Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Construcción de Abstracciones de comportamiento para contratos inteligentes mediante ejecución simbólica

Tesis de Licenciatura en Ciencias de la Computación

Daniel Wappner

Director: Javier Godoy

Codirector: Diego David Garbervetsky

Buenos Aires, 2024

CONSTRUCCIÓN DE ABSTRACCIONES DE COMPORTAMIENTO PARA CONTRATOS INTELIGENTES MEDIANTE EJECUCIÓN SIMBÓLICA

Los smart contracts son programas inmutables que se despliegan en una blockchain. Dado que a menudo manejan activos de alto valor real, su verificación y validación antes de desplegarlos es de gran importancia. Por esta razón, es una práctica común contratar empresas de seguridad especializadas para auditar el código de los smart contracts. Sin embargo, se han explotado numerosas vulnerabilidades en los últimos años provocando pérdidas a miles de personas.

Las *Enabledness Preserving Abstractions* (EPAs), son máquinas de estado finitas que abstraen el comportamiento de artefactos de código, basándose en predicados sobre la habilitación de los métodos disponibles. En general, han resultado útiles como herramienta para la validación de código tanto contra especificaciones formales como contra modelos informales o “mentales” del comportamiento esperado.

Presentamos un protitpo que genera EPAs de contratos inteligentes a partir de código fuente, haciendo uso y extensión de una herramienta open source de ejecución simbólica dinámica: “Manticore”. Discutimos las optimizaciones implementadas y comparamos el prototipo desarrollado con otras estrategias alternativas.

Palabras claves: Contratos inteligentes, Ejecución simbólica, Construcción de abstracciones, Validación, Modelado, Solidity, Análisis estático.

AGRADECIMIENTOS

Antes que nada, quería agradecer a los jurados de esta tesis: Sebastián Uchitel y Ariel Waisbein por tomarse el tiempo de leer y evaluar este trabajo.

Quería agradecer al LaFHIS en su conjunto, no solo por abrirme las puertas para realizar este trabajo y otras oportunidades que me ofrecieron, sino también por brindar un ambiente tan lindo para realizar este trabajo.

Me gustaría agradecer también a todo el apoyo que recibí para poder llegar hasta acá en la carrera. A les docentes que año tras año no sólo dictan las clases cada vez más masivas sino que además preparan material, exámenes y las materias; y administratives que que resuleven tantos problemas en horas extra para que la gente pueda estudiar en la universidad públca. A todas las personas involucradas en mi camino personal que recorrí por la facultad, y a mis amigos y compañeros de cursada en particular: Erik, Ale, Abrol, Laureano, Charles, Dani, Blas, Ivo, Franco, Luciana, Sujo, Luciano, Juli, Fran, Yuri y Javi.

A mis amigues de cuando no estoy en la facultad, gracias por estar dispuestos a hacer tantas cosas distintas juntos, y en esta vuelta a bancarme con que no pueda ir a ningún lugar nunca. Los aprecio muchísimo Lucio, Eliseo, Iván, Nico, Lucas, Feli, Marto, Aylen, Michu, Bruno.

Javi y Diego, muchísimas gracias. Me ayudaron muchísimo al dirigirme la tesis, fueron siempre personas increíbles, y realmente me hicieron sentir incluido un montón. Realmente siempre están disponibles a cualquier cosa y cada duda, problema o detalle que puedas tener.

Quiero además agradecer a mi familia por acompañarme hasta acá. A quienes me acompañaron desde chiquito, abuelos Sasha y Angel; abuelas Didi y María; y Marta. A todos mis primos y tíos que me encanta ver. A los más cercanos, gracias Papá, Mamá, Marcos, Meli y Cami.

Índice general

1.. Introduccion 1

1.1. Objetivos 2

2.. Motivación 3

3.. Conceptos Preliminares 6

3.1. Ejecución simbólica dinámica 6

3.2. Blockchains 8

3.2.1. Contratos Inteligentes en Ethereum y Solidity 8

3.2.2. Ejemplo 9

3.3. Enabledness-Preserving Abstractions 11

3.4. Modelo Formal 12

4.. Manticore 15

4.1. Herramienta por Línea de Comandos 15

4.2. Manticore-Verifier 16

4.3. Arquitectura 16

4.4. API programable 17

5.. Construcción de EPAs mediante ejecución simbólica 19

5.1. Construcción clásica de EPAs 19

5.2. Algoritmo alternativo 21

5.3. Limitaciones 25

6.. Implementación 28

6.1. Algoritmo clásico 28

6.2. Algoritmo alternativo 29

6.3. Manticore como caja negra 30

6.4. Abstracción de los contratos por combinación de enums 31

7.. Análisis 34

7.1. Comparación de las abstracciones generadas entre el algoritmo clásico y el alternativo 34

7.1.1. Caso `RoomThermostat` 35

7.1.2. Caso `BoundedStack` 35

7.2. Comparación en tiempo de ejecución entre el algoritmo clásico y el alternativo 37

7.3. Evaluación más detallada del tiempo de ejecución del algoritmo alternativo 38

8.. Conclusiones 43

8.1. Trabajo relacionado 44

1. INTRODUCCION

Las redes de blockchain siguen un protocolo para manejar un registro distribuido “confiable” de los hechos. La participación en estas redes es por diseño anónima y descentralizada, y hace gran hincapié en la resistencia del sistema a ataques criptográficos. Una vez registrada, resulta casi imposible alterar información introducida en una blockchain. A fines prácticos, una transacción en una de estas bases de datos suele ser considerada eterna e inmutable. Adicionalmente, la cadena de bloques propiamente dicha, es decir, el historial de la información registrada, es transparente y público [49] [30].

Desde el origen de Bitcoin en 2009, estas tecnologías se utilizaron para construir un “libro contable” distribuido de transacciones financieras [30]. Más adelante, Ethereum en 2014 [9] es pionera y se torna desde entonces en la más popular de las redes que soportan la ejecución de software en ellas. A los programas que se ejecutan en las blockchain se los conoce como contratos inteligentes o *smart contracts* y la capacidad de computarlos se integra íntimiamente al diseño de las blockchain. De hecho, los protocolos de consenso de las redes que soportan smart contracts quedan firmemente atados al modelo de cómputo elegido. Como ejemplo, la red Ethereum provee la *Ethereum Virtual Machine*[47], una máquina de pila de profundidad finita, y para su programación desarrolló Solidity[28], un lenguaje de programación compilado curly-brace. Otra blockchain popular que soporta smart contracts es Algorand mediante la *Algorand Virtual Machine* [1], que es también una máquina de pila y TEAL, el lenguaje de programación tipo assembly correspondiente.

Una propiedad de las blockchains como sistema de cómputo distribuido es que el modo en el que los contratos inteligentes son incluidos en la blockchain garantiza la seguridad de su ejecución, en el sentido de que puede asegurarse que el resultado obtenido será el mismo que el de un código fuente que los usuarios conocen [30]. La inmutabilidad de las transacciones registradas asegura que los contratos no pueden ser modificados, lo que además garantiza a los usuarios que el comportamiento de los contratos se mantendrá siempre estable. Por otro lado, dado que los cambios en el estado del programa son registrados en la blockchain, estos también se consideran irreversibles. Esto, a pesar de ciertas garantías de estabilidad que les otorga a los usuarios, significa que los defectos en la implementación de los contratos inteligentes no pueden ser reparados, y las transacciones no deseadas originadas de estos defectos no pueden ser revertidas. Si se quieren evitar defectos, la verificación y validación de los contratos antes de publicarlos en la red es de suma importancia. De hecho, los ataques a contratos inteligentes para abusar *bugs* han causado históricamente pérdidas materiales a miles de personas [8].

Por esto, es común en la industria contratar a auditores independientes para la validación de contratos inteligentes, quienes a menudo usan herramientas para facilitar su trabajo y a menudo cuentan sólo con especificaciones informales del comportamiento deseado. En estas situaciones, una de las tantas técnicas útiles para la validación es la construcción de una máquina de estados finita que abstraiga el comportamiento del contrato, destacando sólo ciertas propiedades interesantes [20].

En 2013, De Caso et. al [11] propone las EPAs como abstracciones para implementaciones de APIs especificadas en formato pre- y post-condición. En ese trabajo exploran la utilidad de las EPAs para la asistencia de validación de código y argumentan su usabilidad para bug finding, mejora de especificaciones formales y de documentación. Además, presentan una técnica para construir EPAs en general e implementan una herramienta que las sintetiza automáticamente a partir de código fuente de estructuras en C.

En 2022, Godoy et. al [20] propone el uso de abstracciones similares a las EPAs en el contexto de validación de contratos inteligentes, estableciendo su aplicabilidad a estos debido a la costumbre de programar precondiciones directamente en los contratos. Proponen una estrategia de validación guiada por expertos más general, en el que los auditores refinan las abstracciones generadas sucesivamente. Sugieren partir de abstracciones idénticas a las EPAs y/o basadas en el valor de variables de tipo enum, agregando predicados ideados por los auditores para discernir propiedades que les resulten interesantes.

En 2023, Torres et. al [46] propone el uso de VeriSol [25] para la generación automática de estas abstracciones por predicados de contratos inteligentes. Presentan una herramienta que analiza automáticamente los contratos, reduciendo su tiempo de ejecución a unos pocos minutos para un benchmark conocido. VeriSol, la herramienta utilizada, es una herramienta de bounded model checking para contratos escritos en versiones de Solidity anteriores a `0.6`. Realiza una traducción de los contratos a un lenguaje de verificación intermedio, Boogie [27], perdiendo en el proceso la capacidad de modelar interacciones de los contratos con el resto de la blockchain.

1.1. Objetivos

Nos interesa explorar otras alternativas para la construcción de EPAs para contratos inteligentes implementados en Solidity de manera automática. Buscamos implementar un prototipo que se base en ejecución simbólica utilizando Manticore, una herramienta open source de ejecución simbólica dinámica con soporte tanto para la EVM como código nativo desarrollada por trailofbits [5]. Esperamos generar abstracciones correctas y de manera eficiente. Además, Manticore respalda la simulación de una blockchain completa, manteniendo el estado de múltiples contratos y usuarios dentro de la red. Buscamos aprovechar estas cualidades para detectar comportamientos más complejos en las abstracciones generadas.

2. MOTIVACIÓN

El objetivo de esta sección es presentar un escenario sencillo para mostrar cómo el uso de EPAs puede ayudar a un auditor a ganar conocimiento sobre contratos inteligentes que no escribió, y de los que no cuenta ninguna especificación formal. El ejemplo es extraído del artículo publicado por Godoy et al. en 2022[20], y se refiere a una implementación de `Crowdfunding` perteneciente a la librería de contratos de OpenZeppelin durante 2022 [36]. Las figuras utilizadas son provenientes del trabajo publicado por Torres et. al en 2023[46].

Supongamos que un auditor ofrece servicios de auditoría de contratos inteligentes como contratista, y accede a evaluar un contrato de un cliente. El cliente quiere desplegar un contrato en una blockchain que se encargue de recaudar el dinero que necesitará para su próximo proyecto. Cuenta con una implementación de este contrato, pero la comunidad de inversores exige que el contrato sea validado por un experto para corroborar que hace lo prometido con el dinero. Por este motivo, el cliente provee el código fuente al auditor y una descripción burda del comportamiento deseado:

- Al desplegar el contrato, se debe anunciar el objetivo deseado (en dinero a recaudar) y la fecha límite hasta la que estará abierta la recaudación.
- La comunidad de inversores puede contribuir hacia la meta con cuanto dinero desee, siempre y cuando no se haya alcanzado la fecha límite.
- De alcanzarse la meta a tiempo, entonces el cliente podrá recolectar todo el dinero recaudado.
- Si la recaudación llega a su fecha límite sin haber alcanzado la meta, entonces el cliente pretende garantizar a los inversores que podrán recuperar todo su dinero.
- Sin embargo, antes de la fecha límite ni los inversores ni el cliente pueden retirar dinero, para garantizar transparencia de cuál es el porcentaje de avance hacia la meta.

El auditor comienza su trabajo sobre la implementación del contrato `Crowdfunding` (cuyo código podemos ver en el fragmento 2.1). Puede observar que aparte del constructor inicial, está programado con tres métodos: `Donate`, `GetFunds` y `Claim`. Debido a que estos no son muy complejos puede rápidamente ver que están bien nombrados; cada uno se corresponde con una de las acciones descritas por el cliente que se desea que se puedan hacer en la recaudación. Sin embargo, es crucial corroborar que estos métodos estén habilitados de la manera que el cliente desea. Para realizar este análisis el auditor utiliza una herramienta automática que abstrae el contrato a una máquina de estados finita, que podemos ver en la figura 2.1

Esta máquina de estados se corresponde razonablemente bien con las etapas en la recaudación que describe el cliente. Por ejemplo, podemos ver que desde el estado en el

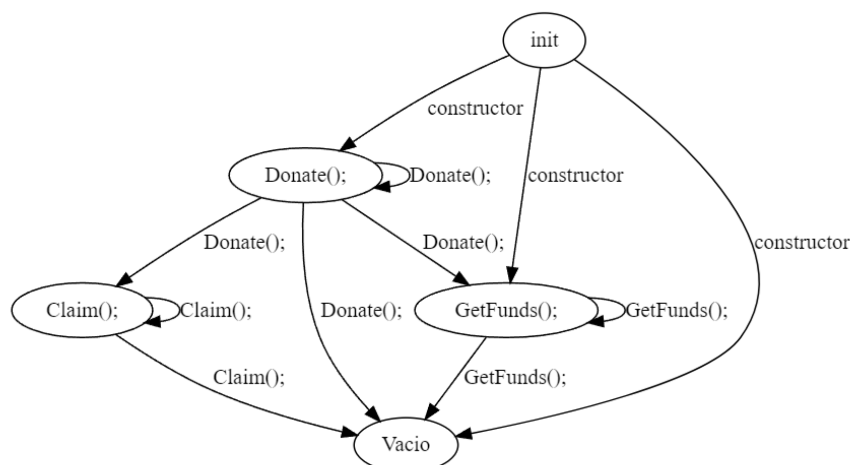


Fig. 2.1: Enabledness Preserving Abstraction del contrato `Crowdfunding`

que se puede donar (el etiquetado como `Donate`) es posible donar múltiples veces. Además, es posible donar y transicionar a un estado en el que lo unico que se puede hacer es que el cliente reclame los fondos (el etiquetado como `Claim`) o a un estado donde lo unico que se puede hacer es que los inversores recuperen su dinero (el etiquetado como `GetFunds`).

De esta manera, el auditor puede observar todos los caminos que son posibles en la abstracción y contrastarlos con los caminos que piensa que deberían ser posibles dada la descripción del cliente. Aún más, al analizar la abstracción puede buscar caminos que le resulten sospechosos; que parezcan indicar que es posible una secuencia de ejecuciones que no resulta deseable en base a la especificación con la que cuenta. Por ejemplo, al auditor podrían llamarle la atención las transiciones que llevan al estado etiquetado como “`Vacío`”, ya que no desea que el contrato pueda bloquearse. Sospechando que la causa de este estado “`Vacío`” en realidad son detalles en la implementación con respecto al manejo del tiempo (recordemos que el auditor es un experto), le pide a la herramienta que genere una nueva abstracción en que modele el paso del tiempo, como la que podemos ver en la figura 2.2

Al observar esta nueva máquina de estados, el auditor puede confirmar su sospecha de que el estado “`Vacío`” en realidad era un estado al que se puede entrar y salir con el paso del tiempo. Sin embargo su trabajo no está terminado; debe poner a prueba todos los escenarios que se le ocurran hasta garantizar que el dinero de todos los participantes esté a salvo. Por ejemplo, podría continuar refinando la abstracción pidiéndole a la herramienta que divida los estados del contrato en base a si el contrato controla dinero o no, ya que el único caso problemático sería que el contrato se bloquee cuando controla dinero de las otras personas.

A grandes rasgos, podemos decir que este procedimiento de generar abstracciones y refinarlas permite al auditor validar el comportamiento del contrato. Hace esto enfocándose en casos de uso o ejecución complejos, pero puede hacerlo sin depender de análisis manuales intensivos del código fuente, que son propensos a errores.

```
1 contract Crowdfunding {
2     address payable owner;
3     uint max_block;
```

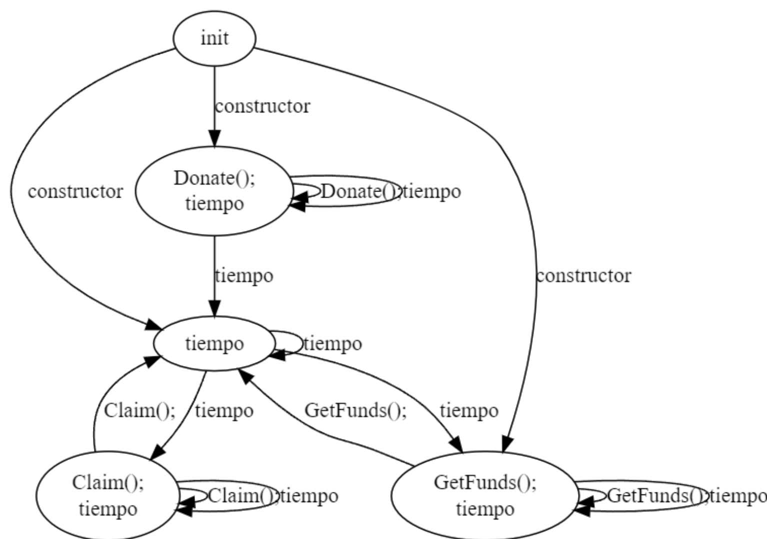


Fig. 2.2: Abstracción EPA del contrato `Crowdfunding`, con modelado del paso del tiempo.

```
4      uint goal;
5      mapping (address=>uint) backers;
6      bool funded = false;
7
8      constructor (address payable _owner, uint _max_block, int _goal) public {
9          owner = _owner;
10         max_block = _max_block;
11         goal = _goal;
12     }
13     function Donate () public payable {
14         require (max_block > block.number);
15         require (backers [msg.sender] == 0);
16         backers [msg.sender] = msg.value;
17     }
18     function GetFunds () public {
19         require (max_block < block.number);
20         require (msg.sender == owner);
21         require (goal <= address(this).balance);
22         funded = true;
23         owner.transfer(address(this).balance);
24     }
25     function Claim () public {
26         require (max_block < block.number);
27         require (backers [msg.sender] > 0 && ! funded);
28         require (goal > address (this).balance);
29         uint val = backers [msg.sender];
30         backers [msg.sender] = 0;
31         msg.sender.transfer (val);
32     }
33 }
```

Listing 2.1: Implementación de la recaudación

3. CONCEPTOS PRELIMINARES

3.1. Ejecución simbólica dinámica

En ejecuciones concretas, un programa toma valores de entrada fijos y realiza un único camino en el árbol de control de flujo. Una única ejecución simbólica, en cambio, explora múltiples caminos que el programa podría tomar bajo valores de entrada diversos [2] [10]. El concepto clave detrás de estas ejecuciones es permitir que los programas sean ejecutados con valores “simbólicos” como entrada. A lo largo de una dada ejecución, se mantiene para cada camino explorado en el programa una fórmula que describe las condiciones requeridas para tomar ese camino y una memoria que asigna expresiones o valores a las variables simbólicas.

```
1  void aFunction ( int a , int b ) {
2      int x = 1 , y = 0;
3
4      if( a != 0) {
5          y = 3 + x ;
6
7          if( b == 0) {
8              x = 2 * ( a + b ) ;
9          }
10     }
11
12     assert ( x - y != 0) ;
13 }
```

Listing 3.1: Ejemplo de código para ilustrar los árboles de ejecución simbólica

En el contexto de pruebas automáticas de software, la ejecución simbólica permite explorar tantos caminos como sea posible en el tiempo que se le otorga y luego, para cada camino explorado, generar valores concretos que lo recorren. Esto permite para un gran abanico de caminos obtener valores que fuerzen su ejecución concreta. Si lo que se busca es generar casos de prueba, la ejecución simbólica permite obtener una alta cobertura de caminos en el programa analizado. A la hora de econtrar defectos, los valores concretos ofrecidos por ejecución simbólica permiten corroborar y depurar la presencia de bugs al ejecutarlos concretamente, en un contexto distinto a la ejecución simbólica que los generó. En el fragmento de código 3.1 y en la figura 3.1 podemos ver una función sencilla y su árbol de ejecución simbólica asociado.

Dada una condición de camino, que es la fórmula booleana mencionada, la generación de valores de entrada concretos que la satisfagan se basa típicamente en un SMT (Satisfibality Modulo Theory) solver. La resolución de condiciones es uno de los principales desafíos que enfrenta la ejecución simbólica. Los SMT solvers escalan y permiten resolver combinaciones complejas de condiciones sobre cientos de variables. Sin embargo, ciertos elementos presentes en código comúnmente analizado, como la aritmética no lineal suelen

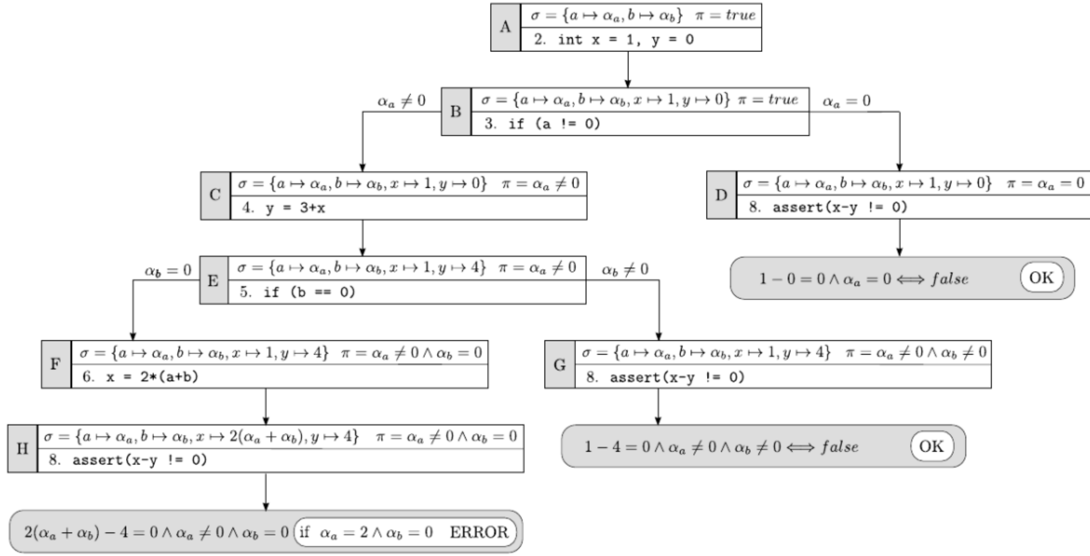


Fig. 3.1: Ejemplo del árbol de ejecución generado por la función `aFunction`. Los estados de ejecución, indicados con una letra mayúscula, indican la sentencia a ser ejecutada, la memoria simbólica σ y las condiciones de camino π. Las hojas son evaluadas contra la condición del `assert` correspondiente. Este ejemplo fue tomado de [6] y [2]

ser un obstáculo en la eficiencia.

La ejecución simbólica clásica no puede lidiar con la resolución de condiciones demasiado complejas. De hecho, algunas condiciones pueden resultar indecidibles o no tener una estrategia computacionalmente razonable de resolución, como las que requieren inversión de funciones de hash [5]. Un enfoque popular para resolver este problema, conocido como ejecución simbólica dinámica, es hacer que la ejecución simbólica sea dirigida por una ejecución concreta. Comenzando por valores concretos arbitrarios, se ejecuta el programa tanto concretamente como simbólicamente, generando las condiciones de camino correspondientes al camino realizado por la ejecución concreta. Con esta técnica el motor simbólico no necesita invocar al solucionador de restricciones para decidir si una condición de rama es satisfacible. En cambio, para explorar diferentes caminos, se busca generar valores concretos nuevos que **no** cumplan las condiciones de camino del último camino generado.

Existen múltiples herramientas enfocadas en la detección de errores de contratos inteligentes vía ejecución simbólica. Oyente [31][6] fue una de las primeras estrategias publicadas, y se enfoca en vulnerabilidades de contratos inteligentes conocidas. Pakala [37][6] analiza interacciones de múltiples transacciones de manera eficiente, buscando vulnerabilidades relacionadas a las criptomonedas. Mythril [35][6] es una herramienta open source que utiliza ejecución simbólica para analizar propiedades de seguridad predefinidas. teEther [24][6] realiza ejecución simbólica sólo en una porción de los programas analizados, para alcanzar caminos que puedan ejecutar ciertas funciones críticas referentes a las transferencias de criptomonedas.

Manticore [5] realiza análisis de programas binarios y cualquier número de contratos inteligentes en interacción. Cuenta con un motor de ejecución simbólica agnóstico a la plataforma subyacente, y hace énfasis en el análisis de propiedades definidas por el usuario mediante el uso de callbacks y APIs programables. Recientemente, otros estudios [6] indicaron que Manticore resulta prometedor como herramienta de ejecución simbólica para contratos inteligentes.

3.2. Blockchains

Una blockchain es una base de datos pública, distribuida e inmutable. Las transacciones se agrupan en conjuntos de *bloques*, que son la unidad mínima con la que se actualiza el estado de la blockchain. El historial de bloques agregados a la blockchain (y por ende, las transacciones ejecutadas) es público, y el estado actual de la red puede calcularse a partir de este historial.

Las redes de blockchain, al ser sistemas distribuidos, cuentan con protocolos de consenso para que las partes participantes puedan acordar cuál es el estado actual de la red. La elección de protocolo de consenso es uno de los elementos más decisivos con respecto a una blockchain, y suelen estar pensados para garantizar propiedades de seguridad criptográfica sobre su estado [17][23][48]. Ethereum, que será la tecnología de blockchain de la que hablaremos de aquí en adelante, cambió el protocolo de su red principal desde un protocolo Proof-of-Work llamado `Ethash` hacia un protocolo Proof-of-Stake denominado `Gasper` el 15 de Septiembre de 2022 en un proceso apodado “The Merge” (“La Fusión”) [30] [47] [7] [15]. Desde la funcionalidad de los contratos inteligentes este evento es mayoritariamente un detalle; lo único que es pertinente saber es que a fines prácticos se considera que en Ethereum es imposible que ningún actor controle el ingreso de, o altere, la información en la red.

3.2.1. Contratos Inteligentes en Ethereum y Solidity

A continuación explicaremos los aspectos relevantes de la arquitectura de la red Ethereum. Queremos abordar cómo los contratos inteligentes existen en ella, y cuáles son las peculiaridades de su ejecución. Más adelante, explicaremos las características del lenguaje Solidity y las abordaremos con un ejemplo.

Las transacciones en Ethereum se refieren a propiedades sobre *direcciones*, como el balance de criptomonedas o el estado de los contratos inteligentes. Los contratos inteligentes se corresponden con direcciones dentro de la blockchain, de la misma forma que se corresponden con direcciones los usuarios “humanos”. De hecho, en Ethereum los contratos inteligentes y los usuarios tradicionales cuentan con dos tipos de *accounts* distintas, que pueden pertenecer a usuarios humanos por un lado o a contratos inteligentes por el otro. En el caso de los usuarios humanos el almacenamiento que le corresponde a cada dirección guarda sólo información sobre el balance de criptomonedas, mientras que un contrato inteligente utiliza dos campos adicionales: el `codeHash`, que es utilizado para

obener el código ejectuable del contrato y el `storageRoot`, que es necesario para conocer una especie de memoria persistente exclusiva del contrato. Ambos tipos de account pueden recibir o enviar mensajes y Ether[1]. La diferencia en los dos tipos radica en qué ocurre cuando la account es seleccionada como destinataria de una transacción. Para las accountes de usuarios humanos simplemente se registra el suceso en la blockchain. En el caso de un contrato inteligente, el momento de recibir una transacción es el momento en el que el programa se ejecuta, y el mensaje que dispara la ejecución es utilizado como los parámetros de entrada de la misma. Para poder incluir la transacción en el próximo estado de la blockchain, el minero[2] que registre la transacción debe levantar la EVM y simular la ejecución del contrato para conocer su estado final.

La EVM es una máquina de pila casi turing completa. El lenguaje assembly asociado cuenta con instrucciones para números enteros, instrucciones para control de flujo y otras instrucciones referentes a particularidades de la ejecución en blockchain, como el contenido de las transacciones de la misma [12]. A pesar de que el conjunto de programas expresablas en el lenguaje de la EVM es turing completo, la cantidad de operaciones que se pueden ejecutar dentro de una única transacción están limitadas artificialmente por la cantidad de *gas* asociado a la transacción.

El gas es una unidad de medida del esfuerzo de cómputo, y está ideado como elemento limitante de lo que se les permite hacer a los contratos inteligentes en Ethereum. Debido a que los mineros deben ejecutar los contratos inteligentes en hardware y con recursos de los que ellos son dueños, el gas asociado a una transacción permite conocer cuánto deben pagarles los remitentes a los mineros por incluir las transacciones en un bloque. Para controlar este costo, los remitentes de transacciones pueden establecer una cota superior al gas que tienen pensado gastar, al mismo tiempo que pueden ofertar distintos montos de criptomonedas por unidad de gas gastada. Por otro lado, existe una cota superior a la cantidad de gas que tiene permitido consumir una transacción, lo que significa que una ejecución dada de la EVM siempre termina en un número finito de instrucciones.

En la actualidad, existen lenguajes utilizados para programar contratos en la red Ethereum. De ellos, el que más adopción que tiene actualmente es Solidity. Es un lenguaje imperativo curly-brace que provee una sintaxis similar a la de las clases en lenguajes orientados a objetos. La sintaxis de Solidity incluye variables de estado, un método constructor, métodos internos (ejecutables sólo por el mismo contrato), y un conjunto de métodos externos que representan la interfaz con otros contratos y el mundo, con la que los usuarios pueden interactuar.

3.2.2. Ejemplo

En el fragmento de código 3.2 presentamos un ejemplo de un programa escrito en Solidity, `SimpleMarketplace`[3], el cual implementa un mecanismo simple para vender un

[1] Ether es la criptomoneda utilizada por Ethereum.
[2] Los “mineros” son quienes participan en el protocolo de la blockchain y registran las transacciones en bloques.
[3] Este contrato es extraido de los contratos de ejemplo “Mircrosoft Azure Blockchain Workbench” [34]

bien. A continuación haremos un breve análisis línea por línea del contrato.

En la línea 2 podemos ver que se define un tipo `enum` "`StateType`" que tiene tres valores posibles; que representan estados de completitud de la venta. Las líneas 3 a 8 definen las variables de estado del contrato, entre las que tenemos variables de tipo `int`, `StateType` y `address`, indicando además mediante la palabra clave `public` que estas variables resultan accesibles a contratos externos. El tipo `address` utilizado en las variables `InstanceOwner` y `InstanceBuyer` consiste en valores enteros de 20 bytes de tamaño, y se utiliza para representar direcciones en la blockchain.

```
1  pragma solidity >=0.4.25 <0.9.0;
2  pragma experimental ABIEncoderV2;
3
4  contract SimpleMarketplace {
5
6      enum StateType {ItemAvailable, OfferPlaced, Accepted}
7      address public InstanceOwner;
8      string public Description;
9      int public AskingPrice;
10     StateType public StateEnum;
11     address public InstanceBuyer;
12     int public OfferPrice;
13
14     constructor(string memory description, int price, address sender) public
15     {
16         InstanceOwner = sender;
17         AskingPrice = price;
18         Description = description;
19         StateEnum = StateType.ItemAvailable;
20     }
21
22     function MakeOffer(int offerPrice) public
23     {
24         require (offerPrice != 0 && StateEnum == StateType.ItemAvailable &&
               InstanceOwner != msg.sender);
25         InstanceBuyer = msg.sender;
26         OfferPrice = offerPrice;
27         StateEnum = StateType.OfferPlaced;
28     }
29
30     function Reject() public
31     {
32         require (StateEnum == StateType.OfferPlaced && InstanceOwner == msg.
               sender);
33         StateEnum = StateType.ItemAvailable;
34     }
35
36     function AcceptOffer() public
37     {
38         require (StateEnum == StateType.OfferPlaced && msg.sender ==
               InstanceOwner);
39         StateEnum = StateType.Accepted;
40     }
41 }
```

Listing 3.2: Contrato Inteligente `SimpleMarketplace` en Solidity

Observando el método `constructor` de `SimpleMarketplace`, que es el método que siempre se llama al desplegar una instancia del contrato, vemos que define el objeto que se planea vender, indicando la descripción, el precio del producto y la dirección de su dueño actual [4]. Luego, las líneas 20, 28 y 34 indican precondiciones de los métodos `MakeOffer`, `Reject` y `AcceptOffer`. En ellas, la expresión `msg.sender` se refiere a la dirección desde la que se envió el mensaje que disparó la ejecución actual del contrato. Si prestamos atención y debido a que el valor inicial de `SateEnum` es `ItemAvailable`, el único método que está permitido llamar inmediatamente después del constructor es `MakeOffer`. Mediante este, un potencial comprador puede realizar ofertas sobre el producto indicando un precio, que luego el dueño puede aceptar o rechazar. Si el dueño original no está satisfecho con la oferta puede rechazarla llamando a `RejectOffer`, regresando al estado incial en el que se aceptan nuevas ofertas. Si eventualmente al dueño le interesa la última oferta realizada y la acepta, llama a `AcceptOffer` y termina satisfactoriamente la ejecución del contrato, ya que a partir de ese estado ningún otro método se encuentra habilitado. Los detalles de la venta realizada permanecen expuestos en las variables públicas del contrato, que a pesar de quedarse bloqueado a nuevos llamados a métodos permanece visible en la blockchain.

3.3. Enabledness-Preserving Abstractions

Una EPA es un Labeled Transition System (LTS) finito que busca abstraer el comportamiento de un contrato inteligente agrupando los estados del contrato en base a cuáles de sus metodos están habilitados [11]. Las transiciones en estos LTS representan el llamado a una función del contrato, indicando cómo un llamado a una función puede transformar el contrato de un estado abstracto a otro.

Comenzando por un ejemplo, en la figura 3.2 presentamos la EPA del contrato `SimpleMarketplace`. El estado inicial, denominado **A**, está etiquetado `init` e indica el estado "vacío" previo al llamado al constructor del contrato. Lo que podemos ver es que luego de ejecutar el método constructor se transiciona en la EPA a un único otro estado, al que llamamos **B**. Allí, la etiqueta `_MakeOffer` indica que `MakeOffer` es el único método que se encuentra habilitado en **B**. La única transición desde **B**, etiquetada `MakeOffer`, indica lo que ocurre cuando se ejecuta ese método desde ese estado. Como vemos, seguir esa transición nos traslada al estado **C**, que es un estado del contrato en el que solamente `AcceptOffer` y `Reject` se encuentran habilitados. Luego, desde el estado **C** podemos ejecutar cualquiera de los dos métodos; la transición por `Reject` nos llevará de vuelta al estado **B** y la transición por `AcceptOffer` nos llevará al estado final **D**. Este último, etiquetado "`vacio`" indica que ningún método se encuentra habilitado, por lo que representa el fin forzoso de la ejecución.

[4] En la definición de `constructor`, la palabra clave `memory` indica qué estrategia debe usarse para sostener en memoria el string que entra por parámetro. Otras opciones, como `calldata`, impactan en el costo de gas de la función.

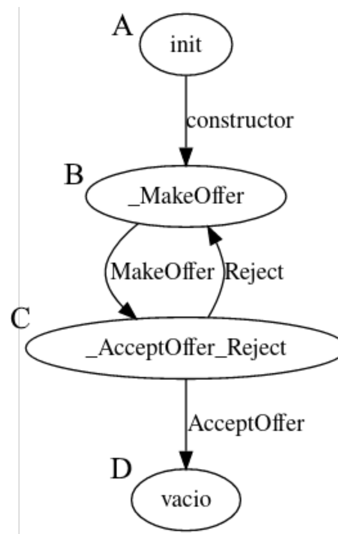


Fig. 3.2: EPA de `SimpleMarketplace`. Las etiquetas en los estados indican los métodos que se encuentran habilitados. Las etiquetas en las transiciones indican el método por el que ocurre la transición.

3.4. Modelo Formal

Dado que trabajaremos sobre el código fuente de un contrato escrito en Solidity, nos interesa formalizar qué aspectos del contrato consideraremos relevantes. Para obtener una discusión más detallada de estas formalizaciones de los artefactos de código, referirse a De Caso et. al. [11]. Asimismo, la abstracción presentada del comportamiento de los contratos inteligentes origina en Godoy et. al. [20]. En esta sección solamente presentamos una compatibilización de los formalismos para facilitar la discusión de las EPAs más adelante.

Dicho eso, llamaremos *configuraciones* a las posibles combinaciones de las variables de estado del contrato y de la blockchain, y notaremos C al conjunto de todas las posibles configuraciones.

Definition 3.41. (Formalización de un contrato inteligente) Definimos a un contrato inteligente como la tupla $SC = \langle M, F, R, inv, init \rangle$ donde:

- $M = m_1, \ldots m_n$ es el conjunto finito de métodos externos definidos en la interfaz del contrato
- F es un conjunto de funciones indexadas por M.
  Para cada $m \in M$, $F_m : C \times \mathbb{Z} \to (C \cup \perp)$ es la implementación del método m.
- R es un conjunto de precondiciones indexado por M.
  Para cada $m \in M$, $R_m : C \times \mathbb{Z} \to \{\textbf{true}, \textbf{false}\}$ indica si el método m está habilitado para la configuración y parámetros indicados
- $inv : C \to \{\textbf{true}, \textbf{false}\}$ indica si una configuración cumple el invariante del contrato

- $init : C \to \{\textbf{true}, \textbf{false}\}$ indica si una configuración puede ser resultante de ejecutar el constructor del contrato

Una particularidad presente en los métodos de los contratos inteligentes es que algunas instrucciones como `msg.sender`, `msg.value`, `tx.gasprice` o `tx.origin` [5] hacen referencia a variables de la blockchain. Sin embargo podemos modelar estas variables, junto con los parámetros explícitos como input codificable en $\mathbb{Z}$ (los números enteros) sin pérdida de generalidad [11]. La semántica de un contrato la definimos como el siguiente Labeled Transition System:

Definition 3.42. (Semántica de un contrato inteligente) Dado $SC = \langle M, F, R, inv, init\rangle$ un contrato inteligente, su semántica está provista por el LTS concreto $L_c = \langle \sigma, S_c, S_{0c}, \Delta_c\rangle$ que satisfaga:

- $S_c = \{conf | conf \in C \wedge inv(conf) = \textbf{true}\}$
- $S_{0c} = \{conf | conf \in S_c \wedge init(conf) = \textbf{true}\}$
- $\sigma = (F \times \mathbb{Z}) \cup \tau$ es el conjunto de todos los posibles llamados a funciones, junto con un elemento τ que representa un cambio en la blockchain que ocurra de manera independiente al contrato
- $\Delta_c \subseteq S_c \times \sigma \times S_c$
- $\forall s_1, s_2 \in S_c, m \in M, z \in \mathbb{Z}.$
  $(s_1, (F_m, z), s_2) \in \Delta_c \iff \Big(R_m(s_1, z) = \textbf{true} \wedge F_m(s_1, z) = s_2\Big)$
  $(s_1, \tau, s_2) \in \Delta_c \iff$ un cambio independiente en la blockchain puede llevarnos del estado s_1 al estado s_2

Notar que para cualquier contrato, el conjunto S_c de estados de su LTS concreto es infinito. Esto es porque las configuraciones tienen en cuenta variables de la blockchain externas al contrato. Incluso para contratos donde las configuraciones de las variables internas es finita, siempre habrá infinitas configuraciones de las variables externas.

A la EPA (es decir, el LTS abstracto) la definimos entonces de la siguiente manera:

Definition 3.43. (Enabledness-Preserving-Abstraction) Dado $SC = \langle M, F, R, inv, init\rangle$ un contrato inteligente y $L_c = \langle \sigma, S_c, S_{0c}, \Delta_c\rangle$ su LTS concreto, entonces el LTS asbtracto $L_A = \langle M \cup \hat{\tau}, \#\text{M}, P_0, \Delta_A\rangle$ es una EPA del contrato y $\alpha : S_c \to \#\text{M}$ es la función de abstracción, donde se cumple que:

- #M es el conjunto de partes de M

[5] `msg.value` indica cuánto Ether está recibiendo el contrato. `tx.gasprice` indica cuánto Ether se le cobra al remitente por cada unidad de gas utilizada y `tx.origin` se refiere a el usuario humano que haya enviado el mensaje que disparó la ejecución actual.

- $\forall s \in S_c \, . \, \alpha(s) = \{m | m \in M \wedge \exists z \in \mathbb{Z}. R_m(s, z) = \textbf{true}\}$
- $P_0 = \{\alpha(s_0) | s_0 \in S_{0c}\}$
- $\forall s_1, s_2 \in S_c, m \in M, z \in \mathbb{Z}.$
  $(s_1, (F_m, z), s_2) \in \Delta_c \Rightarrow (\alpha(s_1), R_m, \alpha(s_2)) \in \Delta_A$
  $(s_1, \tau, s_2) \in \Delta_c \Rightarrow (\alpha(s_1), \hat{\tau}, \alpha(s_2)) \in \Delta_A$

El conjunto de estados de la *EPA* es #M (el conjunto de partes de los métodos). La *función de abstracción* de los estados s_c del LTS concreto a los estados abstractos es $\alpha(s_c) =$ "el conjunto de métodos cuyas precondiciones son satisfechas por s_c". Una transición en la EPA entre los estados s y s' etiquetada con el método m significa que existen algún estado concreto s_c y un valor de entrada z para los que $\alpha(s_c) = s$ y $F_m(s, z) = s'_c$ con $\alpha(s'_c) = s'$. Es decir que algún llamado de m a un estado que se abstrae a s nos da de resultado otro estado que se abstrae a s'. Una transición de s a s' etiquetada con $\hat{\tau}$, la versión abstracta de τ, indica que existe un estado concreto s_c con $\alpha(s_c) = s$ y que puede ocurrir $\alpha(\tau(s_c)) = s'$.

4. MANTICORE

En esta sección haremos un breve repaso de las funcionalidades provistas por `Manticore`. Luego podremos discernir dadas estas funcionalidades la mejor estrategia para utilizar `Manticore` como el back end de la construcción de EPAs. Toda esta sección se refiere a la versión de `Manticore 0.3.7`.

`Manticore` es un proyecto desarrollado por `TrailOfBits` lanzado en 2017. Es una herramienta de ejecución simbólica, implementada en Python, que soporta análisis para diversas plataformas: EVM, bytecode nativo (arquitecturas `x86`, `x86_64`, `aarch64` y `ARMv7`) y WASM. Principalmente funciona como un motor de ejecución simbólica programable (mediante APIs en Python), aunque también incluye una herramienta plug-and-play por línea de comandos, y existe un proyecto que intenta integrar la herramienta con una interfaz gráfica, `ManticoreUI` [32] que nunca se lanzó.

4.1. Herramienta por Línea de Comandos

El comportamiento por defecto de la herramienta de línea de comandos varía mucho dependiendo de la plataforma a la que es aplicada, pero en general busca explorar todos los caminos de ejecución factibles en el código fuente provisto, utilizando valores simbólicos para cada valor generalmente introducido por usuarios. Luego, por cada camino explorado, genera un caso de test (es decir, genera un valor concreto que fuerce el camino para cada valor "input" simbólico). Además, la exploración de los caminos incluye el seguimiento de algunas propiedades interesantes por defecto, dependientes de la plataforma. Para los binarios nativos, por ejemplo, registra el conjunto (total) de instrucciones visitadas, y registra para cada caso de test el número y la traza exacta de instrucciones ejecutadas.

En el caso de programas de la EVM, la herramienta por consola funciona con código fuente Solidity (no acepta precompilados). Para explorar caminos de ejecución la herramienta toma los métodos externos del contrato y los ejecuta (en cualquier orden) hasta alcanzar 100 % de line coverage o, dentro de un límite si es que se introdujo uno, hasta cubrir todo el espacio de secuencias de llamados a métodos externos. De no alcanzar ninguno de estos dos criterios, termina el análisis por time out. La herramienta cuenta con una batería de `detectors` que registran eventos de interés específico a Ethereum, como la presencia de integer overflows, la ejecución de opcodes inválidos, la lectura de memoria o storage no inicializado, bugs de reentrancy o la ejecución de ciertas instrucciones específicas con parámetros controlados por el usuario. Estos `detectors` se encuentran apagados por defecto, al igual que por defecto se descartan caminos que incluyan el rollback de una transacción. Esto significa que, por ejemplo, para realizar una simple búsqueda de incumplimiento de una aserción (la instrucción `assert` en Solidity), es preciso activar el modo detallado (`--thorough-mode`) de la herramienta.

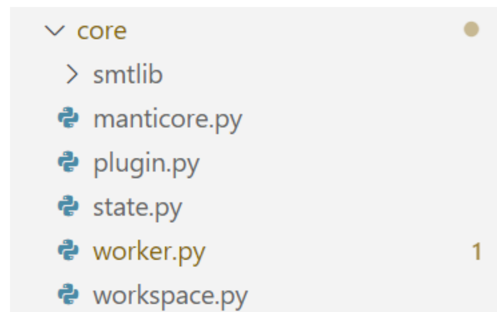


Fig. 4.1: Módulos que componen el mecanismo central de ejecución simbólica de Manticore

4.2. Manticore-Verifier

Manticore cuenta con otra herramienta por consola de comandos de análisis de smart contracts denominada `manticore-verifier`. Permite marcar ciertos métodos externos como "invariantes" que la herramienta luego busca falsificar. Una fortaleza de esta herramienta es que la sintaxis para marcar los invariantes es la misma que la utilizada por `Echidna`, un fuzzer desarollado también por `TrailOfBits` [4], permitiendo el análisis por ambas herramientas con una única intervención manual.

4.3. Arquitectura

La arquitectura de Manticore se organiza en tres partes:

- El mecanismo central de ejecución simbólica
- Los módulos que implementan la simulación de cada una de las plataformas soportadas (Ethereum, `x86`, etc)
- SMT solvers externos

Por defecto, la instalación de Manticore incluye una instalación del `Z3 Theorem Prover`, un SMT solver desarrollado por Microsoft Research desde 2012 [42]. Sin embargo, Manticore puede integrarse con cualquier SMT solver que se conforme a la interfaz definida por `SMTLIB2` [43], como lo es por ejemplo también `Yices` [45], otro SMT solver open source.

El mecanismo central de ejecución simbólica de Manticore está compuesto por varios módulos de Python: **`smtlib`**, **`manticorebase`**, **`plugin`**, **`state`**, **`worker`** y **`workspace`**, representados en la figura 4.1.

`smtlib` provee a los demás módulos de la aplicación APIs para generar y manipular variables, expresiones y constraints simbólicas, y para realizar consultas de (in)satisfacibilidad sobre las expresiones simbólicas generadas, delegando estas consultas en última instancia al SMT solver externo. Los módulos **`manticorebase`** y **`state`** implementan en conjunto el

mecanismo central de ejecución simbólica de Manticore asumiendo lo mínimo posible sobre las instrucciones emuladas [1]. Por último, **worker** y **workspace** sirven de auxiliares que soportan los aspectos de perststistencia, entrada/salida y multithreading, entre otros, de la aplicación. El módulo **plugin** provee una API de *callbacks* que otorgan acceso al estado interno emulado en distintos momentos de la ejecución. Este módulo es una parte central de la API programable accesible al usuario de Manticore, pero también es la manera en la que la aplicación base puede realizar sus análisis de coverage, etc.

La abstracción principal para la ejecución simbólica utilizada por Manticore es la del objeto `state`. Un `state` representa, habiendo realizado un camino de ejecución particular, el estado del programa emulado hasta cierto punto. Los `state` son responsables de conocer cuáles son los próximos pasos en su ejecución, cuál es el conjunto de variables simbólicas que existieron en su ejecución, y qué conjunto de fórmulas deben satisfacerse. Asimismo, cada `state` individual cuenta con una instancia entera emulada del estado del programa (en el caso de Ethereum, esto es simulaciones enteras de blockchains). Por otro lado, el estado global de la aplicación (mantenido por el módulo **manticorebase**) consiste simplemente en una colección de `state`.

4.4. API programable

La principal API presentada al usuario permite manejar el conjunto global de `state`, siempre "entre medio" de la ejecución de métodos del contrato (es decir, antes de comenzar a ejecutarlos o después de que terminen, pero no durante). Los principales métodos de la API permiten realizar acciones globales como introducir nuevas constraints, ejecutar métodos de un smart contract, deployear un contrato nuevo, o iniciar la generación de casos de test a partir de los `state`. Además, permiten acceder a `state` individuales (siempre y cuando no estén corriendo) y leer y/o modificar los elementos simbólicos de su estado. Sin embargo, al menos mediante la API expuesta, no es posible expandir el camino de ejecución de sólo algunos `state`. Esto siempre debe hacerse mediante la ejecución de métodos externos al nivel más alto, afectando a todos los `state` [2].

Por otro lado los callbacks como los del módulo **plugin** permiten interactuar con los `state` mientras estos se ejecutan. Los callbacks pueden suscribirse a cualquiera de los eventos publicados nativamente por la aplicación, que son de naturaleza muy amplia y granularidad diversa, y no simplemente eventos que modifiquen el estado de los `state`. Algunos de los eventos pueden categorizarse en:

- eventos de `state` (`will\did_fork_state, will\did_terminate_state`)
- eventos de smt (`will\did_solve`)
- eventos de plataforma (`will\did_open_transaction`, `will\did_evm_read_storage`)

[1] Es decir, sobre la plataforma que use el programa que se está analizando
[2] Veremos que esta limitación guió el diseño del algoritmo presentado en la sección 5.2

Los callbacks pueden resultar útiles para debugear o mantener el registro de propiedades, pero también es posible utilizarlos para modificar el estado de la ejecución en vivo.

5. CONSTRUCCIÓN DE EPAS MEDIANTE EJECUCIÓN SIMBÓLICA

En las últimas dos secciones vimos las formalizaciones relacionadas con las EPAs y las capacidades de Manticore. A continuación, presentaremos dos algoritmos para la construcción de EPAs. El algoritmo clásico es presentado, con algunas diferencias de notación, como fue presentado por De Caso et. al en 2013 [11]. El algoritmo novedoso hace uso específicamente de ejecución simbólica y, al haber sido diseñado para Manticore, se adapta a las peculiaridades mencionadas en la sección anterior.

5.1. Construcción clásica de EPAs

Tradicionalmente, existe un algoritmo genérico para la construcción de *EPAs* de artefactos de código. Por cuestiones de notación, para ver el algoritmo de generación de EPAs es conveniente definir el siguiente predicado presentado por De Caso et. al [11] sobre las configuraciones dado un conjunto de precondiciones de un contrato:

Definition 5.11 (Predicado de un conjunto de métodos)**.** Dados un contrato $SC = \langle M, F, R, inv, init \rangle$ y un conjunto de metodos, $\mathcal{M} \subseteq M$, definimos $pred_{\mathcal{M}} : \mathcal{C} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ como

$$pred_{\mathcal{M}}(c) \iff inv(c) \land \bigwedge_{m \in \mathcal{M}} \exists p \in \mathbb{Z}.R_m(c, p) \land \bigwedge_{m \notin \mathcal{M}} \nexists p \in \mathbb{Z}.R_m(c, p)$$

El algoritmo, que presentamos a continuación, genera la porción de la EPA que es alcanzable desde P_0 (los estados iniciales), realizando *Breadth-First-Search* en el grafo de las transiciones [11].

Algoritmo 1 Construcción de EPAs

Input $SC = \langle M, F, R, inv, init \rangle$ contrato
Output La *EPA* $L_A = \langle \Sigma, S, P_0, \Delta \rangle$

1: $\Sigma = M$; $S = \emptyset$
2: $\Delta(s, m) = \emptyset \quad \forall s \in 2^R, m \in M$
3: $\mathcal{M}^- = \{m \in M | \forall c \,.\, init(c) \Rightarrow \nexists p \in \mathbb{Z} \,.\, R_m(c, p)\}$
4: $\mathcal{M}^+ = \{m \in M | \forall c \,.\, init(c) \Rightarrow \exists p \in \mathbb{Z} \,.\, R_m(c, p)\}$
5: $P_0^c = \{\mathcal{M} \in \#M | \mathcal{M}^+ \subseteq \mathcal{M} \wedge \mathcal{M}^- \cap \mathcal{M} = \emptyset\}$
6: $P_0 = \{\mathcal{M} \in P_0^c | \exists c.init(c) \wedge pred_{\mathcal{M}}(c)\}$
7: $W =$ Cola con los elementos de P_0
8: **while** hay un conjunto $\mathcal{M}$ en la cabeza de W **do**
9: $\quad S = S \cup \{\mathcal{M}\}$
10: $\quad$ **for** $m \in \mathcal{M}$ **do**
11: $\quad\quad \mathcal{N}^- = \{n \in M | \forall c \in \mathcal{C}, p \in \mathbb{Z} \,.\, pred_{\mathcal{M}}(c) \wedge R_m(c, p) \Rightarrow \nexists p' \in \mathbb{Z} \,.\, R_n(F_m(c, p), p')\}$
12: $\quad\quad \mathcal{N}^+ = \{n \in M | \forall c \in \mathcal{C}, p \in \mathbb{Z} \,.\, pred_{\mathcal{M}}(c) \wedge R_m(c, p) \Rightarrow \exists p' \in \mathbb{Z} \,.\, R_n(F_m(c, p), p')\}$
13: $\quad\quad S^C = \{\mathcal{N} \in \#M | \mathcal{N}^+ \subseteq \mathcal{N} \wedge \mathcal{N}^- \cap \mathcal{N} = \emptyset\}$
14: $\quad\quad$ **for** $\mathcal{N} \in S^C$ **do**
15: $\quad\quad\quad$ **if** $\exists c \in \mathcal{C} \,.\, pred_{\mathcal{M}}(c) \wedge \exists p \in \mathbb{Z} \,.\, R_m(c, p) \wedge pred_{\mathcal{N}}(F_m(c, p))$ **then**
16: $\quad\quad\quad\quad \Delta(\mathcal{M}, m) = \Delta(\mathcal{M}, m) \cup \mathcal{N}$
17: $\quad\quad\quad\quad$ **if** $\mathcal{N} \notin S \wedge \mathcal{N} \notin W$ **then**
18: $\quad\quad\quad\quad\quad W.push(\mathcal{N})$
19: $\quad\quad\quad\quad$ **end if**
20: $\quad\quad\quad$ **end if**
21: $\quad\quad$ **end for**
22: $\quad$ **end for**
23: **end while**
24: **return** $\langle \Sigma, S, P_0, \Delta \rangle$

Aquí, calcular los conjuntos $\mathcal{M}^-$, $\mathcal{M}^+$, $\mathcal{N}^-$ y $\mathcal{N}^+$ es una optimización que permite reducir la cantidad de transiciones candidatas de la EPA [11].

Decidir si una transición pertenece o no la EPA, como está planteado en el algoritmo, es resolver problemas de validez de fórmulas de primer orden. Esto en general es indecidible, sin embargo la sugerencia principal consiste en transformar estas preguntas de validez de fórmulas de primer orden en problemas de alcanzabilidad de código, dado que se espera que las precondiciones y los invariantes definidos en la formalización 3.41 estén debidamente implementados en el contrato.

En general, por optimizaciones de gas, es común que en los métodos externos los contratos definan las precondiciones explícitamente mediante instrucciones `require`, como en el contrato ejemplo 3.2 `SimpleMarketplace`. Dado que los inputs externos nunca garantizan estar bien formados, resulta menos costoso para el contrato abortar estas ejecuciones lo antes posible en lugar de hacerlas avanzar hasta llegar a un estado de error. Sin embargo, debido a que no se espera que sea posible generar instancias que no satisfagan el invariante, por el mismo motivo de ahorro de gas, no es usual contar con una implementación explícita del invariante en el código fuente del contrato. Tenerlo implicaría ejecutar

código que se espera que dé siempre el mismo resultado, por lo que resulta más eficiente no hacerlo.

5.2. Algoritmo alternativo

Implementar el invariante del contrato, además de no ser una práctica estándar, es a menudo exigente para el programador, y es también una actividad propensa a errores. Sin embargo, es importante en la definición 3.42 de la semántica de un contrato inteligente la presencia del invariante. Intentar construir EPAs incluyendo estados que no satisfagan el invariante suele producir abstracciones demasiado sobreaproximadas [11]. Por eso, en continuación del trabajo presentado por Godoy et al. [20], trabajaremos construyendo las EPAs a partir de las precondiciones explícitas dadas por las declaraciones `require` al comienzo de los métodos, y haciendo uso del invariante implícito dado por los métodos mismos del contrato.

Trabajemos con la acepción de que un contrato es correcto si la ejecución de todos sus métodos preservan el invariante si se satisfacen las precondiciones del método. La idea es que bajo la asunción de que un contrato es correcto, ejecutar cualquier sucesión de métodos del contrato con parámetros válidos (es decir, que satisfagan las precondiciones) genera instancias que satisfacen el invariante. Esto significa que explorar cadenas de transacciones que comienzen por el constructor siempre considera estados que satisfacen el invariante, siempre y cuando cada llamado individual a métodos cumpla las precondiciones correspondientes. Entonces, es posible asumir que se satisface el invariante si se presenta una traza de llamados válidos a métodos que comienza por el constructor.

Por otro lado, si todos los métodos de un contrato implementan explícitamente sus precondiciones, podemos decir aún mas. Si este es el caso entonces es imposible que ningún llamado a un método genere un estado inválido. En su lugar, un llamado que no satisfaga las precondiciones del método será revertido. De esta manera, *cualquier* traza de métodos que comienze por el constructor genera una instancia del contrato que satisface el invariante, si los métodos implementan explícitamente sus precondiciones.

En esta sección presentamos el algoritmo que construye EPAs haciendo uso de una herramienta de ejecución simbólica, siguiendo la idea esbozada anteriormente para mantener el análisis en estados que satisfagan el invariante. Dado que trabajaremos asumiendo que no hay acceso explícito al invariante, la nueva formalización que utilizaremos para los contratos inteligentes es la siguiente:

Definition 5.21. (Formalización laxa de un contrato inteligente) Definimos a la versión laxa de un contrato inteligente como la tupla $SC = \langle M, F, R, Constructor \rangle$ donde:

- M, F y R son iguales a la definición original
- *inv* no forma parte de SC porque asumimos que se preserva luego de cada llamado a $F_m(c, p)$ independientemente de si $R_m(c, p) = \textbf{true}$

- $Constructor \in M$ reemplaza el predicado *init* y representa un método del contrato que genera las instancias iniciales.

El LTS concreto (la semántica) y el LTS abstracto (la EPA) de smart contracts que usan esta definición pueden formalizarse de manera análoga a las secciones anteriores. Por cuestiones de notación, introduciremos algunas funciones sobre los caminos posibles en los métodos de un contrato:

Definition 5.22 (Conjunto de posibles caminos de ejecución de una secuencia de funciones)**.** Dados un contrato $SC = \langle M, F, R, Constructor \rangle$ definimos la función `paths_of` que opera en el dominio de las secuencias de funciones $f_1, f_2, \cdots f_k$ tales que $\forall i, f_i \in F \cup R$:

$$\begin{aligned} \texttt{paths_of}(f_1, f_2, \cdots f_k) = \{p | p \text{ es una secuencia de instrucciones del código fuente de SC} \\ \text{que podría ser un camino que toma la ejecución de } f_1, f_2, \cdots f_k\} \end{aligned} \tag{5.1}$$

En particular, `paths_of` se refiere al conjunto de caminos que son efectivamente factibles. Es decir, no se consideran caminos de ejecución para los que no exista una serie de inputs que obligan a recorrerlos.

Definition 5.23 (Resultado simbólico del camino de ejecución en una secuencia de funciones)**.** Dados un contrato $SC = \langle M, F, R, Constructor \rangle$ definimos la función `sym_res_of` que opera en el dominio de los caminos de secuencias de funciones:

$$\texttt{sym_res_of}(p) \quad = \quad \text{le asocia un valor simbólico a cada función que fue llamada en } p \tag{5.2}$$

A continuación usaremos la notación "$\mathcal{M} = sy$" donde $\mathcal{M}$ es un estado abstracto de la EPA y sy uno de los estados simbólicos mencionados anteriormente. Esta notación la usaremos para referirnos a la "fórmula de compatibilidad" entre estos dos. Brevemente, recordemos que un estado de la EPA $\mathcal{M}$ es un subconjunto de los métodos del contrato, y representa los estados concretos en los que los métodos contenidos en el subconjunto se encuentran habilitados, y los demás se encuentran deshabilitados. Podremos decir que para un camino de ejecución p de las funciones $R_{m_1}, R_{m_2} \cdots R_{m_n}$ el resultado simbólico de p, $sy = \texttt{symb_res_of}(p)$ es compatible con $\mathcal{M}$ sí y sólo sí para cada R_m, el resultado de R_{m_i} en sy es **true** $\iff m_i \in \mathcal{M}$.

Consideremos el siguiente ejemplo para el contrato `SimpleMarketplace`: Sean $\mathcal{M} = \{MakeOffer, Reject\}$ y p_1 que representa un camino hasta cierto punto de `SimpleMarketplace`. Sea $p_2 \in \texttt{paths_of}(R_{MakeOffer}; R_{AcceptOffer}; R_{Reject})$ que representa un camino posible de ejecutar las precondiciones del contrato. Además, sea $sy = \texttt{symb_res_of}(p_1; p_2)$, donde con ";" notamos la concatenación de caminos, tal que los resultados de las precondiciones en sy son $\{R_{MakeOffer} = r_1,\ R_{AcceptOffer} = r_2,\ R_{Reject} = r_3\}$. La fórmula de compatibilidad entre $\mathcal{M}$ y sy será $\Phi = (r_1 = \textbf{true} \land r_2 = \textbf{false} \land r_3 = \textbf{true} \land sy)$ y significa que en la ejecución de $(R_{MakeOffer}; R_{AcceptOffer}; R_{Reject})$ existen valores de entrada que recorren $p_1; p_2$ y que los resultados dan **true** para `MakeOffer` y `Reject` y false para `AcceptOffer`. Dicho de otra manera, esto es que existen parámetros de entrada que generan un estado que se abstrae a $\mathcal{M}$ luego de la ejecución de p_1. Luego, esto quiere decir que si la fórmula

Φ es satisfacible, entonces tenemos garantía de que luego de ejecutar p_1 el contrato *puede* encontrarse en un estado que se abstrae a $\mathcal{M}$. Por otro lado la no satisfacibildad de Φ indicaría que es imposible llegar al estado de la epa $\mathcal{M}$ luego de ejecutar $p1$. El algoritmo para expresar la fórmula de compatibilidad entre p_1 y sy es el siguiente:

Algoritmo 2 Operador "=" entre estados de una EPA y resultados simbólicos (ecuación de compatibilidad)

Input $SC = \langle M, F, R, Constructor \rangle$ contrato
Input $\mathcal{M}$ un estado de la EPA de SC
Input sy resultado simbolico de una ejecución de SC
Output La fórmula de compatibilidad Φ

1: $\Phi = sy$
2: **for** $m \in M$ **do**
3: res_m = valor de retorno de la última aparición de R_m en sy
4: **if** $m \in \mathcal{M}$ **then**
5: $\Phi = \Phi \wedge (res_m ==$ **true**$)$
6: **else**
7: $\Phi = \Phi \wedge (res_m ==$ **false**$)$
8: **end if**
9: **end for**
10: **return** Φ

Utilizando estas notaciones, podemos presentar el algoritmo3 que explora la porción de la EPA que es alcanzable desde P_0, sin acceso al invariante y haciendo uso de ejecución simbólica. Este algoritmo realiza una versión modificada de *Depth-First-Search* en el grafo de las transiciones, explorando varias aristas del grafo a la vez. En particular, en cada iteración del ciclo principal este algoritmo agrega múltiples transiciones a la EPA etiquetadas por un mismo método. La exploración siempre se realiza a partir del conjunto de estados de la EPA "$S_{current}$" que en cada iteración se actualiza con el conjunto de estados en los que puede resultar la ejecución del último método. Cuando esta exploración llega a un punto muerto, se retrocede usando la pila del *DFS* hasta otro punto en el que queden llamados de métodos sin explorar.

La decisión de explorar los estados de la EPA de esta manera conjunta se debe a las capacidades de la API programable de Manticore. Al ejecutar un método, debemos ejecutarlo en todos los caminos hasta el momento. Por este motivo es que exploramos las transiciones desde todos los estados del conjunto "$S_{current}$" cada vez, en lugar de realizar una exploración más tradicional en el grafo de transiciones.

Por último, podemos señalar que para garantizar *soundness* de las EPAs generadas, lo tradicional es consultar por la insatisfacibilidad de que una transición pertenezca a la EPA, e incluirla en el resultado en cualquier caso en el que no se pueda demostrar esto. Sin embargo, en este caso sólo estamos incluyendo las transiciones para las que tengamos una demostración de su existencia. Esta decisión, aunque discutible, fue prinicipalmente tomada para aprovechar la funcionalidad de generación de casos de test concretos de Manticore.

Algoritmo 3 Construcción de EPAs mediante ejecución simbólica

Input $SC = \langle M, F, R \rangle$ contrato
Output La *EPA* $L_A = \langle \Sigma, S, P_0, \Delta \rangle$

1: $\Sigma = M; S = \emptyset$
2: $\Delta(s, m) = \emptyset \quad \forall s \in \#M, m \in M$
3: $Paths_{pre} =$ `paths_of`$(F_{Constructor}; R_1; R_2; \ldots R_n)$
4: $Symb_{pre} = \{$`sym_res_of`$(p) | p \in Paths_{pre}\}$
5: $P_0 = \{s \in \#M | \exists sy \in Symb_{pre}$ **tal que** $SAT(sy = s)\}$
6: $S_{current} = P_0$
7: $W =$ Pila de tuplas con $(S_{current}, Symb_{pre})$
8: **while** hay un elemento en la cabeza de W **do**
9: $S = S \cup S_{current}$
10: **if** hay un $m \in M$ **tal que** $\{\mathcal{M} \in S_{current} | m \in \mathcal{M} \wedge (\mathcal{M}, m)$ no está marcado$\} \neq \emptyset$ **then**
11: $Paths_{post_m} =$ `paths_of`$(F_m; R_1; R_2; \ldots R_n)$
12: $Symb_{post} = \{$`sym_res_of`$(p_1; p_2) \mid p_1 \in Paths_{pre} \wedge p_2 \in Paths_{post_m}\}$
13: $S_{next} = \emptyset$
14: **for** $\{\mathcal{M} \in S_{current} | m \in \mathcal{M} \wedge (\mathcal{M}, m)$ no está marcado$\}$ **do**
15: marcar $\{(\mathcal{M}, m)\}$
16: **for** $\mathcal{N} \in \#M$ **do**
17: **if** $\exists sy_1 \in Symb_{pre}, sy_2 \in Symb_{post_m}$ **tales que** $SAT(sy_1 = \mathcal{M} \wedge sy_2 = \mathcal{N})$ **then**
18: $\Delta(\mathcal{M}, m) = \Delta(\mathcal{M}, m) \cup \mathcal{N}$
19: **end if**
20: **end for**
21: $S_{next} = S_{next} \cup \Delta(\mathcal{M}, m)$
22: **end for**
23: $S_{current} = S_{next}$
24: $Symb_{pre} = Symb_{post}$
25: $Paths_{pre} = (Paths_{Pre};\ Paths_{Post})$
26: W.Push$((S_{next}, Symb_{post}))$
27: **else**
28: $(S_{current}, Symb_{pre}) = W$.Pop$()$
29: **end if**
30: **end while**
31: **return** $\langle \Sigma, S, P_0, \Delta \rangle$

5.3. Limitaciones

La mayor evidente diferencia entre el algoritmo presentado en la seccion 5.2 y el algoritmo clásico es que en lugar de poder explorar transiciones entre estados completamente abstractos, la manera en la que explora las transiciones de la EPA implica que sólo está considerando ciertos caminos de métodos al evaluar cada transición. Esto se debe a que en cada paso selecciona un par (método, estado abstracto) que no haya explorado ya, pero la exploración **no** considera todos los estados posibles que puedan abstraerse al estado abstracto. En cambio, sólo puede considerarse un subconjunto de los estados que la herramienta pudo alcanzar hasta ese momento.

Consideremos el siguiente ejemplo, el contrato `BoundedStack`, cuyo código podemos ver en el fragmento de código 5.1. Este contrato consiste en una simple implementación de una pila como estructura de datos, con la peculiaridad de que la pila cuenta con un tamaño máximo. El contrato tiene sólo dos métodos externos: push y pop.

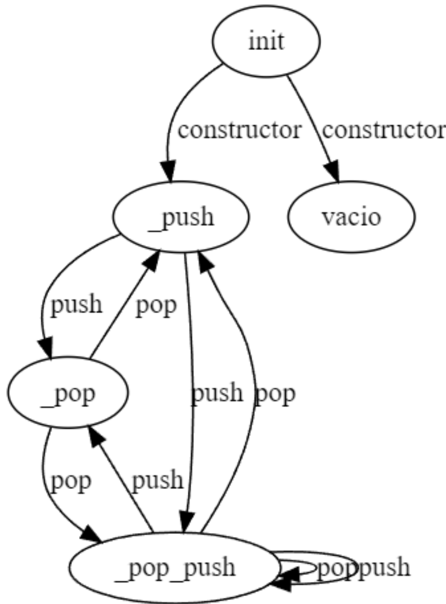
```
1  contract SizedStack {
2      uint256 public size;
3      uint256 public maxSize;
4      uint256[] internal_arr;
5      constructor() public {
6          maxSize = 10;
7          size = 0;
8      }
9      function isEmpty() public view returns (bool) {
10         return size == 0;
11     }
12     function top() public view returns (uint256) {
13         require(!isEmpty());
14         return internal_arr[size - 1];
15     }
16     function push(uint256 new_elem) public {
17         require(size < maxSize);
18         internal_arr.push(new_elem);
19         size += 1;
20     }
21     function pop() public returns (uint256) {
22         require(!isEmpty());
23         uint256 was = top();
24         internal_arr.pop();
25         size -= 1;
26         return was;
27     }
28 }
```

Listing 5.1: Contrato Inteligente `BoundedStack` en Solidity

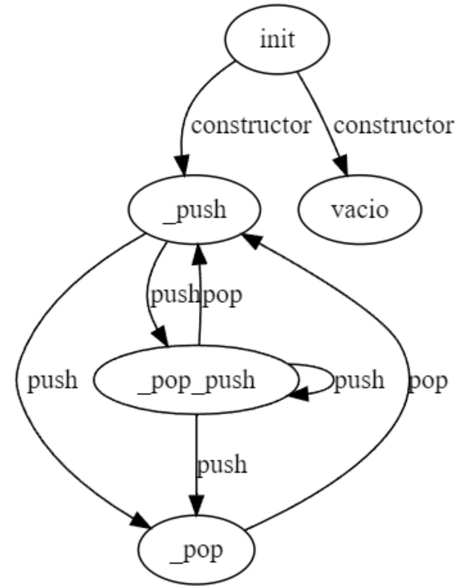
La EPA de este contrato por otro lado también es bastante sencilla, y la podemos ver en la imagen 5.1a. Por fuera de los estados **init** y **vacio**, contamos con solo tres estados alcanzables, que son **__push**, **__pop** y **__push__pop**. El estado **__push** se corresponde con la pila vacía, y es el que se produce al desplegar el contrato o luego de ejecutar pop reiteradas veces. Cuando la pila tiene elementos pero no está llena se encuentra en el estado **__push__pop**, al que se puede llegar tanto desde sí mismo, como desde **__push**

y desde **__pop**. Finalmente, el estado **__pop** se corresponde con la pila llena, y se puede llegar tanto desde **__push** (si el tamaño de la pila fuese uno) o desde **__push__pop**, pero no desde el constructor. Notoriamente, a menos que el contrato sea deployeado con un tamaño máximo de cero, nunca será posible alcanzar el estado **__vacio**.

Este es un buen ejemplo para observar la limitación introducida por el algoritmo alternativo al no explorar realmente todas las ejecuciones posibles de un método. Como se representa en la figura 5.1b, veremos que es posible que el análisis del contrato `BoundedStack` por el algoritmo alternativo no pueda encontrar algunas de las transiciones presentes en la EPA, dependiendo del orden en el que elija ejecutar los métodos del contrato. Realicemos manualmente la exploración que realizaría el algoritmo para este contrato y veamos que es posible llegar al resultado presentado en la figura 5.1b, en el que algunas transiciones son omitidas de la EPA.



(a) Enabledness Preserving Abstraction del contrato `BoundedStack`



(b) EPA errónea del contrato `BoundedStack` generada por el algoritmo alternativo. No se encuentran presentes las transiciones **__pop** a **__push__pop** mediante `pop` ni **__push__pop** a **__push__pop** mediante `pop`.

El algoritmo explora los estados alcanzables comenzando desde el constructor. Como vimos en la EPA de `BoundedStack`, los estados que resultan alcanzables son **__push** y **vacio**. Hasta ahora tenemos entonces $P_0 = \{\emptyset, \textbf{__push}\}, S = \{\emptyset, \textbf{__push}\}$ y que $\Delta(s, m) = \emptyset$ para todos los m y s. Luego, el algoritmo elige un método de los disponibles a ejecutar al azar, que en este caso sería solamente `push`. Luego de ejecutar ese método, los estados que resultan alcanzables desde el estado **__push** son **__push__pop** (con la restricción de que el tamaño de la pila sea mayor a uno) y **__pop** (con la restricción de que el tamaño de la pila sea exactamente uno). En este punto, $P_0 = \{\emptyset, \textbf{__push}\}, S = \{\emptyset, \textbf{__push}, \textbf{__push__pop}, \textbf{__pop}\}$ y $\Delta(\textbf{__push}, \texttt{push}) = \{\textbf{__push__pop}, \textbf{__pop}\}$.

Ahora, este es el punto en el que el próximo paso de la exploración del algoritmo puede cambiar el resultado final de la abstracción generada. Por un lado, podemos elegir explorar pop desde **__push__pop** y **__pop**, marcando los pares (**__push__pop**, pop) y (**__pop**, pop) como ya explorados, o explorar push desde **__push__pop** y dejar la exploración de push para más adelante. Por otro lado se puede explorar push, marcando el par (**__push__pop**, push). Si continuamos la ejecución suponiendo que en este paso se elige explorar pop, descubriremos dos transiciones nuevas: de **__push__pop** a **__push** mediante pop y de **__pop** a **__push** mediante pop también. Nos puede llamar la atención que desde los estados (concretos) que elegimos para explorar la ejecución de pop hay dos transiciones que no podemos ver: de **__pop** a **__push__pop** mediante pop y de de **__push__pop** a **__push__pop** mediante pop. Estas se deben a que en este momento todos los estados concretos que estamos considerando tienen su tamaño real de la pila en 1. Sin embargo, el algoritmo no presta atención a cuál fue la traza de métodos ejecutada para llegar hasta acá, por lo que marca a ambas tuplas (**__push__pop**, pop) y (**__pop**, pop) como ya exploradas. Esto significa que la función de transición Δ quedará definida en Δ(**__push__pop**, pop) = **__push** y Δ(**__pop**, pop) = **__push**. De aquí en adelante la ejecución del algoritmo proseguirá hasta terminar con (al menos) esas dos transiciones faltantes.

6. IMPLEMENTACIÓN

En esta sección describimos los puntos más importantes involucrados en la implementación de los algoritmos con Manticore. Además, haremos un breve resumen de los desafíos afrontados en la etapa de implementación de otros acercamientos que fueron tratados con menor profundidad en este trabajo.

6.1. Algoritmo clásico

Para implementar el algoritmo clásico de construcción de EPAs necesitamos lo siguiente:

- Obtener la lista de métodos y de precondiciones de un contrato
- Poder definir estados de un contrato que garantizen el invariante
- Poder ejecutar los métodos y precondiciones del contrato de manera aislada
- Poder ejecutar el invariante del contrato de manera aislada
- Realizar consultas sobre el resultado de la ejecución de estos

Afortunadamente, la API de Manticore de los contratos inteligentes deployeados ofrece acceso a la lista de sus métodos de manera nativa. Sin embargo, dado que la abstracción de estos es a nivel bytecode, no resultaba posible extraer de manera automática las precondiciones. Por otro lado, como mencionamos en la sección 4.4 la API programable sólo nos permite generar estados a partir de la ejecución directa de métodos a instancias desde el constructor inicial. Estas dos limitaciones significaron que tuvimos que introducir las precondiciones como métodos explícitos a los contratos de manera manual, y que necesitamos introducir alguna forma de simular la ejecución de los métodos en el vacío.

Debido al alcance previsto para la experimentación con la herramienta, y para mantener el tiempo de desarrollo al mínimo, optamos por no automatizar la extracción de precondiciones de los métodos en métodos independientes, sino que se realizó de manera totalmente manual en los contratos con los que experimentamos. De todas formas, debido a que los contratos inteligentes incluyen sus precondiciones de manera explícita al comienzo de los métodos, esta traducción manual consistió en copiar las secciones relevantes y reemplazar las apariciones de `require` por `return`. Luego, para identificar estos métodos automáticamente desde Manticore, establecimos la convención de nomenclatura de introducirles el sufijo `_precondition`. Por ejemplo, el método que implementa la preconidición explícita del método `MakeOffer` se llamaría `MakeOffer_precondition`.

Para poder emular la ejecución de los métodos de manera aislada, era necesario poder construir dentro de Manticore estados completamente genéricos. Es decir, obtener valores para las variables de la blockchain y las variables de estado totalmente arbitrarios, que no encontraran restrictos a ser una de las posibles configuraciones resultantes de una secuencia de métodos en particular. Dado que la API de Manticore sobre los contratos no provee acceso directo a las variables o el storage de los contratos (sólo permite interacturar con los métodos y el estado de la blockchain), decidimos solucionarlo modificando los valores de las variables externamente mediante métodos. Para esto, también consideramos que lo más veloz en tiempo de desarrollo sería introducir manualmente un método externo, "`setter`", al contrato que recibe un parámetro por cada variable de estado en el contrato, y que asigna el valor recibido a esta. De esta manera, ejecutar `setter` con parámetros simbólicos genera un estado genérico del contrato.

Los estados resultantes de este proceso, sin embargo, al ser totalmente irrestrictos, no garantizaban el cumplimiento del invariante. Para reintroducir esta propiedad luego de llamar al `setter` del contrato, aprovechamos la interfaz del módulo **`smtlib`** de Manticore. Lo más sencillo fue introducir el invariante como método explícito de manera manual sólo en el código fuente de los contratos. Luego usamos este método para restringir los estados genéricos sobre su resultado, de manera análoga a como usamos las implementaciones de las precondiciones. Los invariantes a lo largo del desarrollo fueron producidos de manera manual para cada contrato, razonando artesanalmente sobre las variables de estado del contrato y como sus métodos las modificaban.

6.2. Algoritmo alternativo

Los requisitos para implementar el algoritmo alternativo son muy similares. Resulta necesario lo siguiente:

- Obtener la lista de métodos y de precondiciones de un contrato
- Poder ejecutar los métodos y precondiciones del contrato de manera aislada
- Realizar consultas sobre el resultado de la ejecución de estos
- Poder retroceder pasos en la ejecución del contrato hasta un estado anterior

De estos items, notamos que los primeros tres se comparten con el algoritmo clásico y, de hecho, fueron tratados con la mismas soluciones ya descritas. Sin embargo, la necesidad de poder realizar *rollbacks* en la ejecución del contrato analizado, a pesar de ser deseable en la implementación del algoritmo clásico, era crucial en la implementación del algoritmo alternativo. Esto es porque en este algoritmo los estados de la EPA se descubren mediante sucesiones crecientes de llamados a métodos. Si no fuera posible realizar un paso en la EPA, retroceder, y luego realizar otro, significaría que para explorar estados profundos deberíamos levantar una nueva instancia del contrato y repetir la ejecución de los métodos que nos hacen alcanzar el estado cada vez.

Por este motivo, modificamos el módulo **manticorebase** para permitir retroceder en el historial de transacciones externas del contrato. Estos cambios, junto con otras pequeñas modificaciones y correcciones de defectos en la herramienta pueden verse en el repositorio [39]. La implementación de estos dos algoritmos, por su parte, está pública en el repositorio [41].

Otro detalle implementativo que hubo que solucionar en esta etapa surge de una peculiaridad de Manticore frente a las variables de tipo `account` (que representan entidades en la blockchain, sea usuario o contrato) simbólicas. El problema surgía de que Manticore, al mantener una simulación completa de la blockchain, buscaba resolver estas variables a algún valor que se condijera con una `account` verdadera presente en la blockchain, en lugar de proveer algún otro tipo de abstracción. Esto quiso decir, considerando la frecuencia con la que los contratos comparan valores de tipo `account`[1], que los análisis deberían ser realizados con al menos dos `account` externas al contrato presentes en la blockchain. Por otro lado, debido a que en Ethereum todas las transacciones tienen una dirección de origen “`msg.sender`”, esto significa que ejecutar métodos sucesivos con valores simbólicos para `msg.sender` en Manticore hace explotar la cantidad de `state` activos de manera exponencial en la cantidad de `account` definidas en la blockchain [2].

Debido a que no vimos que produjera problemas en las EPAs generadas, la experimentación se realizó con un número de accounts externas fijo igual a 2.

6.3. Manticore como caja negra

En 2023 torres et al. utilizó VeriSol, una herramienta de bounded model checking sobre contratos Solidity, como caja negra para la construcción de EPAs de smart contracts [46] [25]. Este approach consiste en codificar las queries de cada transición de la EPA en métodos de un smart contract. Como mencionamos en la seccion 5.1, esta query puede traducirse a una consulta de alcanzabilidad de código. Por ejemplo, en un contrato con tres métodos: `A`, `B` y `C` sin parámetros, para consultar si se puede transicionar desde $\mathcal{M} = \{\texttt{A}, \texttt{B}\}$ a $\mathcal{N} = \{\texttt{A}, \texttt{C}\}$ mediante un llamado a `A`, el método construido era el siguiente:

```
1 function ABnotC_AnotBC_viaA() public returns (bool) {
2     if(A_precondition() && B_precondition() && !C_precondition()){
3         A();
4         if(A_precondition() && !B_precondition() && C_precondition()){
5             assert(false);
6         }
7     }
8 }
```

Aquí, la alcanzabilidad del `assert(false)` significaba la existencia de esta transición. Intentamos utilizar el mismo procedimiento para encontrar las transiciones en la EPA con Manticore, utilizando la herramienta por línea de comandos y la herramienta

[1] Para un ejemplo, ver las sentencias que predican sobre `msg.sender` en el contrato `SimpleMarketplace` (Fragmento de código 3.2)

[2] Asumiendo que todas las `account` son aceptadas como remitente por las precondiciones de los métodos

`manticore-verifier`. Sin embargo, la estrategia de exploración de estas dos herramientas era muy poco sofisticada, y no resultaba capaz de encontrar transiciones incluso para contratos muy simples. Además, la herramienta `manticore-verifier` no funcionaba correctamente para contratos simples, generando errores críticos en tiempo de ejecución[3].

6.4. Abstracción de los contratos por combinación de enums

Otro método que intentamos para construir las abstracciones de los contratos inteligentes fue no construir las abstracciones en base a predicados sobre las precondiciones, como las EPAs, sino en particionar los estados del contrato en base a los valores que tomaran sus variables de instancia de tipo `enum`. A menudo los contratos inteligentes están diseñados considerando una cantidad finita de configuraciones posibles, como si se abstrayeran a una máquina de estados finita particular. Estos contratos están programados haciendo uso de variables de estado de tipo `enum` que indican el estado en el que se encuentra el contrato con respecto a diversas propiedades, a menudo indicando propiedades independientes con distintas variables.

Consideremos el siguiente ejemplo, el contrato `RoomThermostat` del benchmark "Microsoft Azure Blockchain Workbench" [34] que podemos ver en el fragmento de código 6.1. Como mencionamos, hace uso de las variables de tipo `enum` para indicar los estados de ejecución en los que se encuentra el contrato. En particular, la variable `State` controla el acceso a cada uno de los métodos del contrato, mientras que la variable `Mode` indica otras propiedades relevantes del estado contrato. De los métodos disponibles, podemos ver que los métodos `SetMode` y `SetTargetTemperature` se encuentran habilitados cuando el valor de `State` es `InUse`, mientras que el método `StartThermostat` requiere que el valor de `State` sea `Created`.

```
1  pragma solidity >=0.4.25 <0.6.0;
2  contract RoomThermostat
3  {
4      //Set of States
5      enum StateType { Created, InUse}
6
7      //List of properties
8      StateType public State;
9      address public Installer;
10     address public User;
11     int public TargetTemperature;
12     enum ModeEnum {Off, Cool, Heat, Auto}
13     ModeEnum public  Mode;
14
15     constructor(address thermostatInstaller, address thermostatUser) public
16     {
17         Installer = thermostatInstaller;
18         User = thermostatUser;
19         TargetTemperature = 70;
20     }
21
```

[3] Algunos de estos defectos se encuentran corregidos como parte de los cambios que introdujimos en la herramienta [39]

```
22     function StartThermostat() public
23     {
24         if (Installer != msg.sender || State != StateType.Created)
25         {
26             revert();
27         }
28 
29         State = StateType.InUse;
30     }
31 
32     function SetTargetTemperature(int targetTemperature) public
33     {
34         if (User != msg.sender || State != StateType.InUse)
35         {
36             revert();
37         }
38         TargetTemperature = targetTemperature;
39     }
40 
41     function SetMode(ModeEnum mode) public
42     {
43         if (User != msg.sender || State != StateType.InUse)
44         {
45             revert();
46         }
47         Mode = mode;
48     }
49 }
```

Listing 6.1: Contrato Inteligente `RoomThermostat` en Solidity

Siguiendo esta idea, consideramos generar abstracciones donde los estados están definidos por las posibles combinaciones de las variables de estado de tipo `enum`, confiando en cada distinta combinación de estas variables se corresponde con un estado de ejecución único del contrato interesante. En estas abstracciones consideramos que las transiciones sean por los métodos externos del contrato, al igual que en las EPAs.

Continuando el ejemplo del contrato `RoomThermostat`, en la figura 6.1 podemos ver la abstracción resultante. Si prestamos atención a la variable `State`, vemos que la máquina de estados se divide en dos secciones: Por un lado los estados etiquetados como `InUse` presentan transiciones por los métodos `SetMode` y `SetTargetTemperature`. Por otro lado el único estado etiquetado como `Created` tiene una única transición, que ocurre por el método `SetMode`. Además, el conjunto de estados etiquetados con `InUse` están particionados por la variable `Mode`, pudiendo transicionar entre cualquiera de los estados al ejecutar el método `SetMode`. En cambio ejecutar el método `SetTargetTemperature`, que solo afecta variables internas que no son de tipo `enum`, siempre resulta en una transición al mismo estado desde el que se lo ejecutó.

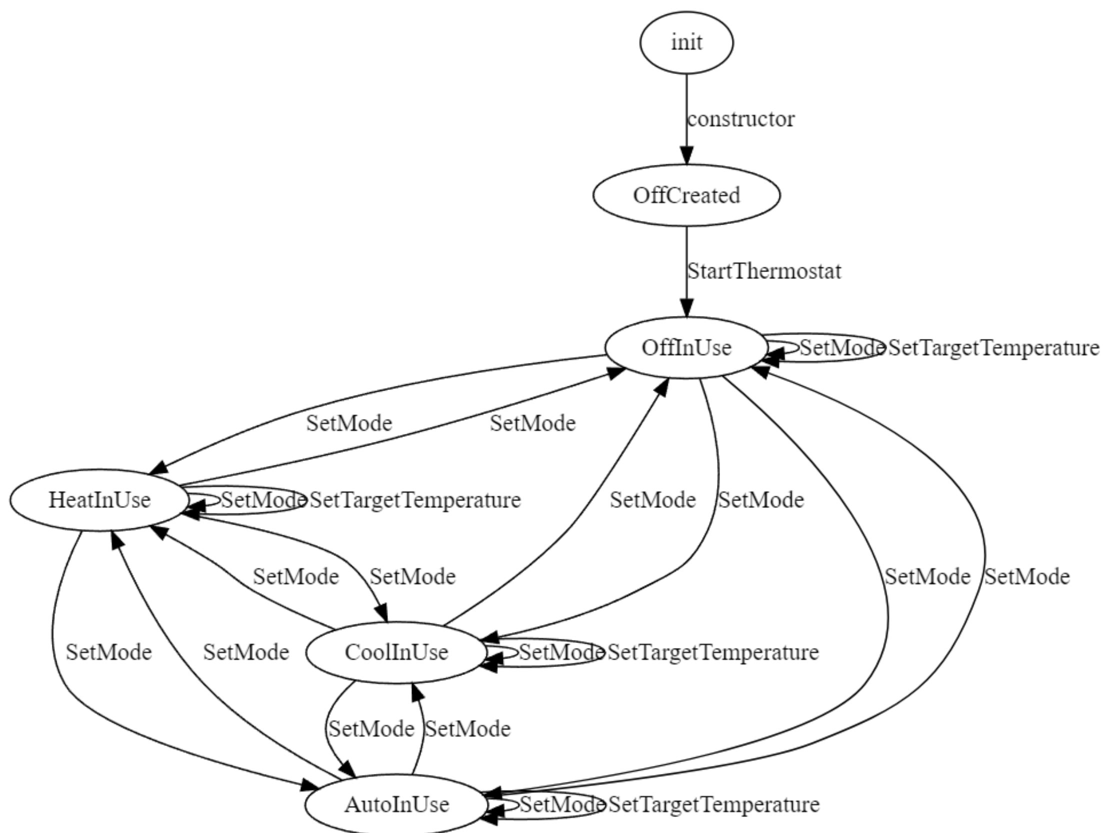


Fig. 6.1: Abstracción por estado de las variables `enum` del contrato `RoomThermostat`

A la hora de implementar la generación de estas abstracciones, para poder utilizar el mismo mecanismo de generación de abstracciones que el utilizado para las EPAs era necesario conocer el valor de las variables de instancia de tipo `enum`. Aquí, nuevamente por consideraciones en el tiempo de desarrollo, lo mas sencillo fue tratar la introspección de estas variables de manera similar a los valores de las precondiciones de los métodos. Lo que hicimos fue definir métodos externos para cada una de las variables de instancia de tipo `enum` que permitieran observar el valor de esta variable. Luego, podíamos detectar estos métodos introduciendo un nuevo tipo sufijo "`_enum`". Por ejemplo, para la variable `Mode` en el contrato `RoomThermostat`, el método generado era el siguiente:

```
1    function Mode_enum() public returns (ModeEnum) {
2        return Mode;
3    }
```

Utilizando estos nuevos métodos externos el mecanismo para generar las abstracciones era el mismo que el de las EPAs, con la diferencia de que los estados se construían tomando combinaciones de los `enum` en lugar de de las precondiciones.

7. ANÁLISIS

En esta sección realizamos algunos análisis sobre el comportamiento y performance de los prototipos implementados. Primero realizamos una evaluación manual de la calidad de las abstracciones generadas por el algoritmo clásico y el alternativo. Luego, evaluamos el tiempo empleado en promedio de cada algoritmo sobre un benchmark conocido y por último realizamos un análisis sobre cuáles son las partes más costosas en tiempo del prototipo implementado.

Los análisis fueron realizados sobre el benchmark *Microsoft Azure Blockchain Workbench* [34], un benchmark conocido para el cual ya contabamos con las EPAs correctas debido a que fue utilizado en [20] [46]. Además, realizamos algunos estudios sobre el contrato `BoundedStack` presentado en la sección 5.1. Los contratos analizados fueron los siguientes:

Contrato	**Lineas de codigo**	**Cantidad de estados de la EPA**
DefectiveComponentCounter	33	2
SimpleMarketplace	66	4
BasicProvenance	48	3
RoomThermostat	48	3
BoundedStack	28	5

Los experimentos se efectuaron en una máquina con un procesador Intel Core i7-4770 3,40GHz x 8, 16 GB RAM, Mesa Intel HD Graphics4600 (HSW GT2) y almacenamiento SSD 256 GB, ejecutando Ubuntu 22.04.3 LTS.

7.1. Comparación de las abstracciones generadas entre el algoritmo clásico y el alternativo

Como mencionamos en la sección 5.3, las abstracciones generadas por el algoritmo alternativo no siempre son Enabledness Preserving Abstractions[1] de los contratos analizados, sino que son unas subaproximaciones de estas. Al mismo tiempo, las EPAs son sobreaproximaciones del comportamiento de los contratos, por lo que el mecanismo resultante del algoritmo alternativo yace en algún lugar intermedio: no es ni sound ni complete. Por esto, nos interesa explorar en particular cuáles son las diferencias producidas entre el algoritmo alternativo y las generadas por el algoritmo clásico (las EPAs) para evaluar su utilidad. Para responder estas preguntas analizamos el desempeño de los prototipos implementados usando ambos algoritmos sobre algunos casos de prueba.

[1] Es decir, en algunos casos no cumplen la definición presentada en la sección 3.43.

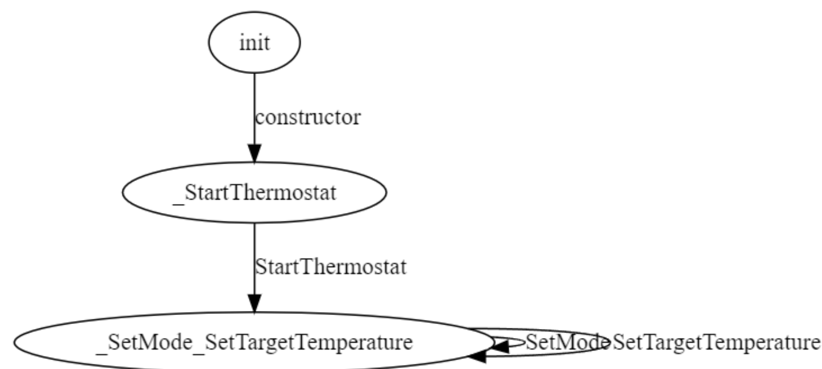


Fig. 7.1: Enabledness Preserving Abstraction del contrato `RoomThermostat`

7.1.1. Caso `RoomThermostat`

El primer caso que evaluamos fue el contrato `RoomThermostat`, cuyo código fuente podemos ver en el fragmento de código 6.1. Es un contrato pequeño, con pocos métodos y un invariante sencillo. Para poder emplear el algoritmo clásico sobre este contrato, el invariante propuesto fue el siguiente:

```
1   function invariant(StateType stateNew, address installerNew, address
        userNew, int targetTemperatureNew, ModeEnum modeNew) public returns(
        bool){
2       bool result = (stateNew == StateType.Created || stateNew == StateType.
            InUse);
3       result = result && (modeNew == ModeEnum.Auto || modeNew == ModeEnum.
            Cool || modeNew == ModeEnum.Heat || modeNew == ModeEnum.Off);
4       if(stateNew == StateType.Created){
5           result = (targetTemperatureNew == 70) && (modeNew == ModeEnum.Off);
6       }
7       return result;
8   }
```

Luego, evaluamos tanto el algoritmo clásico como el alternativo sobre el contrato. En ambos casos la EPA generada fue la misma, una máquina de estados sencilla de tan sólo tres estados que podemos ver en la figura 7.1.

7.1.2. Caso `BoundedStack`

El siguiente caso que evaluamos fue el ejemplo teórico que expusimos en la sección 5.3, el contrato `BoundedStack`, con intención de evidenciar las subaproximaciones realizadas por el algoritmo nuevo. El código del contrato se encuentra en el fragmento de código 5.1. Nos interesa corroborar el comportamiento de ambos algoritmos, si se produce alguna subaproximación, y cuáles.

Lo primero que vimos al evaluar la abstracción generada por el algoritmo alternativo sobre el contrato `BoundedStack` lo podemos ver en la imagen 7.2. En esa abstracción podemos ver que faltan algunas de las transiciones presentes en la EPA del contrato, pero

además podemos observar que las transiciones por pop siempre llevan al estado desde el que se ejecutó el método. Cuando realizamos esta comparación, esta observación generó bastante confusión, hasta que comprendimos que se debía a un defecto en el código fuente de `BoundedStack` que no decrementaba la variable `size` al ejecutar pop. Luego de corregir el defecto, la abstracción generada por el prototipo fue la referenciada en la imagen 5.1b. Es decir, la misma que elaboramos durante el ejemplo teórico del comportamiento del algoritmo alternativo, y la que evidenciaba la falencia del algoritmo alternativo para este ejemplo. Decidimos incluir este pequeña caso de un error en el código fuente en lugar de omitirlo para ejemplificar que, a pesar de que quede evidenciado que el algoritmo nuevo genere subaproximaciones de las EPAs en la práctica, las abstracciones generadas pueden seguir resultando útiles a la hora de atrapar errores.

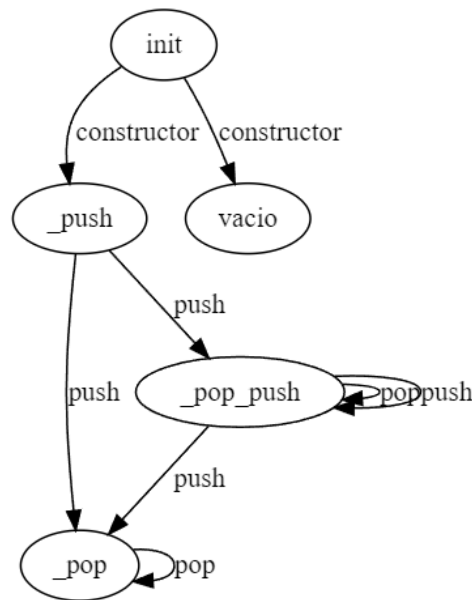


Fig. 7.2: Enabledness Preserving Abstraction del contrato `BoundedStack` con un error que no decrementa el tamaño luego de ejecutar pop

A la hora de emplear el algoritmo clásico sobre el contrato, el invariante propuesto fue el siguiente:

```
1    function invariant() public returns(bool){
2        bool result = (size >= 0) && (size <= maxSize);
3        result = result && (internal_arr.length == size);
4        return result;
5    }
```

Sin embargo, el prototipo desarrollado con el algoritmo clásico no pudo analizar este contrato satisfactoriamente, sino que terminó el analisis por time out luego de 24 horas. Esto se debe a que el paso de restringir las instancias del contrato a aquellas que satisfagan el invariante resultó demasiado difícil para el motor de ejecucion simbólica de Manticore. Algunos experimentos y análisis intermedios nos hacen creer que se debe a la pobre representación de las variables de tipo `uint256[]` del mismo. Esto significó que termina-

mos el análisis comparando la abstracción generada por el algoritmo alternativo no con la generada por el algoritmo clásico, sino con una EPA producida manualmente.

7.2. Comparación en tiempo de ejecución entre el algoritmo clásico y el alternativo

Otra hipótesis generada durante el desarrollo del algoritmo alternativo fue que el evitar la ejecución de los invariantes mejoraría el tiempo de ejecución. Esto se debe a que los invariantes, comparados con las precondiciones de los métodos externos, son propiedades relativamente complejas que además pueden hablar sobre el estado de variables internas de tipos complejos como arrays, mappings, etc. Para evaluar esta hipótesis estudiamos el tiempo de ejecución empleado por los prototipos que implementan el algoritmo alternativo y el clásico en Manticore.

Ya que el benchmark que utilizamos había sido utilizado por Godoy et al. en 2022 [20], a la hora de experimentar contamos con EPAs de todos los contratos involucrados, por lo que además pudimos usarlas para corroborar la correctitud de las abstracciones generadas. Los experimentos fueron realizados ejecutando ambos prototipos sobre cada contrato cinco veces y luego tomando el promedio del tiempo total de ejecución.

En la figura 7.3 podemos ver los resultados del experimento realizado. Las mediciones fueron solo realizadas para los contratos `DefectiveComponentCounter`, `RoomThermostat`, `BasicProvenance` y `SimpleMarketplace` porque como podemos ver el tiempo de ejecución de la implementación del algoritmo clásico nunca fue menor a diez horas. A pesar de que en todos los contratos analizados podamos ver una diferencia significativa al emplear el algoritmo alternativo, es importante destacar que los tiempos de ejecución de esta técnica aún se mantuvieron muy altos. La mejora en tiempo parecería ser relativamente constante a lo largo de los cuatro contratos analizados, habiendo tardado el algoritmo alternativo alrededor de diez horas menos que el clásico. Esta diferencia cercana a constante posiblemente se deba a que todos los contratos eran muy simples, por lo que no se pudo observar la diferencia en comportamiento que habría en contratos más complejos.

A pesar de que la mejora del algoritmo alternativo frente al clásico fuese tan grande, los valores obtenidos con el algoritmo alternativo siguieron sin resultar razonables. Si observamos los valores obtenidos, el tiempo de cómputo en promedio fue de entre media hasta casi ocho horas, lo que no resulta aplicable para un usuario de la herramienta que pretende generar las abstracciones en tiempo real. Por este motivo decidimos realizar los experimentos que mostraremos a continuación, en los que buscamos conseguir mejoras en el tiempo de ejecución o claridad en la razón de que tarde tanto.

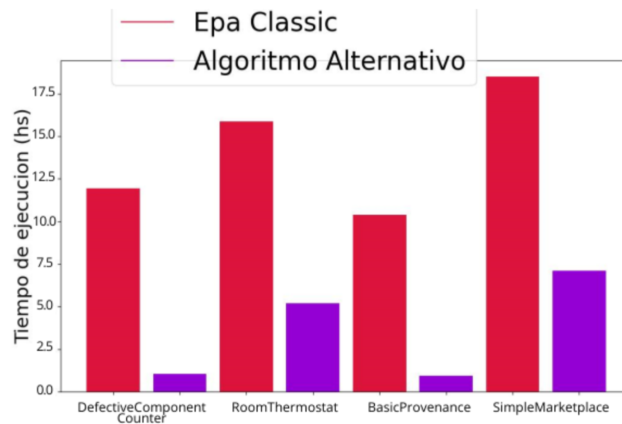


Fig. 7.3: Tiempo promediado para 5 ejecuciones (en horas) de la generación de abstracciones del prototipo que implementa el algoritmo alternativo y el algoritmo clásico

7.3. Evaluación más detallada del tiempo de ejecución del algoritmo alternativo

Para mejorar el tiempo de ejecución del prototipo buscamos identificar en qué partes del análisis era que se consumía la mayor parte del tiempo empleado. Lo primero que hicimos fue identificar "etapas" en el algoritmo alternativo que pudimos diferenciar para medir el tiempo empleado en cada una de ellas. Por otro lado analizamos el código fuente de Manticore y diferenciamos en distintos niveles de abstracción las funcionalidades que estábamos usando.

La separación de la ejecución del algoritmo en etapas la hicimos en estas dos:

1. La ejecución simbólica de los métodos [2]
2. La solución de queries de satisfacibilidad sobre las transiciones en la EPA

Refiriéndonos al algoritmo alternativo, podemos ver que este alterna entre estas dos etapas. Nos resulta de interés ver cuál, si alguna, consume la mayor parte del tiempo de ejecución.

A continuación indicamos cuál fue la división en niveles de abstracción que hicimos de las funcionalidades de Manticore. En particular, por como es la arquitectura de multithreading y `workers` de Manticore, solo pudimos hacer esta distinción en los métodos relacionados a la resolución de valores simbólicos. La manera en la que se realizaba la ejecución simbólica y la generación de las condiciones de camino era demasiado dependiente de objetos creados dinámicamente como para poder ser analizada de la misma manera.

1. **Nivel Externo**[3] : utilizado para medir el tiempo que consumían las funciones de

[2] Al desplegado simbólico del contrato lo consideramos parte de la ejecución simbólica de los métodos.

[3] En el momento de realizar las mediciones, los valores para este nivel se registraron bajo el nombre de "Nivel 3"

la API de manticore que nuestro prototipo llamaba directamente. La única función cuyo tiempo medimos en este nivel fue `generateTestCases`.

2. **Nivel `state`** : Utilizado en las funciones del módulo **`state`** que identificamos que eran llamadas. Los métodos registrados bajo este nivel fueron `can_be_true` y `solve_one_n_batched`.
3. **Nivel SMT solver** : Utilizado alrededor de cada llamado directo al SMT solver en los métodos del módulo **`smtlib`**. Los métodos registrados bajo este nivel fueron `_is_sat`, `_get_value` y `__get_value_all`.

Cada uno de los métodos fue modificado para medir el tiempo empleado entre el principio y el fin de la zona delimitada de interés. Por ejemplo, en el fragmento de cógdio 7.1 podemos ver los cambios introducidos en el método `__get_value_all` del **Nivel SMT solver** para registrar el tiempo.

```
1  def __getvalue_all(self, expressions_str: List[str], is_bv: List[bool]) -> Dict
       [str, int]:
2       start = time.time()
3      all_expressions_str = " ".join(expressions_str)
4      self._smtlib.send(f"(get-value ({all_expressions_str}))")
5      ret_solver: Optional[str] = self._smtlib.recv()
6       print(f"(level _getvalue_all_z3_call) took {time.time()- start} seconds")
7      assert ret_solver is not None
8      return_values = re.findall(RE_GET_EXPR_VALUE_ALL, ret_solver)
9
10     return {value[0]: _convert(value[1]) for value in return_values}
```

Listing 7.1: Método `__get_value_all` del modulo **`smtlib`** modificado para registrar el tiempo empleado por la llamada al SMT solver. El código resaltado indica las líneas agregadas para registrar el tiempo.

De entre los métodos a los que les registramos el tiempo de ejecución, sabíamos que los métodos en **Nivel Externo** solo eran ejecutados cuando el prototipo implementado los llamaba explícitamente. Por otro lado, los métodos en **Nivel `state`** eran métodos que a veces eran llamados desde la implementación directamente, pero además eran ejecutados como métodos internos de otros procesos durante el análisis. Por último, los métodos del **Nivel SMT solver** eran ejecutados en diversos momentos del análisis.

Una vez hecha esta separación en etapas del algoritmo y niveles de abstracción, medimos el tiempo total en promedio empleado por cada etapa y cada nivel de abstracción. Por otro lado, además, calculamos para cada uno de los niveles de abstracción la cantidad de veces que este mismo fue registrado y el tiempo total empleado. Esto fue para intentar percibir discrepancias entre los niveles que esperábamos que consumieran la mayoría del tiempo y lo que pudiéramos medir.

En la figura 7.4 podemos ver las mediciones obtenidas sobre el tiempo empleado por las dos etapas del algoritmo. Se ve que la gran mayoría del tiempo empleado durante la generación de EPAs con el algoritmo involucrado fue consumido por la ejecución simbólica de los métodos. La resolución de predicados para consultar por la presencia de transiciones en la EPA, en cambio, fue comparativamente pequeña. Por ejemplo, para el contrato

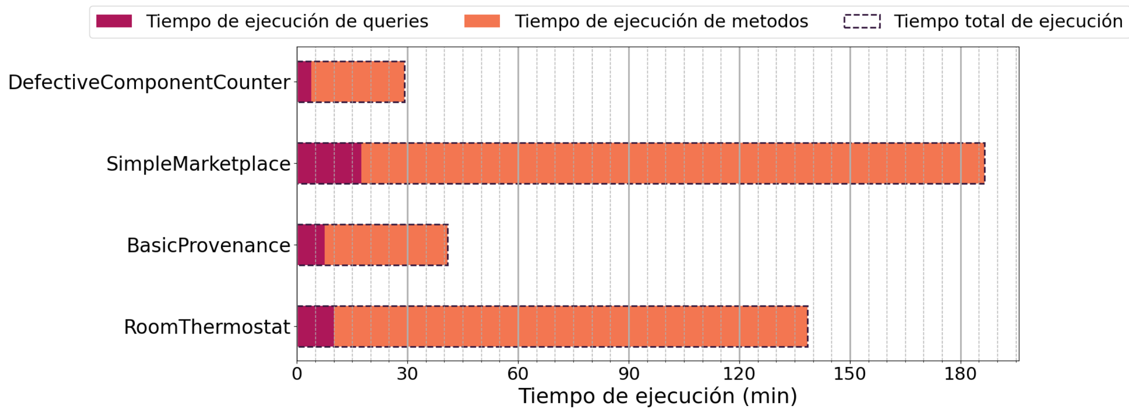


Fig. 7.4: Proporción del tiempo de ejecución promediado entre 5 (en horas) del algoritmo alternativo. En naranja el tiempo empleado por la ejecución simbólica de los métodos, en violeta por la resolución de queries de predicados y en líneas punteadas el tiempo total (medido) empleado por el algoritmo.

`BasicProvenance`, podemos ver que a pesar que el promedio del tiempo total fue cercano a los 40 minutos, la etapa de resolución de queries demoró tan sólo valores cercanos a los 7 minutos. Por otro lado, la suma del tiempo empleado en promedio por ambas etapas resulta a simple vista idéntico al tiempo total promedio que se pudo medir, por lo que tampoco observamos un consumo inusualmente alto de tiempo por algún otro aspecto del análisis que no hayamos considerado.

En las tablas 7.1 a 7.4 podemos ver los resultados obtenidos en las mediciones del tiempo empleado en cada nivel de abstracción de los métodos de Manticore para los contratos. Podemos ver en la tabla 7.4 los valores obtenidos para el contrato `BasicProvenance`, en el que nos queremos enfocar primero, para señalar las relaciones entre los valores para algunos de los métodos. Haciendo memoria sobre el análisis de la figura 7.4, podemos ver que los aproximadamente 450 segundos empleados por el método `generateTestCases`, que es el más externo de todos, representan casi la totalidad (7.5 minutos) del tiempo empleado por la etapa de solución de queries del programa. De hecho, los métodos `solve_one_n_batched` y `get_value_in_batch`, que pertenecen a un nivel de abstracción menor, siguen siendo responsables por la mayoría del tiempo empleado (y si observamos el número de veces que cada uno fue llamado, a fines del análisis que estamos haciendo ambos métodos resultan prácticamente iguales). Sin embargo, al realizar el salto a los métodos de **Nivel SMT solver**, vemos que comparativamente estos casi ni aportaron al tiempo empleado. Los tres métodos `_getvalue_all`, `_is_sat` y `_getvalue`, todos llamados de manera directa por los métodos del nivel superior, apenas aportan 5 segundos de los 7.5 minutos involucrados.

Por otro lado, notamos que esta relación entre los métodos de los tres niveles de abstracción se preserva para los otros contratos (tablas 7.1, 7.2 y 7.3) también: los métodos `generateTestCases`, `solve_one_n_batched` y `get_value_in_batch` consumen casi la totalidad del tiempo empleado mientras que los metodos del **Nivel SMT Solver** no consumen casi nada. De aquí podemos intentar obtener dos conclusiones. O bien las llamadas al SMT solver funcionan de una manera asincrónica, que no pudimos capturar en nuestras mediciones, o bien en efecto la mayoría del tiempo consumido por consultas a

Manticore es empleado no en tiempo de procesamiento del SMT solver, sino en alguna etapa de la aplicación misma (que no fuimos capaces de medir). Considerando que la mayor parte del tiempo de ejecución de la aplicación fue consumido por la etapa de ejecución simbólica, y no por la de resolución de queries, es posible que el preprocesamiento de las condiciones simbólicas de Manticore, construido durante la etapa de ejecución simbólica, sea extraordinariamente complejo y por ende caro en tiempo de ejecución. Además, un rudimentario y primer análisis del código involucrado sugiere que los llamados realizados al SMT solver son completamente sincrónicos, aunque no se pudo realizar un experimento para corroborar ninguna de estas dos hipótesis.

Método de Manticore	Número de llamados	Tiempo empleado (s)
`_getvalue`	4	0.00188
`_is_sat`	80	0.48341
`state.can_be_true`	21	0.21036
`_getvalue_all`	30	1.62389
`get_value_in_batch`	54	230.79537
`solve_one_n_batched`	54	231.65494
`generateTestCases`	3	231.87251

Tab. 7.1: Tiempo empleado por los métodos de Manticore para el contrato `DefectiveComponentCounter`.

Método de Manticore	Número de llamados	Tiempo empleado (s)
`_getvalue`	110	2.16616
`_is_sat`	370	9.89551
`state.can_be_true`	61	0.63042
`_getvalue_all`	55	4.25464
`get_value_in_batch`	84	1028.78724
`solve_one_n_batched`	84	1029.55111
`generateTestCases`	4	1030.08048

Tab. 7.2: Tiempo empleado por los métodos de Manticore para el contrato `SimpleMarketplace`.

Método de Manticore	Número de llamados	Tiempo empleado (s)
`_getvalue`	249	16.3826
`_is_sat`	684	62.46842
`state.can_be_true`	89	2.59814
`_getvalue_all`	59	3.97965
`get_value_in_batch`	88	583.78606
`solve_one_n_batched`	88	584.46269
`generateTestCases`	4	584.95501

Tab. 7.3: Tiempo empleado por los métodos de Manticore para el contrato `RoomThermostat`.

Método de Manticore	Número de llamados	Tiempo empleado (s)
`_getvalue`	62	0.4647
`_is_sat`	196	2.24389
`state.can_be_true`	21	0.19231
`_getvalue_all`	31	2.33352
`get_value_in_batch`	54	449.60244
`solve_one_n_batched`	54	450.04921
`generateTestCases`	3	450.27618

Tab. 7.4: Tiempo empleado por los métodos de Manticore para el contrato `BasicProvenance`.

Tiempo empleado por los métodos de Manticore involucrados en las queries por transiciones en la EPA para los distintos contratos. En el color más oscuro los métodos del **Nivel Externo**, en el color intermedio los del **Nivel `state`** y en el más claro los del **Nivel SMT solver**

8. CONCLUSIONES

En este trabajo realizamos una apreciación de las funcionalidades de Manticore, al mismo tiempo que un desglose de su arquitectura interna y su API programable. Dicho análisis fue relevante a la hora de poder idear una estrategia para utilizar Manticore en la construcción de EPAs, y en la indagación de los tiempos de ejecución de la herramienta.

Desarrollamos un algoritmo para la construcción de EPAs en el caso de tener acceso a las precondiciones de los métodos pero no al invariante, orientado al uso de ejecución simbólica y en particular, Manticore. Vimos las limitaciones de este nuevo algoritmo frente a cierto tipo de contratos, y pudimos corroborar tanto la facilidad con la que se lo puede hacer fallar como corroborar su funcionamiento en benchmarks de smart contracts utilizados anteriormente.

Pudimos contrastar la performance en tiempo del algoritmo novedoso contra el algoritmo de construcción de EPAs clásico para ejemplos simples, y obtuvimos que el algoritmo novedoso presentaba grandes mejoras sobre el clásico al implementar ambos con Manticore, aunque el tiempo total de ejecución de este algoritmo seguía sin resultar satisfactorio.

Al buscar los principales culpables de la alta cantidad de tiempo empleado por el prototipo implementado, en busca de optimizaciones, identificamos que la etapa responsable de la mayoría del tiempo de ejecución era la de ejecución simbólica de métodos de los contratos. Esta etapa asimismo resultaba la más difícil de analizar (y por ende de optimizar), debido a los patrones de asignación dinámica y multithreading que empleaba. Por otro lado, realizamos un análisis evaluando si la mayoría del tiempo era consumido por Manticore mismo o por los programas de SMT solving, y obtuvimos resultados que sugieren que este fuera consumido por los procesos de Manticore.

Consideramos que áun existen algunas líneas que se pueden explorar en términos de optimización en tiempo del prototipo implementado. Por un lado, podría indagarse más profundo en el comportamiento de Manticore, intentando observar cuáles son las consultas que realiza a los programas de SMT solving. Por otro lado, explorar el impacto que tiene la explosión de estados producida por la presencia de múltiples `accounts` durante los análisis en el tiempo de ejecución de Manticore puede proveer un mayor entendimiento, además de ser un punto de partida para explorar técnicas que supriman esta misma explosión.

Creemos que una línea de investigación futura es el uso de los callbacks de la API de manticore para realizar un análisis en el que se avanza de a un único `state` por vez, más similar a las estrategias clásicas de construcción de EPAs. Por otro lado, es posible que la capacidad de emular blockchains enteras de Manticore le permita encontrar propiedades en las abstracciones que le resulten más difíciles a otras herramienta que analizan un único contrato. Creemos que analizar esta hipótesis podría ser valioso para evaluar plenamente la estrategia presentada en este trabajo.

8.1. Trabajo relacionado

Desde hace un tiempo se emplea abstracción por predicados, similares a las de este trabajo, para realizar verificación de programas [22]. Cuando el objetivo es verificación, el nivel de abstracción, el tamaño del modelo resultante y la facilidad de vincular las abstracciones generadas con el programa original varían. Abstraer al llamado de función permite generar abstracciones mucho mas amenas para la validación, aunque agrupar tanto el comportamiento las hace menos útiles para la verificación.

La generación de EPAs es una técnica similar a la generación de typestates [13] de un programa. Estas abstracciones pensadas para la verificación son menos permisivas que las EPAs, y sólo permiten trazas válidas en el artefacto original. Otra técnica similar es la inferencia de interfaces [21]. Existen técnicas para inferir modelos del comportamiento a partir de trazas de ejecución, infiriendo especificaciones para sucesivamente generar nuevos casos de test [19].

En el contexto de contratos inteligentes, existen múltiples herramientas ya mencionadas dedicadas a encontrar vulnerabilidades de seguridad [31] [4] [5] [37] [24] o verificación de propiedades [18] [14] [25]. Otros acercamientos [33] construyen contratos inteligentes correctos a partir de especificaciones. Con respecto a herramientas orientadas a la validación, existen visualizadores [44] [16] de grafos de control de flujo y herencia de contratos en Solidity, pero que no visualizan funcionalidad o comportamiento como lo hacen las abstracciones de este trabajo.

En 2013 de Caso et al. [11] construyen EPAs para programas escritos en `C` haciendo uso de Blast [3], un model checker, como back end al estilo "caja negra". Entre las alternativas evaluadas, mencionan la posibilidad de construir EPAs mediante herramientas de generación de casos de test o mediante ejecución simbólica, y predicen algunas de las dificultades enfrentadas en este trabajo para mantener consistentes las sobreaproximaciones producidas al emplear ejecución simbólica. En 2016 Lera Romero et al. [29] generan EPAs de programas escritos en `.NET` utilizando Corral [26] como back end, un model checker desarrollado por Microsoft Research. Corral es capaz de analizar código Boogie [27], por lo que esta construcción de EPAs requería la previa traducción de `.NET` a Boogie mediante la herramienta BytecodeTranslator [40]. En 2017 Palladino et al. [38] construyen un traductor de Java a Boogie para utilizar Corral como back end de generación de EPAs. En 2023 Torres et al. [46] construyen EPAs para Solidity utilizando VeriSol, que traduce los contratos en código fuente de Solidity a Boogie, aplicando también el model checker Corral sobre el código Boogie generado.

En este trabajo presentamos la primer implementación de construcción de EPAs mediante una herramienta de ejecución simbólica, y expandimos la técnica desarrollada de manera general para cualquier herramienta de ejecucion simbólica.

BIBLIOGRAFíA

[1] *Algorand Virtual Machine.* URL: https://developer.algorand.org/docs/get-details/dapps/avm/.

[2] Roberto Baldoni y col. «A Survey of Symbolic Execution Techniques». En: *ACM Comput. Surv.* 51.3 (mayo de 2018). ISSN: 0360-0300. DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[3] Dirk Beyer y col. «The Software Model Checker BLAST: Applications to Software Engineering». En: *International Journal of Software Tools and Technology Transfer* 9 (oct. de 2007), págs. 505-525. DOI: 10.1007/s10009-007-0044-z.

[4] Trail of Bits. *Echidna: A Fast Smart Contract Fuzzer.* URL: https://github.com/crytic/echidna.

[5] Trail of Bits. *Manticore: Symbolic Execution for Humans.* URL: https://github.com/trailofbits/manticore.

[6] Vera Bogdanich Espina. «Ethereum smart contracts verification: a survey and a prototype tool.» En: *Tesis de Grado* (2019). URL: https://hdl.handle.net/20.500.12110/seminario_nCOM000440_Bogdanich.

[7] V Buterin. «Combining GHOST and Casper». En: (mayo de 2020). DOI: 10.48550/arXiv.2003.03052. URL: https://arxiv.org/abs/2003.03052.

[8] V. Buterin. *Critical update re: Dao vulnerability.* 2016. URL: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability.

[9] V. Buterin. *White Paper: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* 2014.

[10] Cristian Cadar y Koushik Sen. «Symbolic execution for software testing: three decades later». En: *Commun. ACM* 56.2 (feb. de 2013), págs. 82-90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[11] Guido De Caso y col. «Enabledness-Based Program Abstractions for Behavior Validation». En: *ACM Trans. Softw. Eng. Methodol.* 22.3 (jun. de 2013). ISSN: 1049-331X. DOI: 10.1145/2491509.2491519. URL: https://doi.org/10.1145/2491509.2491519.

[12] *Conjunto de instrucciones de la EVM.* https://github.com/wolflo/evm-opcodes. Accedido: 2023–30-10.

[13] Robert DeLine y Manuel Fähndrich. «Enforcing high-level protocols in low-level software». En: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation.* PLDI '01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, págs. 59-69. ISBN: 1581134142. DOI: 10.1145/378795.378811. URL: https://doi.org/10.1145/378795.378811.

[14] Wang Duo, Huang Xin y Ma Xiaofeng. «Formal Analysis of Smart Contract Based on Colored Petri Nets». En: *IEEE Intelligent Systems* 35.3 (2020), págs. 19-30. DOI: 10.1109/MIS.2020.2977594.

[15] Ethereum. *What was The Merge?* 2024. URL: https://ethereum.org/en/roadmap/merge/.

[16] Josselin Feist, Gustavo Grieco y Alex Groce. «Slither: A Static Analysis Framework For Smart Contracts». En: *CoRR* abs/1908.09878 (2019). arXiv: 1908.09878. URL: http://arxiv.org/abs/1908.09878.

[17] Juan Garay, Aggelos Kiayias y Nikos Leonardos. «The Bitcoin Backbone Protocol: Analysis and Applications». En: *Advances in Cryptology - EUROCRYPT 2015*. Ed. por Elisabeth Oswald y Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, págs. 281-310. ISBN: 978-3-662-46803-6.

[18] Ikram Garfatta y col. «Model checking of vulnerabilities in smart contracts: a solidity-to-CPN approach». En: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, págs. 316-325. ISBN: 9781450387132. DOI: 10.1145/3477314.3507309. URL: https://doi.org/10.1145/3477314.3507309.

[19] Carlo Ghezzi, Andrea Mocci y Mattia Monga. «Synthesizing intensional behavior models by graph transformation». En: *2009 IEEE 31st International Conference on Software Engineering*. 2009, págs. 430-440. DOI: 10.1109/ICSE.2009.5070542.

[20] Javier Godoy y col. «Predicate Abstractions for Smart Contract Validation». En: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, págs. 289-299. ISBN: 9781450394666. DOI: 10.1145/3550355.3552462. URL: https://doi.org/10.1145/3550355.3552462.

[21] Thomas A. Henzinger, Ranjit Jhala y Rupak Majumdar. «Permissive interfaces». En: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: Association for Computing Machinery, 2005, págs. 31-40. ISBN: 1595930140. DOI: 10.1145/1081706.1081713. URL: https://doi.org/10.1145/1081706.1081713.

[22] Ranjit Jhala, Andreas Podelski y Andrey Rybalchenko. «Predicate Abstraction for Program Verification». En: *Handbook of Model Checking*. Ed. por Edmund M. Clarke y col. Cham: Springer International Publishing, 2018, págs. 447-491. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_15. URL: https://doi.org/10.1007/978-3-319-10575-8_15.

[23] Aggelos Kiayias y col. «Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol». En: jul. de 2017, págs. 357-388. ISBN: 978-3-319-63687-0. DOI: 10.1007/978-3-319-63688-7_12.

[24] Johannes Krupp y Christian Rossow. «TEETHER: gnawing at ethereum to automatically exploit smart contracts». En: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, págs. 1317-1333. ISBN: 9781931971461.

[25] Shuvendu K. Lahiri y col. «Formal Specification and Verification of Smart Contracts for Azure Blockchain». En: *CoRR* abs/1812.08829 (2018). arXiv: 1812.08829. URL: http://arxiv.org/abs/1812.08829.

[26] Akash Lal, Shaz Qadeer y Shuvendu Lahiri. «Corral: A Solver for Reachability Modulo Theories». En: jul. de 2012. ISBN: 978-3-642-31423-0. DOI: 10.1007/978-3-642-31424-7_32.

[27] K Rustan M Leino. «This is Boogie 2». En: (ene. de 2008).

[28] *Lenguaje Solidity*. URL: https://docs.soliditylang.org/en/latest/.

[29] Leandro Lera Romero. «A Corralando EPAs : acercando el modelo mental al computacional». En: *Tesis de Grado* (2016). URL: http://hdl.handle.net/20.500.12110/seminario_nCOM000495_LeraRomero.

[30] Xiaoqi Li y col. «A survey on the security of blockchain systems». En: *Future Generation Computer Systems* 107 (2020), págs. 841-853. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.020. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17318332.

[31] Loi Luu y col. «Making Smart Contracts Smarter». En: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, págs. 254-269. ISBN: 9781450341394. DOI: 10.1145/2976749.2978309. URL: https://doi.org/10.1145/2976749.2978309.

[32] *ManticoreUI*. URL: https://github.com/trailofbits/ManticoreUI.

[33] Anastasia Mavridou y col. «VeriSolid: Correct-by-Design Smart Contracts for Ethereum». En: *CoRR* abs/1901.01292 (2019). arXiv: 1901.01292. URL: http://arxiv.org/abs/1901.01292.

[34] *Microsoft Azure Blockchain Workbench*. URL: https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.

[35] *Mythril: A security analysis tool for EVM bytecode*. URL: https://github.com/Consensys/mythril.

[36] OpenZeppelin. *OpenZeppelin: Build secure smart contracts in solidity*. URL: https://www.openzeppelin.com/contracts.

[37] *Pakala: A tool to search for exploitable bugs in Ethereum smart contracts*. URL: https://github.com/palkeo/pakala.

[38] Patricio Palladino. «ContractorJ: validando el comportamiento de clases de Java». En: *Tesis de Grado* (2017).

[39] *Repositorio con los cambios realizados sobre Manticore*. URL: https://github.com/DaniWppner/Manticore-tiny-changes.

[40] *Repositorio de BytecodeTranslator*. URL: https://github.com/boogie-org/bytecodetranslator.

[41] *Repositorio del prototipo implementado*. URL: https://github.com/DaniWppner/manticore-predicate-abstraction.

[42] Microsoft Research. *The Z3 Theorem Prover*. URL: https://smt-lib.org/index.shtml.

[43] *SMT-LIB 2: The Satisfiability Modulo Theories Library*. URL: https://smt-lib.org/index.shtml.

[44] *Surya, The Sun God: A Solidity Inspector*. URL: https://github.com/ConsenSys/surya.

[45] *The Yices SMT Solver*. URL: https://yices.csl.sri.com/.

[46] Edén Torres. «Generador de abstracciones para smart contracts.» En: *Tesis de Grado* (2023).

[47] Gavin Wood. *Ethereum Yellow Paper*. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[48] Jie Xu, Cong Wang y Xiaohua Jia. «A Survey of Blockchain Consensus Protocols». En: *ACM Comput. Surv.* 55.13s (jul. de 2023). ISSN: 0360-0300. DOI: 10.1145/3579845. URL: https://doi.org/10.1145/3579845.

[49] Zibin Zheng y col. «An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends». En: *2017 IEEE International Congress on Big Data (BigData Congress)*. 2017, págs. 557-564. DOI: 10.1109/BigDataCongress.2017.85.