for founder to receive
       Eth or not
9    bool     public isCrowdSaleSetup                 = false;      // boolean for crowdsale setup
10
11   mapping(address => uint256) balancesArray;
12   mapping(address => uint256) usersEPXfundValue;
13
14   // default function, map admin
15   /* function EPXCrowdsale() public onlyOwner { */
16   constructor(uint256 _blockNumber) public onlyOwner {
17     admin = msg.sender;
18     CurrentStatus = "Crowdsale deployed to chain";
19     blockNumber = _blockNumber;
20   }
21
22   // setup the CrowdSale parameters
23   function SetupCrowdsale(uint256 _fundingStartBlock, uint256 _fundingEndBlock) public onlyOwner
        returns (bytes32 response) {
24     if ((msg.sender == admin)
25     && (!(isCrowdSaleSetup))
26   && (!(beneficiaryWallet != address(0x0)))) {
27       // init addresses
28       beneficiaryWallet                    = address(0
      x7A29e1343c6a107ce78199F1b3a1d2952efd77bA);
29       tokenReward                          = StandardToken(address(0
      x35BAA72038F127f9f8C8f9B491049f64f377914d));
30
31       // funding targets
32       fundingMinCapInWei                   = 3;                          // ETH 300 +
      000000000000000000 18 dec wei
33       amountRaisedInWei                    = 0;
34       initialTokenSupply                   = 6;                                  // 20,000,000 +
      4 dec resolution
35       tokensRemaining                      = initialTokenSupply;
36       fundingStartBlock                    = _fundingStartBlock;
37       fundingEndBlock                      = _fundingEndBlock;
38       isCrowdSaleSetup                     = true;
39       isCrowdSaleClosed                    = false;
40       CurrentStatus                        = "Crowdsale is setup";
41       return "Crowdsale is setup";
42     } else if (msg.sender != admin) {
43       //return "not authorised";
44       revert();
45     } else {
46       //return "campaign cannot be changed";
47       revert();
```

```
48      }
49    }
50
51    function checkPrice() internal view returns (uint256 currentPriceValue) {
52      ...
53    }
54
55    function buy() public payable {
56      // 0. conditions (length, crowdsale setup, zero check, exceed funding contrib check,
        contract valid check, within funding block range check, balance overflow check etc)
57      require(!(msg.value == 0)
58      /* && (msg.data.length == 0) */
59      && (blockNumber <= fundingEndBlock)
60      && (blockNumber >= fundingStartBlock)
61      && (tokensRemaining > 0));
62      ...
63    }
64
65    function checkGoalReached() public onlyOwner { // return crowdfund status to owner for each
        result case, update public vars
66      // update state & status variables
67      require (isCrowdSaleSetup);
68      if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber <= fundingEndBlock &&
        blockNumber >= fundingStartBlock)) { // ICO in progress, under softcap
69        areFundsReleasedToBeneficiary = false;
70        isCrowdSaleClosed = false;
71        CurrentStatus = "In progress (Eth < Softcap)";
72      } else if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber < fundingStartBlock)) {
        // ICO has not started
73        areFundsReleasedToBeneficiary = false;
74        isCrowdSaleClosed = false;
75        CurrentStatus = "Crowdsale is setup";
76      } else if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber > fundingEndBlock)) { //
         ICO ended, under softcap
77        areFundsReleasedToBeneficiary = false;
78        isCrowdSaleClosed = true;
79        CurrentStatus = "Unsuccessful (Eth < Softcap)";
80      } else if ((amountRaisedInWei >= fundingMinCapInWei) && (tokensRemaining == 0)) { // ICO
        ended, all tokens bought!
81        areFundsReleasedToBeneficiary = true;
82        isCrowdSaleClosed = true;
83        CurrentStatus = "Successful (EPX >= Hardcap)!";
84      } else if ((amountRaisedInWei >= fundingMinCapInWei) && (blockNumber > fundingEndBlock) && (
        tokensRemaining > 0)) { // ICO ended, over softcap!
85        areFundsReleasedToBeneficiary = true;
86        isCrowdSaleClosed = true;
87        CurrentStatus = "Successful (Eth >= Softcap)!";
88      } else if ((amountRaisedInWei >= fundingMinCapInWei) && (tokensRemaining > 0) && (
        blockNumber <= fundingEndBlock)) { // ICO in progress, over softcap!
89        areFundsReleasedToBeneficiary = true;
90        isCrowdSaleClosed = false;
91        CurrentStatus = "In progress (Eth >= Softcap)!";
92      }
93    }
94
95    function refund() public { // any contributor can call this to have their Eth returned. user's
         purchased EPX tokens are burned prior refund of Eth.
96      //require minCap not reached
97      require ((amountRaisedInWei < fundingMinCapInWei)
98      && (isCrowdSaleClosed)
99      && (blockNumber > fundingEndBlock)
100     && (usersEPXfundValue[msg.sender] > 0) && usersEPXfundValueArray.length != 0);
101
102       //burn user's token EPX token balance, refund Eth sent
103     uint256 ethRefund = usersEPXfundValue[msg.sender];
104     balancesArray[msg.sender] = 0;
105     usersEPXfundValue[msg.sender] = 0;
106
107       //send Eth back
108       msg.sender.transfer(ethRefund);
109   }
110 }
```

Listing 11: Contrato EPXCrowdSale

Al comparar las abstracciones generadas por el programa de este informe en la figura 19 y la abstracción del informe [1] con la figura 20, se puede notar que ambas son similares, con excepción de la transición $CheckGoalReach$ del estado $CheckGoalReach, Refund$ que va a $CheckGoalReach$ en la abstracción de la referencia.

La razón detrás de esta diferencia es que en la abstracción de 20 está utilizando una invariante débil que no tiene en cuenta la condición $fundingStartBlock < fundingEndBlock$ de la función *buy*. En cambio, en la abstracción con $VeriSol$ se considera esta condición, ya que para ejecutar la acción de refund es necesario realizar previamente la acción de *buy*

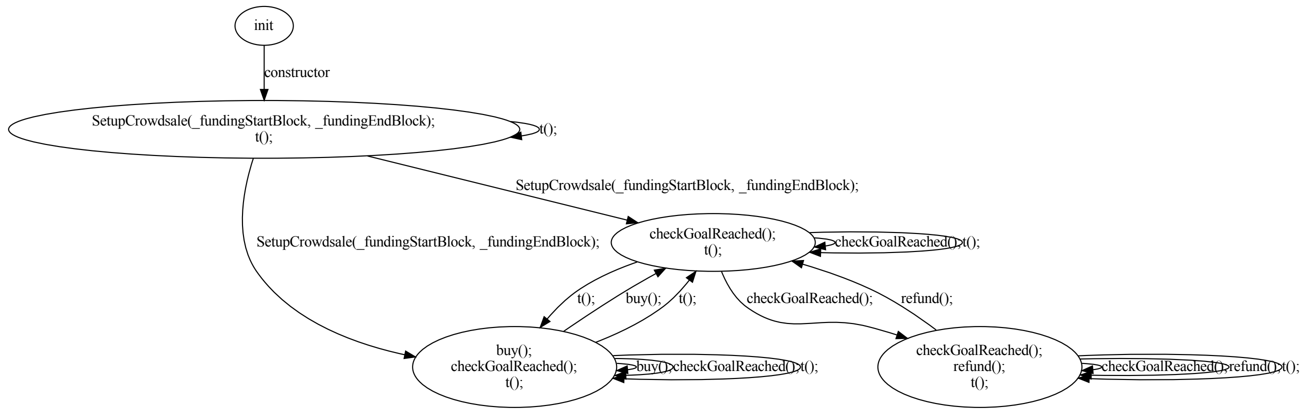

Fig. 19: EPXCrowdSale Abstracción EPA

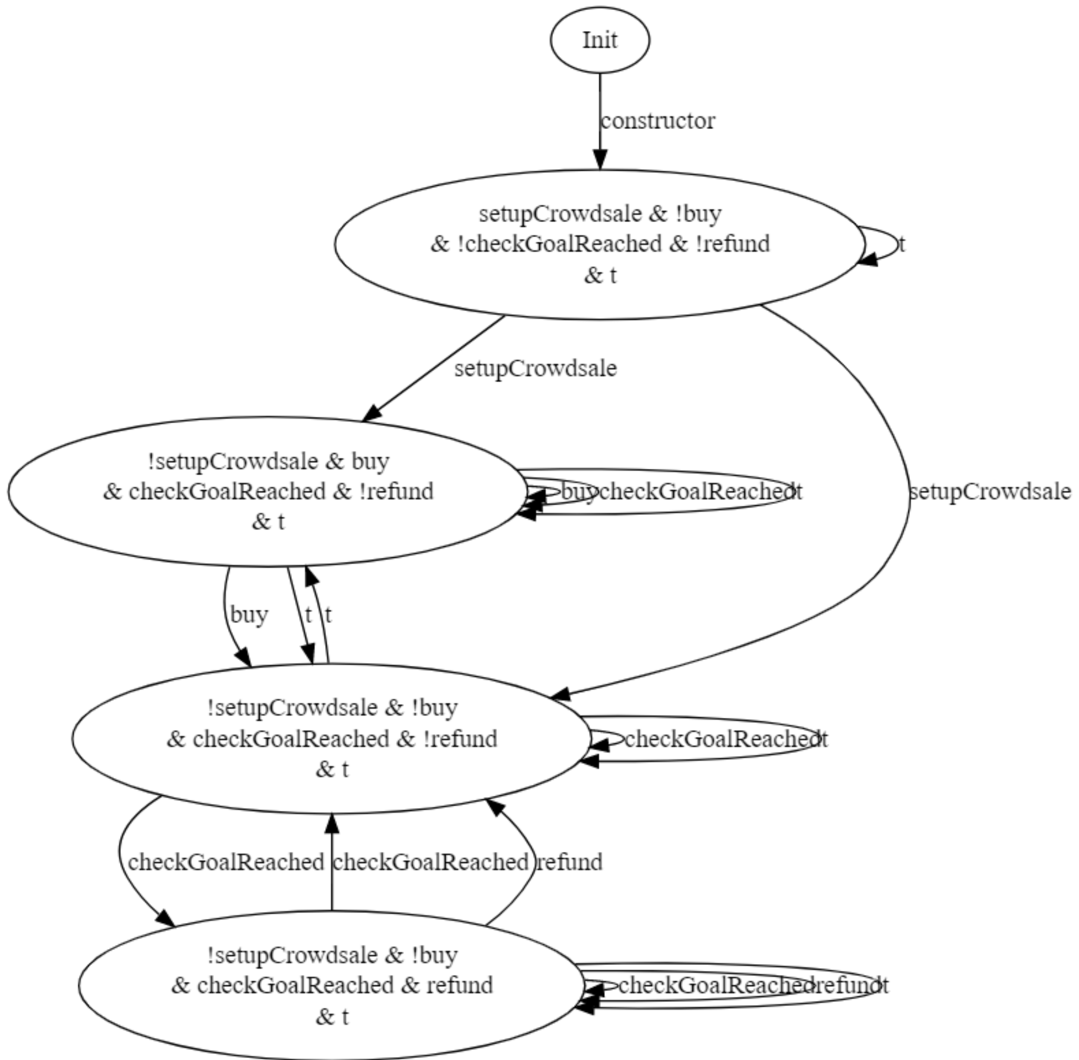

Fig. 20: EPXCrowdSale Abstracción EPA Alloy

y en la precondición de *buy* se establece que $fundingStartBlock < fundingEndBlock$. Por lo tanto, esta precondición se extiende al analizar el estado en el que se habilita refund. Por lo tanto, después de ejecutar $CheckGoalReach$, no existe camino para llegar al estado $CheckGoalReach$.

5. Experimentación

Hemos podido observar que los resultados obtenidos son coherentes con nuestras expectativas comparándolos con el estudio [1]. En el análisis del Benchmark de `Azure` obtuvo los mismos resultados que este trabajo. Algunos resultados del Benchmark de `Smartpulse` fueron diferentes. Los contratos de *Auction* y *EPXCrowdSale* tuvieron diferencias como se explicó en la sección anterior. Además se obtuvo diferencias en las interpretaciones de *Withdraw*, como se explicó en *Auction*, ya que en este estudio se priorizó que no haya un orden entre las acciones de *WithdrawA* y *WithdrawOther*. Estas fueron todas las diferencias encontradas.

Ahora, nos interesa evaluar el desempeño de nuestro programa para diferentes contratos. Deseamos comprender si existe una correlación entre los tiempos de ejecución y la cantidad de funciones que posee un contrato, o si existen otras variables relevantes que debemos tener en cuenta.

En la sección de desarrollo, se describieron varias optimizaciones que fueron incorporadas en el programa. Nos interesa evaluar el desempeño de estas optimizaciones, particularmente aquellas que reducen la cantidad de estados posibles a explorar.

Todas las pruebas de experimentación se corrieron en una computadora con procesador M1 de 8 núcleos con 8 threads de ejecución. Cada prueba se corrió 3 veces y se tuvo un promedio del tiempo.

Se procede a analizar los tiempos de ejecución necesarios para realizar las abstracciones del benchmark de *Azure*.

5.1. Benchmark Azure

Se va a realizar el análisis del tiempo que demora el programa del estudio en crear una abstracción para cada contrato. Se va analizar las versiones de los contratos originales y las versiones arregladas que se presentaron en la sección de Validación. A continuación se presentarán distintas tablas con diferentes análisis de datos. La columna $Config$ y $mode$ representa el nombre de la configuración del contrato que se ejecutó y el tipo de abstracción realizada. Las configuraciones que incluyen el nombre *Fixed* son las versiones arregladas de los contratos. *Initial* indica la cantidad de estados iniciales del contrato. La columna $S1$ muestra la cantidad de estados obtenidos después de aplicar las optimizaciones de reducción mencionadas en la sección de desarrollo. $S2$ es la cantidad de estados obtenidos después de validar los estados S1 en `VeriSol`, tal como se explicó en la sección de desarrollo. La columna FC indica la cantidad de funciones públicas que posee cada contrato.

Se procederá a analizar lo siguiente:

- El tiempo de ejecución en obtener una abstracción de las configuraciones con todas las optimizaciones por defecto.
- El tiempo de ejecución en obtener una abstracción de las configuraciones sin incluir una optimización en particular.
- Comparar los tiempos de ejecución de las configuraciones con y sin la optimización de estados.
- Comparar los tiempos de ejecución de las configuraciones con y sin la optimización de precondiciones.

Tiempos con optimizaciones

Tab. 1: Resumen de ejecución de Azure

Config	Mode	Time	Inital	S1	S2	FC
AssetTransfer	epa	0:18:52	1024	128	9	10
AssetTransfer	states	0:57:20	10	10	10	10
AssetTransferFixedConfig	states	0:53:19	10	10	10	10
BasicProvenance	states	0:00:47	3	3	3	2
BasicProvenance	epa	0:00:33	4	2	2	2
BasicProvenanceFixed	epa	0:00:55	4	4	3	2
BasicProvenanceFixedConfig	states	0:00:46	3	3	3	2
DefectiveComponentCounter	states	0:00:30	2	2	2	2
DefectiveComponentCounter	epa	0:00:17	4	1	1	2
DefectiveComponentCounterFixed	epa	0:00:33	4	2	2	2
DefectiveComponentCounterFixedConfig	states	0:00:29	2	2	2	2
DigitalLocker	states	0:11:19	6	6	6	10
DigitalLocker	epa	0:00:58	1024	1	1	10
DigitalLockerFixed	epa	0:05:52	1024	64	6	10
DigitalLockerFixedConfig	states	0:10:34	6	6	6	10
FrequentFlyerRewardsCalculator	epa	0:00:22	8	1	1	3
FrequentFlyerRewardsCalculator	states	0:00:41	2	2	2	3
HelloBlockchain	states	0:00:29	2	2	2	2
HelloBlockchain	epa	0:00:19	4	1	1	2
HelloBlockchainFixed	epa	0:00:31	4	4	2	2
HelloBlockchainFixedConfig	states	0:00:28	2	2	2	2
RefrigeratedTransportation	states	0:01:41	4	4	4	3
RefrigeratedTransportation	epa	0:00:46	8	4	2	3
RefrigeratedTransportationFixed	epa	0:01:13	8	4	3	3
RefrigeratedTransportationFixedConfig	states	0:01:39	4	4	4	3
RoomThermostat	epa	0:00:35	8	4	2	3
RoomThermostat	states	0:00:39	2	2	2	3
SimpleMarketplace	epa	0:00:53	8	4	3	3
SimpleMarketplace	states	0:01:02	3	3	3	3
SimpleMarketplaceFixed	epa	0:00:53	8	4	3	3
SimpleMarketplaceFixedConfig	states	0:01:01	3	3	3	3

La tabla 1 muestra el tiempo que demora ejecutar la abstracción de cada configuración con todas las optimizaciones por defecto. El programa final va a verificar S2 * FC * S2 para generar la abstracción.

Se plantea la hipótesis de que los programas con valores más altos de $S2$ y FC son los que tardan más tiempo en obtener una abstracción. Esta hipótesis se confirma al observar que los contratos que tardan más de 10 minutos en ejecutarse son aquellos que presentan valores altos de $S2$ y FC, y que no hay contratos con valores bajos de $S2$ y FC que demoren en ejecutarse. Además, se concluye que el tamaño de los estados iniciales no afecta el tiempo de ejecución. Por ejemplo, el contrato DigitalLocker, es el que más estados iniciales posee, pero al reducirse no afecta el tiempo final.

Tiempos sin optimizaciones

Ahora se desea analizar si las optimizaciones presentadas en la sección de desarrollo impactan positiva o negativamente en el tiempo necesario para obtener la abstracción. Se plantea la hipótesis de que estas optimizaciones ayudan a reducir el tiempo final del programa. Para ello, se ejecutará la misma abstracción eliminando una a una las optimizaciones presentadas en la sección 3.5.

Los diferentes modos de reducción que se ejecutaron son los siguientes:

- epa-noReduceTrue: Eliminación de la optimización True.
- epa-noReduceEqual: Eliminación de la optimización Equal.
- epa-noReduceTrueEqual: Eliminación de las optimizaciones de True y Equal.
- epa-noReduceAll: Eliminación de todas las optimizaciones anteriores.

La tabla 2 muestra un resumen de cuanto demora en ejecutar la abstracción de cada configuración sin incluir una optimización en particular. Se puede observar en la tabla que faltan algunos modos de reducción en los contratos, los cuales no se incluyeron en este trabajo debido a la complejidad de su análisis. Sin embargo, el objetivo de este estudio es investigar el funcionamiento de las optimizaciones descritas y esto se puede lograr observando los contratos más pequeños.

Al comparar en la tabla 2 los resultados obtenidos para los diferentes modos de reducción, se puede observar que las optimizaciones reducen el tiempo final del programa. En particular, al comparar el modo epa con el modo epa-noReduceAll, se puede notar una clara diferencia en el tiempo de ejecución. Por ejemplo, para el contrato $RefrigeratedTransportationFixed$, el modo epa tarda 1 minuto mientras que el modo sin reducciones tarda 4 minutos.

Por lo tanto, se puede concluir que las optimizaciones implementadas en este trabajo cumplen su objetivo.

Tiempos sin optimización de reducción de estados

A continuación, se analizará cada optimización por separado, empezando por la optimización reduce states, la cual se describe en la subsección de desarrollo 3.5. Para medir la eficacia de esta optimización, se comparará el tiempo de ejecución de un mismo contrato en el modo epa y en el modo a analizar.

La tabla 4 compara más en detalle la eficiencia de la optimización de reducción de estados. Se puede observar que en la mayoría de los contratos, el modo epa se ejecuta más rápido que el modo sin la optimización noReduceTrueEqual. Esto indica que en ciertos casos, es una buena optimización para agregar a la implementación.

Se puede notar que en los casos en los que esta optimización no es efectiva, es cuando el tamaño de los conjuntos S1 y S2 no se modifica. Esto tiene sentido ya que se están validando siempre estados correctos, por lo tanto, el tiempo de validación no sirve para reducir la búsqueda de transiciones.

Se puede concluir que esta optimización es de utilidad según el contrato a analizar. Si se sabe que al generar los estados, vamos a tener muchos inválidos, entonces tiene sentido agregarla.

Tab. 2: Resumen de ejecución de Azure con optimizaciones parte dos

Config	**Mode**	**Time**	**Inital**	**S1**	**S2**	**FC**
BasicProvenance	epa	0:00:44	4	2	2	2
BasicProvenance	epa-noReduceTrue	0:00:45	4	2	2	2
BasicProvenance	epa-noReduceEqual	0:00:49	4	4	2	2
BasicProvenance	epa-noReduceTrueEqual	0:00:48	4	4	2	2
BasicProvenance	epa-noReduceStates	0:00:36	4	2	2	2
BasicProvenance	epa-noReduceAll	0:01:30	4	4	4	2
BasicProvenanceFixed	epa	0:01:15	4	4	3	2
BasicProvenanceFixed	epa-noReduceTrue	0:01:17	4	4	3	2
BasicProvenanceFixed	epa-noReduceEqual	0:01:15	4	4	3	2
BasicProvenanceFixed	epa-noReduceTrueEqual	0:01:15	4	4	3	2
BasicProvenanceFixed	epa-noReduceStates	0:01:32	4	4	4	2
BasicProvenanceFixed	epa-noReduceAll	0:01:32	4	4	4	2
DefectiveComponentCounter	epa	0:00:23	4	1	1	2
DefectiveComponentCounter	epa-noReduceTrue	0:00:26	4	2	1	2
DefectiveComponentCounter	epa-noReduceEqual	0:00:23	4	1	1	2
DefectiveComponentCounter	epa-noReduceTrueEqual	0:00:30	4	4	1	2
DefectiveComponentCounter	epa-noReduceStates	0:00:18	4	1	1	2
DefectiveComponentCounter	epa-noReduceAll	0:01:23	4	4	4	2
DefectiveComponentCounterFixed	epa	0:00:49	4	2	2	2
DefectiveComponentCounterFixed	epa-noReduceTrue	0:00:53	4	4	2	2
DefectiveComponentCounterFixed	epa-noReduceEqual	0:00:49	4	2	2	2
DefectiveComponentCounterFixed	epa-noReduceTrueEqual	0:00:53	4	4	2	2
DefectiveComponentCounterFixed	epa-noReduceStates	0:00:40	4	2	2	2
DefectiveComponentCounterFixed	epa-noReduceAll	0:01:30	4	4	4	2
FrequentFlyerRewardsCalculator	epa	0:00:30	8	1	1	3
FrequentFlyerRewardsCalculator	epa-noReduceTrue	0:00:32	8	2	1	3
FrequentFlyerRewardsCalculator	epa-noReduceEqual	0:00:30	8	1	1	3
FrequentFlyerRewardsCalculator	epa-noReduceTrueEqual	0:00:39	8	8	1	3
FrequentFlyerRewardsCalculator	epa-noReduceStates	0:00:24	8	1	1	3
FrequentFlyerRewardsCalculator	epa-noReduceAll	0:05:13	8	8	8	3
HelloBlockchain	epa	0:00:26	4	1	1	2
HelloBlockchain	epa-noReduceTrue	0:00:28	4	2	1	2
HelloBlockchain	epa-noReduceEqual	0:00:26	4	1	1	2
HelloBlockchain	epa-noReduceTrueEqual	0:00:31	4	4	1	2
HelloBlockchain	epa-noReduceStates	0:00:21	4	1	1	2
HelloBlockchain	epa-noReduceAll	0:01:23	4	4	4	2
HelloBlockchainFixed	epa	0:00:41	4	4	2	2
HelloBlockchainFixed	epa-noReduceTrue	0:00:42	4	4	2	2
HelloBlockchainFixed	epa-noReduceEqual	0:00:41	4	4	2	2
HelloBlockchainFixed	epa-noReduceTrueEqual	0:00:41	4	4	2	2
HelloBlockchainFixed	epa-noReduceStates	0:01:30	4	4	4	2
HelloBlockchainFixed	epa-noReduceAll	0:01:31	4	4	4	2

Tab. 3: Resumen de ejecución de Azure con optimizaciones parte dos

Config	**Mode**	**Time**	**Inital**	**S1**	**S2**	**FC**
RefrigeratedTransportation	epa	0:01:05	8	4	2	3
RefrigeratedTransportation	epa-noReduceTrue	0:01:05	8	4	2	3
RefrigeratedTransportation	epa-noReduceEqual	0:01:09	8	8	2	3
RefrigeratedTransportation	epa-noReduceTrueEqual	0:01:09	8	8	2	3
RefrigeratedTransportation	epa-noReduceStates	0:02:17	8	4	4	3
RefrigeratedTransportation	epa-noReduceAll	0:06:14	8	8	8	3
RefrigeratedTransportationFixed	epa	0:01:50	8	4	3	3
RefrigeratedTransportationFixed	epa-noReduceTrue	0:01:50	8	4	3	3
RefrigeratedTransportationFixed	epa-noReduceEqual	0:01:55	8	8	3	3
RefrigeratedTransportationFixed	epa-noReduceTrueEqual	0:01:54	8	8	3	3
RefrigeratedTransportationFixed	epa-noReduceStates	0:02:20	8	4	4	3
RefrigeratedTransportationFixed	epa-noReduceAll	0:06:21	8	8	8	3
RoomThermostat	epa	0:00:53	8	4	2	3
RoomThermostat	epa-noReduceTrue	0:00:53	8	4	2	3
RoomThermostat	epa-noReduceEqual	0:00:58	8	8	2	3
RoomThermostat	epa-noReduceTrueEqual	0:00:57	8	8	2	3
RoomThermostat	epa-noReduceStates	0:02:04	8	4	4	3
RoomThermostat	epa-noReduceAll	0:05:37	8	8	8	3
SimpleMarketplace	epa	0:01:27	8	4	3	3
SimpleMarketplace	epa-noReduceTrue	0:01:31	8	8	3	3
SimpleMarketplace	epa-noReduceEqual	0:01:27	8	4	3	3
SimpleMarketplace	epa-noReduceTrueEqual	0:01:30	8	8	3	3
SimpleMarketplace	epa-noReduceStates	0:02:25	8	4	4	3
SimpleMarketplace	epa-noReduceAll	0:05:37	8	8	8	3
SimpleMarketplaceFixed	epa	0:01:15	8	4	3	3
SimpleMarketplaceFixed	epa-noReduceTrue	0:01:15	8	4	3	3
SimpleMarketplaceFixed	epa-noReduceEqual	0:01:18	8	8	3	3
SimpleMarketplaceFixed	epa-noReduceTrueEqual	0:01:18	8	8	3	3
SimpleMarketplaceFixed	epa-noReduceStates	0:02:06	8	4	4	3
SimpleMarketplaceFixed	epa-noReduceAll	0:05:38	8	8	8	3

Tab. 4: Tabla de tiempos Azure con optimización VeriSol

Config	**Mode**	**Time**	**Inital**	**S1**	**S2**	**FC**
BasicProvenance	epa	0:00:44	4	2	2	2
BasicProvenance	epa-noReduceStates	0:00:36	4	2	2	2
BasicProvenanceFixed	epa	0:01:15	4	4	3	2
BasicProvenanceFixed	epa-noReduceStates	0:01:32	4	4	4	2
DefectiveComponentCounter	epa	0:00:23	4	1	1	2
DefectiveComponentCounter	epa-noReduceStates	0:00:18	4	1	1	2
DefectiveComponentCounterFixed	epa	0:00:49	4	2	2	2
DefectiveComponentCounterFixed	epa-noReduceStates	0:00:40	4	2	2	2
FrequentFlyerRewardsCalculator	epa	0:00:30	8	1	1	3
FrequentFlyerRewardsCalculator	epa-noReduceStates	0:00:24	8	1	1	3
HelloBlockchain	epa	0:00:26	4	1	1	2
HelloBlockchain	epa-noReduceStates	0:00:21	4	1	1	2
HelloBlockchainFixed	epa	0:00:41	4	4	2	2
HelloBlockchainFixed	epa-noReduceStates	0:01:30	4	4	4	2
RefrigeratedTransportation	epa	0:01:05	8	4	2	3
RefrigeratedTransportation	epa-noReduceStates	0:02:17	8	4	4	3
RefrigeratedTransportationFixed	epa	0:01:50	8	4	3	3
RefrigeratedTransportationFixed	epa-noReduceStates	0:02:20	8	4	4	3
RoomThermostat	epa	0:00:53	8	4	2	3
RoomThermostat	epa-noReduceStates	0:02:04	8	4	4	3
SimpleMarketplace	epa	0:01:27	8	4	3	3
SimpleMarketplace	epa-noReduceStates	0:02:25	8	4	4	3
SimpleMarketplaceFixed	epa	0:01:15	8	4	3	3
SimpleMarketplaceFixed	epa-noReduceStates	0:02:06	8	4	4	3

Tab. 5: Optimización de reducción de precondiciones

Config	**Mode**	**Time**	**Inital**	**S1**	**S2**	**FC**
BasicProvenance	epa	0:00:44	4	2	2	2
BasicProvenance	epa-noReduceTrueEqual	0:00:48	4	4	2	2
BasicProvenanceFixed	epa	0:01:15	4	4	3	2
BasicProvenanceFixed	epa-noReduceTrueEqual	0:01:15	4	4	3	2
DefectiveComponentCounter	epa	0:00:23	4	1	1	2
DefectiveComponentCounter	epa-noReduceTrueEqual	0:00:30	4	4	1	2
DefectiveComponentCounterFixed	epa	0:00:49	4	2	2	2
DefectiveComponentCounterFixed	epa-noReduceTrueEqual	0:00:53	4	4	2	2
FrequentFlyerRewardsCalculator	epa	0:00:30	8	1	1	3
FrequentFlyerRewardsCalculator	epa-noReduceTrueEqual	0:00:39	8	8	1	3
HelloBlockchain	epa	0:00:26	4	1	1	2
HelloBlockchain	epa-noReduceTrueEqual	0:00:31	4	4	1	2
HelloBlockchainFixed	epa	0:00:41	4	4	2	2
HelloBlockchainFixed	epa-noReduceTrueEqual	0:00:41	4	4	2	2
RefrigeratedTransportation	epa	0:01:05	8	4	2	3
RefrigeratedTransportation	epa-noReduceTrueEqual	0:01:09	8	8	2	3
RefrigeratedTransportationFixed	epa	0:01:50	8	4	3	3
RefrigeratedTransportationFixed	epa-noReduceTrueEqual	0:01:54	8	8	3	3
RoomThermostat	epa	0:00:53	8	4	2	3
RoomThermostat	epa-noReduceTrueEqual	0:00:57	8	8	2	3
SimpleMarketplace	epa	0:01:27	8	4	3	3
SimpleMarketplace	epa-noReduceTrueEqual	0:01:30	8	8	3	3
SimpleMarketplaceFixed	epa	0:01:15	8	4	3	3
SimpleMarketplaceFixed	epa-noReduceTrueEqual	0:01:18	8	8	3	3

Tiempos sin optimización de reducción de precondiciones

Ahora se va a analizar la otra optimización, noReduceTrueEqual, la cual ha sido explicada en detalle en la subsección correspondiente 3.5. Se espera que esta optimización sea más efectiva en todos los casos, ya que no tiene un costo de operación tan alto como la anterior.

Para evaluar la eficacia de esta funcionalidad, se va a comparar el tiempo de un mismo contrato ejecutado en el modo epa con el tiempo de ejecución en el modo a analizar.

La tabla 5 compara más en detalle la eficiencia de la optimización precondiciones. Se observa que la hipótesis planteada es válida. En la mayoría de los casos, los contratos ejecutados sin la optimización de reduceTrueEqual tardan más tiempo en generar una abstracción. Los contratos en los que el tiempo de ejecución es prácticamente el mismo, son aquellos en los que la optimización no tiene ningún efecto. Por lo tanto, se puede concluir que no hay ningún efecto negativo al utilizar siempre esta optimización. Hay casos en los que va a ser muy útil y otros en los que no, pero no aumentará los tiempos de ejecución.

Además, se realizaron pruebas individuales con cada una de estas reducciones. Se puede observar que el impacto en cada contrato es diferente y depende de las precondiciones de los estados.

5.2. Benchmak Smartpulse

En el segundo benchmark, se busca evaluar la ejecución del programa con las optimizaciones implementadas y comparar los resultados obtenidos con los del benchmark de *Azure*.

Tiempos de ejecución con y sin optimizaciones

Se procederá a analizar en la tabla 6 los tiempos que se demora en crear una abstracción con todas las optimizaciones por defecto. Además se incluye lo que se demora el programa sin incluir una optimización en particular.

Al realizar un análisis detallado de la tabla presentada, se puede afirmar que se cumple la hipótesis planteada anteriormente, ya que en la gran mayoría de los casos, la implementación de las optimizaciones ha permitido que el programa del informe ejecute más rápido, tal como se ha demostrado en el benchmark de *Azure*.

Es importante mencionar que, debido a la naturaleza de algunos contratos, no se han llevado a cabo todas las ejecuciones posibles, ya que no tenían sentido en términos de tiempo y recursos. A pesar de ello, los resultados obtenidos permiten concluir que las optimizaciones presentadas tienen un efecto positivo en la velocidad de ejecución del programa.

Al comparar los contratos de *SmartPulse* con los de *Azure*, se puede observar que los primeros suelen tener más estados, lo que demuestra aún más el poder de las optimizaciones cuando estas tienen efecto. En este sentido, se puede destacar la reducción de estados, explicada en detalle en la sección correspondiente 3.5, la cual ha permitido reducir significativamente los tiempos de ejecución.

5.3. Conclusiones de las optimizaciones

Es importante destacar que, aunque en algunos casos la aplicación de esta optimización ha sumado un poco de tiempo, esto se debe a que en dichas ejecuciones la optimización no tenía efecto, tal como se mencionó anteriormente. Por lo tanto, se puede afirmar que, en general, la utilización de esta optimización no tiene efecto negativo en los tiempos de ejecución, y que su aplicación puede ser beneficiosa en la gran mayoría de los casos. Se puede observar concretamente el poder de las optimizaciones en la tabla 8.

Luego de analizar los resultados obtenidos en la experimentación, podemos concluir que los tiempos de ejecución para obtener una abstracción de un contrato dependen en gran medida de la cantidad de funciones públicas que tiene el mismo y de cómo son sus precondiciones. Es decir, cuanto más complejo sea el contrato y más posibles estados tenga, mayor será el tiempo de ejecución necesario para obtener la abstracción.

Las optimizaciones agregadas durante el proceso de abstracción de contratos demostraron ser útiles en la mayoría de los casos, lo cual se ve reflejado en la reducción de tiempos de ejecución. Sin embargo, es importante tener en cuenta que la optimización de reducción de estados mediante la herramienta `VeriSol` solo es útil cuando hay estados a reducir. En los casos donde la optimización no tiene efecto, se puede observar que el tiempo de ejecución aumenta ligeramente.

Tab. 6: Resumen de ejecución de Smartpulse

Config	Mode	Time	Inital	S1	S2	FC
AuctionConfig	epa	0:01:41	8	8	4	3
AuctionConfig	epa-noReduceTrue	0:01:37	8	8	4	3
AuctionConfig	epa-noReduceEqual	0:01:41	8	8	4	3
AuctionConfig	epa-noReduceTrueEqual	0:01:37	8	8	4	3
AuctionConfig	epa-noReduceStates	0:04:31	8	8	8	3
AuctionConfig	epa-noReduceAll	0:04:29	8	8	8	3
AuctionWithdrawConfig	epa	0:02:46	16	16	6	4
AuctionWithdrawConfig	epa-noReduceTrue	0:02:47	16	16	6	4
AuctionWithdrawConfig	epa-noReduceEqual	0:02:49	16	16	6	4
AuctionWithdrawConfig	epa-noReduceTrueEqual	0:02:44	16	16	6	4
AuctionWithdrawConfig	epa-noReduceStates	0:30:42	16	16	16	4
AuctionWithdrawConfig	epa-noReduceAll	0:42:13	16	16	16	4
AuctionEndedConfig	states	0:10:05	8	8	8	3
CrowdfundingConfig	epa	0:00:59	8	8	4	3
CrowdfundingConfig	epa-noReduceTrue	0:00:59	8	8	4	3
CrowdfundingConfig	epa-noReduceEqual	0:01:00	8	8	4	3
CrowdfundingConfig	epa-noReduceTrueEqual	0:00:59	8	8	4	3
CrowdfundingConfig	epa-noReduceStates	0:04:09	8	8	8	3
CrowdfundingConfig	epa-noReduceAll	0:04:09	8	8	8	3
CrowdfundingTimeConfig	epa	0:01:34	16	8	4	4
CrowdfundingTimeConfig	epa-noReduceTrue	0:01:45	16	16	4	4
CrowdfundingTimeConfig	epa-noReduceEqual	0:01:33	16	8	4	4
CrowdfundingTimeConfig	epa-noReduceTrueEqual	0:01:45	16	16	4	4
CrowdfundingTimeConfig	epa-noReduceStates	0:06:05	16	8	8	4
CrowdfundingTimeConfig	epa-noReduceAll	0:35:24	16	16	16	4
CrowdfundingBalanceConfig	states	0:03:53	5	5	5	3
EscrowVaultConfig	epa	0:03:01	256	16	4	8
EscrowVaultConfig	epa-noReduceTrue	0:03:22	256	32	4	8
EscrowVaultConfig	epa-noReduceEqual	0:05:51	256	128	4	8
EscrowVaultConfig	epa-noReduceTrueEqual	0:09:33	256	256	4	8
EscrowVaultConfig	epa-noReduceStates	1:36:05	256	16	16	8
EscrowVaultConfig	states	0:19:33	4	4	4	8
RefundEscrowConfig	epa	0:01:50	64	4	3	6
RefundEscrowConfig	epa-noReduceTrue	0:01:57	64	8	3	6
RefundEscrowConfig	epa-noReduceEqual	0:02:07	64	16	3	6
RefundEscrowConfig	epa-noReduceTrueEqual	0:03:19	64	64	3	6
RefundEscrowConfig	epa-noReduceStates	0:03:00	64	4	4	6
RefundEscrowConfig	states	0:05:29	3	3	3	6
RefundEscrowConfig	epa-noReduceAll	16:23:48	64	64	64	6
RefundEscrowWithdrawConfig	epa	0:03:22	64	16	6	6
RefundEscrowWithdrawConfig	epa-noReduceTrue	0:03:23	64	16	6	6
RefundEscrowWithdrawConfig	epa-noReduceEqual	0:04:32	64	64	6	6
RefundEscrowWithdrawConfig	epa-noReduceTrueEqual	0:04:35	64	64	6	6
RefundEscrowWithdrawConfig	epa-noReduceStates	1:06:29	64	16	16	6

Tab. 7: Resumen de ejecución de Smartpulse parte 2

Config	Mode	Time	Inital	S1	S2	FC
RockPaperScissorsConfig	epa	0:01:17	8	8	4	3
RockPaperScissorsConfig	epa-noReduceTrue	0:01:19	8	8	4	3
RockPaperScissorsConfig	epa-noReduceEqual	0:01:16	8	8	4	3
RockPaperScissorsConfig	epa-noReduceTrueEqual	0:01:16	8	8	4	3
RockPaperScissorsConfig	epa-noReduceStates	0:03:48	8	8	8	3
RockPaperScissorsConfig	states	0:07:36	6	6	6	3
SimpleAuctionConfig	epa	0:04:40	8	8	8	3
SimpleAuctionConfig	epa-noReduceTrue	0:04:38	8	8	8	3
SimpleAuctionConfig	epa-noReduceEqual	0:04:41	8	8	8	3
SimpleAuctionConfig	epa-noReduceTrueEqual	0:04:39	8	8	8	3
SimpleAuctionConfig	epa-noReduceStates	0:04:26	8	8	8	3
SimpleAuctionConfig	states	0:15:17	6	6	6	3
SimpleAuctionTimeConfig	epa	0:09:55	16	8	8	4
SimpleAuctionTimeConfig	epa-noReduceTrue	0:10:08	16	16	8	4
SimpleAuctionTimeConfig	epa-noReduceEqual	0:10:13	16	8	8	4
SimpleAuctionTimeConfig	epa-noReduceTrueEqual	0:10:21	16	16	8	4
SimpleAuctionTimeConfig	epa-noReduceStates	0:09:38	16	8	8	4
SimpleAuctionTimeConfig	epa-noReduceAll	0:42:23	16	16	16	4

Tab. 8: Resumen de mejoras en tiempo con optimizaciones

Config	Mode	Time	Inital	S1	S2	FC
EscrowVaultConfig	epa	0:03:01	256	16	4	8
EscrowVaultConfig	epa-noReduceTrueEqual	0:09:33	256	256	4	8
EscrowVaultConfig	epa-noReduceStates	1:36:05	256	16	16	8
RefundEscrowConfig	epa	0:01:50	64	4	3	6
RefundEscrowConfig	epa-noReduceTrueEqual	0:03:19	64	64	3	6
RefundEscrowConfig	epa-noReduceStates	0:03:00	64	4	4	6
RefundEscrowConfig	epa-noReduceAll	16:23:48	64	64	64	6
RefundEscrowWithdrawConfig	epa	0:03:22	64	16	6	6
RefundEscrowWithdrawConfig	epa-noReduceTrueEqual	0:04:35	64	64	6	6
RefundEscrowWithdrawConfig	epa-noReduceStates	1:06:29	64	16	16	6

6. Trabajo Futuro

Existen diversas oportunidades para extender y mejorar este trabajo en el futuro:

- Mejoras de las limitaciones de $VeriSol$: Se puede realizar contribuciones al desarrollo de $VeriSol$ o crear un script que permita modificar automáticamente las limitaciones de los contratos identificadas en este trabajo. Esto permitiría adaptar $VeriSol$ de manera más eficiente a diferentes tipos de contratos y escenarios.
- Pruebas de las abstracciones generadas: Sería útil realizar pruebas automatizadas sobre las abstracciones generadas y compararlas con conjuntos de referencia. Esto permitiría evaluar la calidad y cobertura de las abstracciones y proporcionaría una base para medir el rendimiento de las optimizaciones implementadas.
- Separación del archivo de configuración de $VeriSol$: Sería conveniente tener un archivo de configuración separado para $VeriSol$, distinto del programa desarrollado en este trabajo. Esto facilitaría la sustitución de esta herramienta en el futuro y permitiría una configuración más flexible y personalizada para diferentes contratos.
- Mejora del analizador de contratos: Sería beneficioso desarrollar un analizador de contratos más sofisticado que pueda extraer automáticamente las funciones públicas y sus precondiciones. Esto eliminaría la necesidad de escribir manualmente esta información en el archivo de configuración, ahorrando tiempo y reduciendo posibles errores.

7. Conclusiones

Los contratos inteligentes son una forma cada vez más popular de llevar a cabo acuerdos entre partes, ya que pueden automatizar procesos y garantizar la transparencia y la confianza en la ejecución del contrato. Sin embargo, estos contratos pueden contener errores que hacen que su validación y verificación sean complicadas. Por lo tanto, es necesario contar con herramientas que permitan verificar la correctitud de los contratos inteligentes y faciliten la comprensión de su comportamiento.

Una de las formas de facilitar la comprensión del comportamiento de los contratos inteligentes es construir abstracciones de máquinas de estado, que muestran los diferentes estados y las transiciones entre ellos. Sin embargo, construir estas abstracciones manualmente puede ser difícil y requerir mucho tiempo, lo que limita su utilidad.

El objetivo del plan propuesto es desarrollar una herramienta que pueda construir automáticamente y eficientemente abstracciones de máquinas de estado para contratos inteligentes utilizando un analizador estático de programas Solidity.

Para validar la eficacia de la herramienta desarrollada, se utilizó como medida de validación la comparación de los resultados generados con los benchmarks de Azure y Smartpulse con el trabajo [1]. El primero contiene implementaciones de contratos inteligentes con una descripción detallada de su comportamiento previsto y un diagrama de transición de estado producido manualmente que describe el comportamiento esperado del contrato. Los resultados obtenidos permiten afirmar que el programa desarrollado cumple con su objetivo principal de generar abstracciones precisas y confiables de estados y transiciones de contratos inteligentes.

En conclusión, la herramienta desarrollada es precisa en la abstracción de estados y transiciones de contratos inteligentes. La validación realizada mediante la comparación con el trabajo [1] respalda la calidad de los resultados generados y confirma la utilidad y efectividad del programa en la verificación y validación de contratos inteligentes. La automatización de la construcción de abstracciones de máquinas de estado para contratos inteligentes representa un gran avance en la industria y puede mejorar significativamente la seguridad y eficiencia de los acuerdos llevados a cabo mediante contratos inteligentes.

Bibliografía

[1] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, Sebastián Uchitel: Predicate abstractions for smart contract validation. MoDELS 2022: 289-299

[2] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. Commun. ACM 62, 9 (September 2019), 66–76.

[3] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, Sebastián Uchitel: Enabledness-based program abstractions for behavior validation. ACM Trans. Softw. Eng. Methodol. 22(3): 25:1-25:46 (2013)

[4] Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer Formal Specification and Verification of Smart Contracts for Azure Blockchain (April 2019)

[5] Azure benchmark [Online] https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench

[6] Smartpulse benchmark [Online] https://github.com/utopia-group/SmartPulseTool/tree/master/smartpulseBenchmarks/livenessBenchmarks

[7] Dr. Gavin Wood Ethereum: a secure decentralised generalised transaction ledger https://ethereum.github.io/yellowpaper/paper.pdf (October 2022)

[8] Phillip Goldberg Smart Contract Best Practices Revisited: Block Number vs. Timestamp https://medium.com/@phillipgoldberg/smart-contract-best-practices-revisited-block-number-vs-timestamp-648905104323 (2018)

[9] K. Rustan M. Leino This is Boogie 2 (2008)

[10] Akash Lal, Shaz Qadeer, Shuvendu Lahiri Corral: A Solver for Reachability Modulo Theories (2012)

[11] Repositorio del trabajo https://github.com/edentorres/tesis