**UNIVERSIDAD COMPLUTENSE DE MADRID**
FACULTAD DE INFORMATICA


# TESIS DOCTORAL

## Análisis estático de sistemas concurrentes y distribuidos: objetos concurrentess y Bytecode de Ethereum

## Static analysis of concurrrent and distributed systems: concurrent objects and Ethereum Bytecode

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

**Pablo Gordillo Alguacil**

Directores

**Elvira Albert Albiol**
**Samir Genaim**

Madrid

# Análisis Estático de Sistemas Concurrentes y Distribuidos: Objetos Concurrentes y Bytecode de Ethereum

# Static Analysis of Concurrent and Distributed Systems: Concurrent Objects and Ethereum Bytecode


**TESIS DOCTORAL**

*Memoria presentada para obtener el grado de doctor en Ingeniería Informática por*

**Pablo Gordillo Alguacil**

*Dirigida por los profesores*

**Elvira Albert Albiol**

**Samir Genaim**

Facultad de Informática
Universidad Complutense de Madrid

Madrid, Noviembre de 2019

# Análisis Estático de Sistemas Concurrentes y Distribuidos: Objetos Concurrentes y Bytecode de Ethereum


**TESIS DOCTORAL**

*Memoria presentada para obtener el grado de doctor en Ingeniería Informática por*
**Pablo Gordillo Alguacil**

*Dirigida por los profesores*
**Elvira Albert Albiol**
**Samir Genaim**

Facultad de Informática
Universidad Complutense de Madrid

Madrid, Noviembre de 2019

# Static Analysis of Concurrent and Distributed Systems: Concurrent Objects and Ethereum Bytecode


**PhD THESIS**

**Pablo Gordillo Alguacil**
Facultad de Informática
Universidad Complutense de Madrid

Advisors:
**Elvira Albert Albiol**
**Samir Genaim**

Madrid, November 2019

UNIVERSIDAD COMPLUTENSE MADRID

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña. Pablo Gordillo Alguacil,
estudiante en el Programa de Doctorado en Ingeniería Informática RD99/2011,
de la Facultad de Informática de la Universidad Complutense de Madrid, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Análisis Estático de Sistemas Concurrentes y Distribuidos: Objetos Concurrentes y Bytecode de Ethereum

Static Analysis of Concurrent and Distributed Systems: Concurrent Objects and Ethereum Bytecode

y dirigida por: Elvira Albert Albiol y Samir Genaim

**DECLARO QUE:**

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita.

Del mismo modo, asumo frente a la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada de conformidad con el ordenamiento jurídico vigente.

En Madrid, a 19 de noviembre de 2019

Fdo.: PABLO GORDILLO ALGUACIL

Esta DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD debe ser insertada en la primera página de la tesis presentada para la obtención del título de Doctor.

**Financial Support.** This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, the MICINN/FEDER, UE project RTI2018-094403-B-C31, by the CM projects S2013/ICE-3006 and S2018/TCS-4314 and by UCM CT27/16-CT28/16 grant.

# Resumen

**Título:** Análisis Estático de Sistemas Concurrentes y Distribuidos: Objetos Concurrentes y Bytecode de Ethereum.

Hoy en día la concurrencia y la distribución se han convertido en una parte fundamental del proceso de desarrollo de software. Indiscutiblemente, Internet y el uso cada vez más extendido de los procesadores multicore ha influido en el tipo de aplicaciones que se desarrollan. Esto ha dado lugar a la creación de distintos modelos de concurrencia. En particular, uno de los modelos de concurrencia que está ganando importancia es el *modelo de objetos concurrentes basado en actores*. En este modelo, los objetos (denominados actores) son las unidades de concurrencia. Cada objeto tiene su propio procesador y un estado local. La comunicación entre los mismos se lleva a cabo mediante el paso de mensajes. Cuando un objeto recibe un mensaje puede: actualizar su estado, mandar mensajes o crear nuevos objetos.

Es bien sabido que la creación de programas concurrentes *correctos* es más compleja que la de programas secuenciales ya que es necesario tener en cuenta distintos aspectos inherentes a la concurrencia como los errores asociados a las carreras de datos o a los interbloqueos. Con el fin de asegurar el correcto comportamiento de estos programas concurrentes se han desarrollado distintas técnicas de análisis estático y verificación para los diversos modelos de concurrencia existentes.

El análisis de *May-Happen-in-Parallel* (Puede-Ocurrir-en-Paralelo) es un análisis estático que infiere pares de puntos del programa que pueden ejecutarse en paralelo en distintas componentes distribuidas. En el modelo basado en actores, el análisis utiliza la información de los puntos del programa en los que se producen las llamadas asíncronas a métodos (es decir, los puntos donde se crean nuevas tareas), y las primitivas de sincronización (es decir, los puntos donde se espera a que las tareas terminen para continuar con su ejecución). Esta información ha resultado ser esencial para inferir propiedades de corrección (ausencia de bloqueo) y propiedades de viveza (terminación y consumo de recursos) de programas asíncronos.

Otro de los aspectos que ha influido en el desarrollo de sistemas distribuidos ha sido el auge de tecnologías y plataformas basadas en *cadena de bloques* (en adelante utilizaremos el término inglés blockchain) y los protocolos de consenso. Estos avances tecnológicos han permitido el desarrollo de plataformas *descentralizadas* como Bitcoin o Ethereum junto a sus respectivas criptomonedas: el Bitcoin y el Ether. Más allá de las criptomonedas, gracias a los *contratos inteligentes* (smart contracts) que han surgido en las plataformas blockchain, las aplicaciones de tecnologías basadas en blockchain son cada vez mayores. Los contratos inteligentes son programas informáticos donde se recogen los términos de

un contrato y son capaces de reaccionar de forma automática ante los distintos eventos especificados en los mismos. Los contratos pueden ser creados por cualquier usuario, y por tanto son susceptibles a errores de programación. Estos errores pueden provocar que se aborte la ejecución, lo que supone un coste económico necesario para revertir el estado de todos los nodos del sistema. Es por ello que el desarrollo de técnicas de análisis y verificación que permitan asegurar el correcto funcionamiento de los contratos inteligentes se ha convertido en un tema de investigación de gran interés y relevancia.

Dentro de este marco, los principales objetivos que se han perseguido con el desarrollo de esta tesis son:

- Estudiar los análisis de *May-Happen-in-Parallel* (MHP) existentes para distintos lenguajes y modelos de concurrencia. Concretamente se ha profundizado en el análisis MHP desarrollado para el lenguaje ABS [61], un lenguaje que implementa el modelo basado en actores y se han investigado mejoras en el análisis para mejorar su precisión y escalabilidad. También se ha estudiado la combinación de otros análisis para incrementar su precisión así como su aplicación a los contratos inteligentes.

- Desarrollar una *representación intermedia* para el bytecode de Ethereum. Esta representación permite la aplicación de análisis de alto nivel para inferir propiedades sobre el bytecode de Ethereum.

- Desarrollar análisis que permitan estudiar vulnerabilidades sobre contratos inteligentes de Ethereum. En particular se han estudiado las vulnerabilidades derivadas del *consumo de gas* asociado a la ejecución de los contratos inteligentes y las provocadas por la ejecución del bytecode `INVALID`.

Los objetivos anteriormente mencionados se ven reflejados en las siguientes publicaciones:

(i) El trabajo publicado en SAS'15 [22] presenta un análisis MHP con sincronización interprocedimental para lenguajes basados en objetos concurrentes. En él se extiende el análisis básico de MHP descrito en [19] para permitir la sincronización de una tarea en un entorno distinto al que la generó, es decir, una tarea generada por otra puede ser sincronizada (esperada) dentro de otra tarea distinta.

(ii) El trabajo presentado en ATVA'17 [23] presenta un análisis de MHP para programas asíncronos que utilizan variables futuras como mecanismo de sincronización. Este análisis es capaz de inferir la relación MHP existente entre variables futuras devueltas por tareas asíncronas. Al igual que el análisis anterior, extiende el análisis básico de MHP desarrollado en [19].

(iii) La publicación en ATVA'18 [24] presenta el framework EthIR para poder aplicar análisis de alto nivel sobre el código de bytes de Ethereum. EthIR decompila el bytecode de Ethereum en una representación intermedia de alto nivel basada en reglas. Esta representación reconstruye el flujo de control y de datos del bytecode a partir de una codificación de bajo nivel.

(iv) La publicación en ISSTA'19 [17] presenta SAFEVM, un verificador de contratos inteligentes de Ethereum que es capaz de hacer uso de tecnologías de verificación

existentes para programas en C. SAFEVM verifica condiciones introducidas en el código fuente original mediante aserciones, accesos a arrays y divisiones por 0.

(v) Por último, la publicación en VECoS'19 [25] presenta la herramienta GASTAP, capaz de inferir de forma automática sobreaproximaciones del gas que van a consumir las funciones públicas del contrato inteligente durante su ejecución. GASTAP implementa un análisis estático que infiere cotas superiores sobre el consumo de gas parametrizadas en función del tamaño de los argumentos de las funciones del contrato, del estado del contrato, o de datos del blockchain.

Finalmente, todos los análisis y técnicas desarrolladas son accesibles a través de interfaces web.

# Abstract

**Title:** Static Analysis of Concurrent and Distributed Systems: Concurrent Objects and Ethereum Bytecode.

Nowadays concurrency and distribution have become a fundamental part in the software development process. The Internet and the more extended use of multicore processors have influenced the type of the applications which are being developed. This has lead to the creation of several concurrency models. In particular, a concurrency model that is gaining popularity is the *actor model*, the basis for concurrent objects. In this model, the objects (actors) are the concurrent units. Each object has its own processor and a local state, and the communication between them is carried out using message passing. In response to receiving a message, an actor can update its local state, send messages or create new objects.

Developing *correct* concurrent programs is known to be harder than writing sequential ones because of inherent aspects of concurrency such as data races or deadlocks. To ensure the correct behavior of concurrent programs, static analyses and verification techniques have been developed for the diverse existent concurrency models.

The *May-Happen-in-Parallel* (MHP) analysis is a static analysis that infers pairs of program points that may execute in parallel across the different distributed components. In the actor model, the analysis uses the information from those program points where asynchronous calls to methods are made (i.e., the points where new tasks are spawned), and the synchronization primitives (i.e., the points where the tasks are awaited). This information has been proven to be essential to infer both safety properties (deadlock freedom) and liveness properties (termination and resource consumption) of asynchronous programs.

Other aspects that have influenced the progress of distributed systems are the growth of technologies and platforms based on blockchain and consensus protocols. These technological advances have allowed the development of decentralized platforms such us Bitcoin and Ethereum with their respective cryptocurrencies: Bitcoin and Ether. In addition, thanks to the *smart contracts*, the applications of these blockchain-based technologies are increasing.

Smart contracts are computing programs that contain the terms of a real contract and are able to react automatically to the various events specified on them. The contracts can be written by any user and, therefore, are prone to programming errors. These errors may abort the execution, which cause an economical cost to revert the state of each node of the system. Thus, analysis and verification techniques urge to ensure the correct behavior of smart contracts.

Within this framework, the main objectives achieved by this thesis are:

- Improving the existing MHP analyses. There are numerous MHP analyses developed for several languages and concurrency models. In particular, this thesis bases on the MHP analysis developed for the ABS language [61], a language based on the actor model and proposes improvements on the analysis to enhance its scalability and accuracy. It also studies how this analysis can be combined with others to increase its precision as well as its applicability to smart contracts.

- Developing an *intermediate representation* for Ethereum bytecode. This representation allows applying existent high level analyses to infer properties over the bytecode.

- Developing analyses to study vulnerabilities over Ethereum smart contracts. Specially, we have investigated vulnerabilities related to the *gas consumption* associated with the execution of smart contracts and those related to the execution of the bytecode instruction `INVALID`.

The objectives mentioned above are reflected in the following publications:

(i) The paper published at SAS'15 [22] presents an MHP analysis with inter-procedural synchronization for languages based on concurrent objects. This work extends the basic MHP analysis introduced in [19] to allow for the synchronization of a task in a distinct scope from the one where it was created, i.e., a task spawned by one task can be awaited within a different task.

(ii) The paper published at ATVA'17 [23] presents an MHP analysis for asynchronous programs that use future variables as synchronization mechanism. This analysis is able to infer the MHP relations that involve future variables that are returned by asynchronous tasks. As the previous analysis, it is based on the basic analysis presented in [19].

(iii) The paper published at ATVA'18 [24] presents the tool EthIR which decompiles the Ethereum bytecode into a high-level rule-based intermediate representation. This representation reconstructs the control and data flow of the bytecode from a low-level codification.

(iv) The paper published at ISSTA'19 [17] presents SAFEVM, a verifier of Ethereum smart contracts that makes use of state-of-the art verification engines for C programs. SAFEVM is able to verify conditions introduced in the original source code via assertions, array accesses and divisions by 0.

(v) Finally, the paper published at VECoS'19 [25] presents Gastap, a tool that is able to infer automatically overapproximations of the gas that a public function of a smart contract is going to consume during its execution. Gastap implements a static analysis that infers gas upper bounds parameterized in terms of the size of the arguments of the functions of the contract, the state of the contract or blockchain data.

Finally, all analyses and techniques developed are available online using a web interface.

# Agradecimientos

En primer lugar, esta tesis no habría sido posible sin la ayuda de mi directora Elvira Albert. Gracias por la inagotable capacidad de trabajo y por esa confianza en mi desde hace más de cinco años. Siempre disponible para cualquier cosa que he necesitado, con buenos consejos y palabras de ánimo. Pocas hay como ella, con su conocimiento y capacidad de liderazgo.

También me gustaría dar las gracias a Samir Genaim, mi otro director, por su paciencia, por estar siempre accesible, dispuesto a enseñar, ayudar y perder horas juntos instalando librerías o intentando crear un ejecutable. Nada malo podía tener un colchonero.

En tercer lugar, quería dar las gracias a Albert Rubio ya que, aunque no ha sido mi director, ha actuado como tal muchas veces. Gracias por el sentido del humor, las frases hechas y juegos de palabras. Ese entusiasmo se contagia trabajando y así es todo mucho más fácil.

No puedo olvidarme del resto de miembros del grupo COSTA, con los que he compartido grandes momentos: a Puri Arenas, nuestro viaje a Oslo y los cigarritos en la puerta de la facultad; a Jesús Correas y Guillermo Román por sus muchos consejos siempre acertados, nuestras tertulias políticas de sobremesa y nuestro "amor" por los artifacts pasando por Pekín; a Miky Gómez-Zamalloa por las horas de laboratorio de EDA y ese viaje relámpago a Ámsterdam; y a Enrique Martín con el que compartí un gran julio intentando arreglar determinado case study.

Mis compañeros del Aula 16 son también los que han compartido conmigo estos tres años: Cristina Alonso, Antonio Calvo, Marta Caro, Joaquín Gayoso y Alicia Merayo. Con ellos he vivido el día a día tanto dentro como fuera de la facultad y nunca les ha faltado un gesto o una palabra de apoyo cuando ha hecho falta. Gracias a Jesús Doménech y Luisma Costero, que además de formar parte del grupo anterior son compañeros de carrera, máster, y comenzamos todo este trayecto a la vez. Gracias por todo este tiempo, los cafés por las mañanas y las tardes en el despacho. Mención especial requiere Miguel Isabel, también a mi lado desde que empezamos la carrera y mi compañero de fatigas durante todo este tiempo. Juntos hemos ido dando los mismos pasos hasta llegar aquí.

Por último, quería agradecer a mi familia y a mi pareja por apoyarme durante estos años. Ellos son los que me aguantan cuando no estoy en la facultad. A mis padres, por darme todo lo que tengo y gracias a los cuales me he convertido en quien soy. Mi madre, la mujer más fuerte del mundo y mi padre, el hombre más bueno (además informático), a quien siempre podías ver con una sonrisa dibujada en la cara. Él fue quien me enseñó el primer ordenador que vi, a utilizar Google y a navegar por Internet mientras comunicaba el teléfono de casa. A mis hermanos, porque siempre les tengo detrás, haciendo el payaso y aguantando mis innumerables tonterías en cuanto llego a casa, y a mi pareja, pues aunque

sé que no le he dedicado todo el tiempo que a ella le hubiese gustado, siempre ha estado a mi lado, con sus palabras de ánimo y apoyo. No quiero olvidarme de mis yayos: ojalá todos los nietos tuviesen con sus abuelos la relación que yo he tenido con los míos.

*–Hijo, pero...¿tú qué haces en el trabajo?*

*–Pues cosas, yayo.*

*–¿Qué pasa?, ¿que son secretas y no me las puedes contar?*

A todos ellos, gracias por conseguir que este camino esté plagado de sonrisas y buenos recuerdos.

# Acknowledgments

First of all, this thesis would have not been possible without the help of my advisor Elvira Albert. She has always been available for everything that I have needed, with many good pieces of advice and encouraging words. There are a few like her, with her knowledge and leadership skills. Thank you for the endless working capacity and the trust in me for more than five years.

I want to thank Samir Genaim, my second advisor, for his patience, for being always available, willing to teach, help and spend hours together installing libraries or trying to build an executable. Not to mention that he supports Atletico de Madrid as I do.

In third place, I want to thank Albert Rubio, that, although he is not my advisor, he has helped me as an advisor several times. Thank you for your sense of humor and play on words. He spreads enthusiasm while working, which makes everything easier.

I would like to thank other researchers of COSTA group with whom I have shared great moments: Puri Arenas, for the trip to Oslo, the dinner in Copenhagen and the smoking breaks in front of the school; Jesús Correas and Guillermo Román for their many prosperous tips, for the discussions about politics after lunch, and for sharing the "love" for artifacts (going through Beijing); Miky Gómez-Zamalloa for the hours spend teaching me EDA in the lab, and for that quick trip to Amsterdam; and Enrique Martín for that July we spent fixing a case study.

My mates from Aula 16 who have shared these three years with me: Cristina Alonso, Antonio Calvo, Marta Caro, Joaquín Gayoso, and Alicia Merayo. They always had a gesture or a word of support when I needed it. I want to thank Jesús Doménech and Luisma Costero, who belong to the previous group, for those morning coffees and afternoons/evenings in the office. Especially, I also want to thank Miguel Isabel, who has always been by my side since we started our studies.

Finally, I want to make a special mention to my family and my partner. They are the ones that have supported me outside the university. My parents gave me everything that I have. My mother is the strongest woman in the world and my father was the best man (and a computer engineer). You could always see him with a smile on his face. He was the one showed me the first computer that I saw, the one that taught me how to use Google and how to surf the Internet. I thank my brothers for always being by my side, enduring my nonsense when I arrive home, and my partner, even though I have not spent as much time with her as she would have liked during these years, she has always been by my side supporting me. I cannot conclude without mentioning my grandparents: I wish all the grandchildren had a relationship with their grandparents like the one I had with them.

To all of them, thank you for making this road full of smiles and good memories.

# Contents

# Part I

# Contents of the Thesis

# Chapter 1

# Introduction

## 1.1 The Actor Concurrency Model

The actor model [13] is a concurrent programming model that has been gaining popularity and is used by languages such as ABS, Erlang or Scala. A program based on this model will define a number of actors, which represent the computing units. Each actor has its own local state and thread of control and communicates with other actors by sending messages asynchronously. Each actor enqueues the pending messages that has received and at each step in the computation of an actor system, an actor is scheduled to process one of its pending messages.

*Concurrent objects* allow implementing actor systems in an object-oriented style of programming. The model considers the objects as actors, which are the concurrency units, with their own dedicated processor. The communication process is modeled with asynchronous calls (tasks) to methods of the same or different object. An intrinsic feature of this model is that the scheduling is *cooperative* (or non-preemptive), i.e., switching between tasks of the same object happens only at specific synchronization program points during its execution, which are explicit in the source code. This contrasts with other concurrency models such as the P-thread model used by Java, where the user does not have any control over the threads that are been executed.

The concurrent objects model uses *future variables* [48, 44] to synchronize the execution of the tasks. A method call `m` on some parameters $\bar{y}$, written as `x=o!m(`$\bar{y}$`)`, spawns an asynchronous task on object `o`, and the future variable `x` allows synchronizing with the termination of such task by means of the instruction **await** `x?`; which checks if method `m` has finished its execution and releases the processor of the current task, allowing other available task to take it. A future variable can be awaited inside the same task that spawned the method which the future variable is bound to (intra-procedural synchronization) or the task bound to the future variable can be awaited in a different task to the one that spawned it (inter-procedural synchronization). Future variables are available in most concurrent languages: Java, Scala and Python allow creating pools of threads. The users fix the set of the pool indicating how many threads it will contain. They can submit tasks to the pool, which are executed when a thread of the pool is idle, and may return future variables to synchronize with the tasks termination. The pool has an internal queue which holds the extra tasks in case that there are more tasks submitted to the pool than threads created. C++ includes the components async, future and promise in its standard thread library,

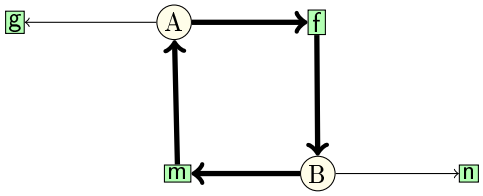



Figure 1.1: Partial data-flow graph with a deadlock cycle.

which allow programmers to create tasks (instead of threads) and return future variables in the same way as we do. The users can create tasks using the async instruction, which receives the name of the method to execute and its arguments or a lambda function. These tasks return future variables that allow the users synchronize with the result of the tasks via the get statement, which blocks the execution until the result is available. The promise objects are related to the future objects because they share the data. The promise objects are used to set the value of their futures inside tasks.

MHP is a fundamental analysis to prove both liveness and safety properties of concurrent programs. The analysis over-approximates which are the pairs of program points whose execution might happen in parallel in an (concurrent) interleaved way within one processor or in parallel across different processors. In this fragment of code x=o!m($\bar{y}$);...; **await** x?; the execution of the instructions of the asynchronous task m may happen in parallel with the instructions between the asynchronous call and the **await**. However, due to the **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. This piece of information can be used to prove more complex safety and liveness properties:

- in [49], MHP pairs are used to discard unfeasible deadlock cycles. Figure 1.1 contains a possible deadlock cycle showed with bold arrows in the graph that involves two concurrent objects A and B. The execution starts from a main block by creating the two concurrent objects that make an asynchronous call to f and m respectively. The call to f in object A spawns a new task n on object B and gets blocked waiting for its termination. The task m on object B, that is running in parallel with f, proceeds in the same way. It spawns task g on object A and waits until it finishes its execution. Thus, if the MHP analysis infers that tasks n and g cannot run in parallel, the deadlock cycle in Figure 1.1 represents an unfeasible one and has to be discarded.

- in [20], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. As a simple example, consider a task p that contains as unique instruction y=−1, where y is a global variable. The following loop y=1; **while**(i>0){i=i−y;} contained in a task r might not terminate if p runs in parallel with it, since p can modify y to a negative value and the loop counter in r will continue increasing. However, if the MHP analysis can guarantee that p will not run in parallel with r, the program terminates and we can infer resource-boundedness for the loop.

## 1.2 Ethereum Smart Contracts

Ethereum [95] is one of the most popular blockchain based technologies. It improves the blockchain platform of Bitcoin [78] adding more functionalities to it. The blockchain of Bitcoin only stores information about the transaction executed. However, the blockchain of Ethereum is able to keep not only information about the transactions, but also data structures and objects. Ethereum adds to its blockchain platform a Turing-complete virtual machine, the *Ethereum Virtual Machine* (EVM) and hence, it is able to execute programs (known as *smart contracts*). Smart contracts are programs that contain the terms of a real contract and are able to react automatically to events specified on them. Typical applications of smart contracts involve implementations of multi-party accounting, voting systems and games with reward distributions. There are several languages to write them as Vyper [6], Serpent [4] or Bamboo [1] although Solidity [5] is the most popular one. Solidity [5] is an object-oriented, high-level language. It is statically typed and supports multiple inheritance, libraries and user-defined types.

The code of a smart contract is compiled into EVM bytecode, which is deployed and stored in the blockchain. The EVM bytecode is a stack-based language that contains a small set of primitives and instructions that allow computing arithmetic and logic operations, calling other contracts, accessing the global blockchain state, initiating sub-transactions or creating new contracts among others. EVM has three different allocations to store items: (a) the *storage* is where the *state* variables (global variables) of the contract reside, is persistent to external function calls and each contract has its own storage; (b) the *memory* is a volatile storage used to keep temporary values and local to each transaction, i.e, each transaction starts with an empty memory; (c) the *stack* is used to compute operations and can only keep a limited amount of elements.

EVM provides a compilation target for the different high-level languages mentioned above. In addition, analyzing the bytecode rather than the source code of a smart contract could be necessary because (i) the blockchain only stores the bytecode of the contracts (it contains the source code for less than 1% of the contracts that it holds [52]); (ii) there is information needed by analyzers that is only available at the bytecode level such as these related to gas consumption or verification purposes; (iii) to avoid that the analyses get affected by optimizations of the compiler. However, the differences between EVM bytecode and the bytecode of other high-level languages like Java make its analysis more challenging:

- Although the source code is typed, EVM bytecode is an untyped language.
- There is no notion of method or data structure. An EVM bytecode can be seen as one function and calls between methods of the contract are translated into jumps between different addresses of the bytecode.
- The jump addresses are not known statically. When a jump instruction is executed it takes the destination address from the top of the stack.
- The size of the stack is not fixed. A program point of the bytecode can be reached with different sizes of the stack due to several calling contexts, i.e, a function call executed in different program points of the contract may be translated into jumps to the same code address (the starting address of the function called) that can be reached with a different number of items in the stack.

Once a smart contract is deployed, it can be invoked by parties involved in the protocol. The execution of a transaction includes calls to methods of the contract. The transaction is replicated across all the system. To maintain its consistency, Ethereum uses the notion of *gas*, a monetary value in *Ether* (the cryptocurrency of Ethereum), that measures the amount of computational resources that a transaction needs to be executed. Computations that require *more computational* or *storage resources*, cost more gas than those that require fewer resources. The gas consumption of each bytecode instruction is specified in [95] and is independent of the platform where the transactions are executed. The execution of a transaction using the same context always costs the same amount of gas. *Gas* is used to avoid potential denial-of-service attacks on the system. When a transaction is executed, a node *pays* for the cost of executing it using Ether. Thus, if the amount of gas is insufficient to execute the transaction, it will be aborted. It also disincentives consuming unnecessary *storage* as it is more expensive than the *memory*.

Note that smart contracts are inmutable once they are deployed in the blockchain, i.e, their behavior can not be modified. Thus, it is of utmost importance to have tools that allow the developers to analyze and test the contracts before they are deployed.

## 1.3 Smart Contracts as Actors

Smart contracts behave as concurrent objects. Like concurrent objects, they have its own mutable state, represented by *state variables* and the *blockchain data*, and public methods that are executed atomically. A contract can be accessed by *accounts* or *addresses* that belong to users or contracts. As mentioned above, the execution of a transaction consists in executing a *concurrent* call that runs a corresponding method of the smart contract and is able to modify the contract state. The execution of the methods involved in the transaction in principle is atomic. However, the atomicity can be broken using the "call" primitive (as we will discuss later). As regards the scheduling of transactions, it is *cooperative* (as explained in Section 1.1). This implies that the order in which they are executed is not known and can lead to different outcomes.

Figure 1.2 shows to the left a fragment of the `EthereumPot` [7] smart contract written in Solidity and to the right the implementation using the concurrent objects language ABS [61]. `EthereumPot` implements a simple lottery. During a game, players call a method `joinPot` to buy lottery tickets; each player's address is appended to an array `addresses` of current players, and the number of tickets is appended to an array `slots`, both having variable length. After some time has elapsed, anyone can call `rewardWinner` which calls the `Oraclize` service to obtain a random number for the winning ticket. If all goes according to plan, the `Oraclize` service then responds by calling the `__callback` method with this random number and the authenticity proof as arguments. A new instance of the game is then started, and the winner is allowed to withdraw the balance using a `withdraw` method.

Note that the differences between the Solidity and ABS programs mainly are syntactic:

- Contracts are concurrent objects (Line 1 and Line 26 in Figure 1.2).

```
1  contract EthereumPot is usingOraclize{
2   address[ ] public addresses;
3   address public winnerAddress;
4   uint[ ] public slots;
5   ...
6   function __callback(bytes32 _queryId,···)
7     oraclize_randomDS_proofVerify(_queryId,···){
8    if(msg.sender != oraclize_cbAddress()) throw;
9    ...
10   winnerAddress = findWinner(rnd_nmbr);
11   amount = this.balance * 98 / 100 ;
12   winnerAnnounced(winnerAddress, amount);
13   if(winnerAddress.send(amount)){
14    if(owner.send(this.balance)){
15      openPot();
16    }}
17   }
18  function findWinner(uint random) returns(address){
19   for(uint i = 0; i < slots.length; i++){
20    if(random <= slots[i]) {
21     return addresses[i];
22    }}
23  }
24  ...
25 }
```

```
26 class EthereumPot{
27  Array<Address> addresses;
28  Address winnerAddress;
29  Array<Int> slots;
30  Oraclize o;
31  Address owner;
32  Int balance = 1000;
33  ...
34  Unit __callback(AddressI msg_sender){
35    Fut<Unit> x;
36    Fut<Address> y;
37    Fut<Bool> s;
38    x = o!oraclize_randomDS_proofVerify();
39    await x?;
40    y = o!oraclize_cbAddress();
41    await y?;
42    if(msg_sender != y.get) return;
43    ...
44    winnerAddress = findWinner(rnd_nmbr);
45    amount = balance * 98 / 100 ;
46    winnerAnnounced(winnerAddress, amount);
47    s = winnerAddress!send(amount);
48    await s?;
49    if(s.get){
50      s = owner!send(balance)
51      await s?;
52      if(s.get){
53        openPot();
54      }}
55  }
56  Address findWinner(Int random){
57   for(Int i = 0; i < length(slots); i++){
58    if(random <= slots[i]){
59     return addresses[i];
60    }}
61  }
```

Figure 1.2: Fragment of `EthereumPot` contract. Solidity implementation (left). Concurrent objects implementation in ABS (right).

- Implicit parameters, mainly blockchain data and information about the transaction such as the balance or the address of the sender of the transaction, of the smart contracts are translated into explicit parameters of the functions that use them (`msg.sender` on Line 8 is translated into the argument `msg_sender` of function `__callback` at Line 34).

- The *balance* is translated into a field variable on the object (Line 11 and Line 32) in ABS language.

- Accounts (Line 1, Line 3, Line 14) are concurrent objects (Line 30, Line 28, 31).

- Calls to methods of the same contract (Line 10 and Line 12) are translated into synchronous calls (Line 44 and Line 46).

- Calls to methods of different contracts (Line 8, Line 13 and Line 14) are translated into asynchronous calls to methods of other object that is stored as a field variable (Line 40, Line 47 and Line 50).

- Returned values of external methods to the contract (Line 8, Line 13 and Line 14) are translated into future variables. The future variables are awaited until the methods bound to them finish their execution in the **await** statement. After that, the value can be accessed using the **get** instruction (Lines 40-42, Lines 47-49 and Lines 50-52).

Therefore, techniques and analyses developed for languages based on the actor model can be adapted or applied directly on smart contracts. Interestingly, the MHP analysis can be considered as a previous step to study *reentrancy vulnerabilities* on smart contracts. Reentrancy vulnerabilities are a well-known type of security bug [27, 55] that may make a contract lose all the funds that it owns. The cause of these bugs is related to the "call" primitive. When you call a contract, you can transfer Ether to the account of the contract or execute a public function of the contract. In particular, when you use the call instruction without specifying a function, the fallback (an anonymous method) of the called contract will be executed. Then, the fallback can invoke any function of the caller contract. If it invokes the same function, it can create a chain of calls that will run until (i) all the Ether of the contract is sent to the called contract (the one that executes the fallback), (ii) the stack limit is reached or, (iii) the execution runs out-of-gas. Although these three cases will throw an exception, the instruction call does not propagate it (call primitives as call, send, delegatecall and staticcall return true or false depending on whether the execution of the called method finishes correctly or not). Hence, only the execution of the last call is reverted and the called contract drains all the funds that the sender contract has. This is the main logic of the DAO attack [74], one of the most popular attacks that caused a contract to lose 60 million dollars in June 2016 [2]. Figure 1.3 shows a sample of its behavior: mainly, the EasyDAO contract has a map of addresses and balances (Line 3) and a function payout (Line 8) that checks if the balance bound to the account of the caller contract is positive (Line 9). In this case, it transfers the corresponding amount of Ether to this account with a call primitive (Line 10) and after that, updates the balance of the account (Line 11). However, this sequence is not atomic due to the use of call. Assume that a malicious user creates the Attacker contract. Once it is deployed, it puts some Ether in EasyDAO using the function setBalance (Line 14). After that, Attacker calls the function payout that transfers to Attacker all the balance that it owns. As the call instruction does not specify any function, the fallback of the Attacker contract (Line 23) is executed and calls payout (Line 24) which starts running. At that point, the balance has not been updated yet so it transfers the specified amount of Ether again, creating a loop between these two calls until it gets all the Ether that the EasyDAO holds. Finally, when the execution ends, Attacker can transfer all the Ether that has drained to a specific user account.

If we view the smart contracts as concurrent objects, we can adapt existing tools to reason and verify their behavior. The MHP analysis could be used to find concurrency vulnerabilities on smart contracts. It could infer the MHP relations between contracts to analyze which functions may interleave their execution or be adapted to build *happens-before* relations together with a *must-have-finished* [22] analysis that allows establishing an order in the execution of the functions of the contracts [64]. The MHP analysis is used also as part of other analyses that can be applied to smart contracts such as termination

```
1  contract EasyDAO{
2  ...
3   mapping (address => uint) public balances;
4  ...
5   function getBalance(address account){
6     return balances[account];
7    }
8   function payout(uint amount){
9     if (balances[msg.sender]>0){
10          msg.sender.call.value(amount)();
11          balances[msg.sender]-=amount;
12    }
13  }
```

```
14   function setBalance(){
15     balances[msg.sender] += msg.value;
16    }
17    ...
18  }
19  contract Attacker{
20  ...
21   EasyDAO d = EasyDAO(···);
22  ...
23   function (){
24     d.payout(dao.getBalance(this));
25   }
26    ...
27  }
```

Figure 1.3: Sample of the behavior of DAO contract.

analyses. Although the execution of a transaction always finishes, an infinite loop would provoke the loss of all the Ether that the user has.

Although concurrency does not exist in smart contracts (and Ethereum), they may behave as distributed systems due to the execution of *callbacks*. Informally, there is a callback when two smart contracts interact in the following way: a function of the first smart contract calls a function of the second one, and the function of the callee calls back a function of the first object. Thus, the execution of a callback can modify the state of the initial smart contract and break its modularity. The fragment of Solidity code of the smart contract `EthereumPot` shown in Figure 1.2 contains a callback. The callback is executed when a function of the `Oraclize` services is invoked.

It is difficult to reason about distributed properties of smart contracts due to two reasons: (i) a smart contract may have several different clients, and (ii) the code of the smart contract that calls the callback is not usually available. State-of-the-art techniques try to prove that callbacks do not affect the modularity of the contract [64, 9, 55] instead of reasoning about distributed aspects. To this end, the MHP analysis can only be used to reason on the clients whose code is available, but would not be useful to infer the effectively callback-free property of [64, 9, 55].

## 1.4 Contributions and Structure of the Thesis

This thesis is presented in "publication format". These publications contain all the technical details and results obtained from the different research works developed. All the papers have been published in the proceedings of international conferences. In what follows, we summarize the main contributions of the thesis.

1. *Enhanced MHP Analyses.* We present to the best of our knowledge the first MHP analyses [22, 23] that captures MHP relations that involve tasks that are awaited in an outer or inner scope from the scope in which they were created. This happens when future variables are returned by the asynchronous tasks or when they are passed to the asynchronous task as parameters respectively, as it can be performed in all programming languages that have future variables. Our analyses are built on top of an existing MHP analysis [21] that is not able to track information propagated through future variables that are returned by tasks or passed as arguments to them.

The original MHP analysis [21] involves two phases: (1) a local analysis which consists in analyzing the instructions of the individual tasks to detect the tasks that it spawns and awaits, and (2) a global analysis which propagates the local information compositionally.

In this thesis we have enhanced the MHP analysis [21] to handle the synchronization of future variables in inner [22] and outer [23] scopes from the scope in which they were created:

- In [22], we develop a novel *must-have-finished* (MHF) analysis which infers *inter-procedural dependencies* among the tasks. Such dependencies allow us to determine that, when a task finishes, those that are awaited for on it must have finished as well. The analysis is based on using Boolean logic to represent abstract states and simulate the corresponding operations. The local phase needs to integrate the above MHF information to consider the inter-procedural dependencies inferred. The global phase has to be the refined in order to eliminate spurious MHP pairs which appear when inter-procedural dependencies are not tracked.
- In [23], the local phase needs to be modified to backpropagate the additional inter-procedural relations that arise from the returned future variables. Backpropagation is achieved by modifying the data-flow of the analysis so that it iterates to propagate the new dependencies. The global phase has to be modified by reflecting in the analysis graph the additional information provided by the local phase.

2. *The* EthIR *framework for smart contracts.* In this thesis we also present a first prototype of a decompiler [24] called EthIR, that creates an intermediate representation for Ethereum bytecode which we have later improved in [25]. It translates the bytecode of the contract into a *rule-based representation* (RBR) that can be injected to existing analyzers with minor changes to infer properties of the `EVM`. The *rule-based representation* contains a set of rules that maintains the control-flow and data-flow of the program. It uses explicit variables to represent the data stored in the stack, state and local variables and blockchain data. It uses the Oyente tool in order to generate a control-flow graph (CFG) of the contract. We had to modify the Oyente tool to capture all feasible execution paths and enrich the information gathered in the CFG. The RBR is built translating each block of the CFG into rules. The decompilation phase is enhanced in [25] making the CFG built by Oyente complete. In particular, the goal of Oyente is to perform symbolic execution in order to detect bugs rather than to generate a complete CFG. We have had to remove execution bounds of Oyente such as loop bound and depth bound and use information about the path under analysis in order to build a complete CFG.

   The RBR generated by EthIR is used by the two next analysis tools presented in this thesis.

3. *A Safety Analyzer for smart contracts.* SAFEVM [17] is, to the best of our knowledge, the first tool that uses existing verification engines developed for C programs to verify low-level `EVM` code. It takes as input a public function of a smart contract

and the source code or the bytecode of the contract and transforms it into a C program. Using C verification engines, SAFEVM produces a verification result for guaranteeing the unreachability of the `INVALID` operations. The reachability of the `INVALID` bytecode instruction means that the transaction analyzed aborts its execution reverting the state of the contract and not refunding the initial gas of the transaction.

4. *A Resource Analyzer for smart contracts.* Finally, GASTAP [25] is, to the best of our knowledge, the first automatic gas analyzer for smart contracts. GASTAP takes as input a smart contract and automatically infers an upper bound on the gas consumption for each of its public functions. The upper bounds that GASTAP infers are given in terms of the sizes of the input parameters of the functions, the contract state and/or on blockchain data. It takes the RBR generated by ETHIR, abstracts it and infers size relations using an adaptation of the size analyzer of SACO [15] that are used to compute the gas equations, which are solved by PUBS [16] to infer closed-form gas bounds.

The remaining of this thesis is structured as follows. In Chapter 2 we first describe an existing MHP analyses and in Section 2.1 and Section 2.2 we introduce the local and global phases of our extensions with inter-procedural synchronization and returned future variables respectively. In Chapter 3 we introduce how the RBR is built taking a CFG as starting point. In Chapter 4 we describe the translation from the RBR of a smart contract into a C program. In Chapter 5 we introduce the generation of size relations and gas equations to infer gas upper-bounds improving the state-of-the art tools. In Chapter 6 we conclude and discuss future work. Finally Chapter 7 includes the papers that make up this thesis where the technical parts of each chapter is explained in detail. The list of papers included in the thesis in chronological order is the following:

- Elvira Albert, Samir Genaim and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015*. Proceedings, volume 9291 of Lecture Notes in Computer Science, pages 72-89. Springer, September 2015.

- Elvira Albert, Samir Genaim and Pablo Gordillo. May-Happen-in-Parallel Analysis with Returned Futures. In *15th International Symposium on Automated Technology for Verification and Analysis, ATVA 2017*. Proceedings, volume 10482 of Lecture Notes in Computer Science, pages 42-58. Springer, October 2017.

- Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018*. Proceedings, volume 11138 of Lecture Notes in Computer Science, pages 513-520. Springer, October 2018.

- Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*. ACM, pages 386–389, July 2019.

- Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In *13th International Conference on Verification and Evaluation of Computer and Communication Systems 2019, VECoS 2019.* Proceedings, volume 11847 of Lecture Notes in Computer Science, pages 63-78. Springer, October 2019.

# Chapter 2

# May-Happen-in-Parallel Analysis

This thesis extends the original MHP analysis of Albert et al. [21]. MHP analysis [21] is a static analysis for languages based on *concurrent objects* [13] that over-approximates the pairs of program points that can interleave their execution. As mentioned in Chapter 1, the concurrent objects model uses a *cooperative* scheduler, i.e, the tasks are synchronized at specific program points that are explicit in the code. The MHP analysis learns from the future variables used in synchronization instructions when tasks are terminated, so that the analysis can discard unfeasible MHP pairs. The analysis is carried out in two phases:

- Local phase: It considers the methods separately and infers information related to the status (active, pending or finished) of the tasks that are created locally in the method under analysis for each of its program points. It does not take into account transitive calls.
- Global phase: It composes the information inferred in the local phase in a MHP graph where the transitive relations are reflected. The analysis uses the graph to infer the set of MHP pairs.

The original analysis [21] only supports intra-procedural synchronization, i.e, the tasks must be awaited in the same scope in which they were created. In this thesis we enhance the original MHP analysis in order to track inter-procedural synchronization arising from (1) passing future variables as method parameters [22]; or (2) returning future variables from one method to another [23]. Both analyses have been implemented in SACO [15], a static analyzer for ABS [61] programs. The ABS language allows inter-procedural and intra-procedural synchronization. SACO has a web interface and can be used online at `https://costa.fdi.ucm.es/saco/web`.

## 2.1 MHP Analysis with Interprocedural Synchronization

In this section we present a MHP analysis with inter-prodecural synchronization [22], which is based on incorporating a MHF set into the original analysis [21]. In Section 2.1.1 we describe the MHF analysis and the information that it provides, and in sections 2.1.2 and 2.1.3 we describe how we modify the two phases of the original analysis, and describe the gain of precision with respect to the original MHP analysis [21] (see Sections 4 and 5 in [22] for technical details).

Figure 2.1 shows a simple program that starts executing method $\mathsf{main}_1$. The **skip** statement abstracts all instructions that do not affect to the concurrency of the program. The call to f at program point 3 creates a task bound to future variable x. At program point 5, method $\mathsf{main}_1$ calls g that receives the future variable x as parameter and is bound to the future variable z. Task g is active until the program reaches program point 7 where the task becomes finished. Task g receives a future variable as parameter and gets blocked at program point 16 until the task bound to the future variable passed as parameter to g finishes its execution. Thus, task f terminates its execution at program point 16.

The original MHP analysis infers that task g finished its execution at program point 7 but it is not able to infer that at this program point task f has finished too. Our goal is to infer precise MHP information that describes, among others, the following representative cases:

(1) any program point of g cannot run in parallel with program point 8, because at program point 7 method $\mathsf{main}_1$ awaits for g to terminate;

(2) program point 11 cannot run in parallel with program point 8, since when waiting for the termination of g at program point 7 we know that f *must-have-finished* as well due to the *dependency* relation that arises when $\mathsf{main}_1$ implicitly waits for the termination of f; and

(3) program point 11 cannot run in parallel with program point 17, because f *must-have-finished* due to the synchronization on the local future variable w at program point 16 that refers to future variable x of $\mathsf{main}_1$.

### 2.1.1 Must-Have-Finished Analysis

The *must-have-finished* (MHF) analysis captures inter-method synchronization and uses it to improve the local and the global phases of the MHP analysis. For instance, this analysis will infer that when reaching *program point 17* (L17 for short) in Figure 2.1, it is guaranteed that whatever task bound to w has finished already, and that when reaching *program point 8*, it is guaranteed that whatever tasks bound to x and z have finished already.

MHF analysis computes, for each program point $\ell$ of the program, a set of *finished future variables*, i.e., whenever $\ell$ is reached those variables are either not bound to any

```
1  main1() {                10  f() {
2    skip;                  11    skip;
3    x=o1!f();              12  }
4    skip;                  13
5    z=o2!g(x);             14  g(w) {
6    skip;                  15    skip;
7    await z?;              16    await w?;
8    skip;                  17    skip;
9  }                        18  }
```

Figure 2.1: Example for MHP analysis with inter-procedural synchronization.

task (i.e., have value $\bot$) or their corresponding tasks are guaranteed to have terminated. We refer to such sets as MHF sets.

**Example 2.1.** *The MHF sets for the program points of Figure 2.1 are:*

| | |
|---|---|
| $L2$: {x,z} | $L9$ : {x,z} |
| $L3$: {x,z} | $L11$: {} |
| $L4$: {z} | $L12$: {} |
| $L5$: {z} | $L15$: {} |
| $L6$: {} | $L16$: {} |
| $L7$: {} | $L17$: {w} |
| $L8$: {x,z} | $L18$: {w} |

*At program points that correspond to method entries, all local variables (but not the parameters) are finished since they point to no task. For* g*: at program point 15 and program point 16 no task is guaranteed to have finished, because the task bound to* w *might be still executing; at program point 17 and program point 18, since we passed through* **await** *w? already, it is guaranteed that* w *is finished. For* $\mathsf{main}_1$*: at program point 8 and program point 9 both* z *and* x *are finished. Note that* z *is finished due to* **await** *z?, and* x *is finished due to the implicit dependency between the termination of* x *and* z*.*

MHF analysis over-approximates the MHF set for each program point. It uses Boolean formulas to represent MHF states, since their models naturally represent MHF sets, and, moreover, Boolean connectives to model the abstract execution of the different instructions. An MHF state for the program points of a method $m$ is a propositional formula. The set of all models of this propositional formula coincide with the MHF sets of method $m$. The execution of the different instructions can be modeled with Boolean formulas. Given a program point, a MHF state and a instruction to execute, we compute a new MHF state that represents the effect of executing the instruction. The procedure builds a set of data-flow equations whose solutions associate with each program point a MHF state that over-approximates the real one.

### 2.1.2 Local MHP

The local MHP analysis (LMHP) considers each method $m$ separately, and for each program point $\ell$ it infers an LMHP state that describes the tasks that might be executing when reaching $\ell$ (considering only tasks invoked in $m$). An LMHP state $\Psi$ is a *multiset* of MHP atoms, where each atom represents a task and can be of the form:

(1) $y{:}\ell'{:}\mathtt{T}(m, \mathtt{act})(\bar{x})$, which represents an *active* task that might be at any of its program points, including the exit one, and is bound to future variable $y$. Moreover, this task is an instance of method $m$ that was called at program point $\ell'$ (the *calling site*) with future parameters $\bar{x}$; or

(2) $y{:}\ell'{:}\mathtt{T}(m, \mathtt{fin})(\bar{x})$, which differs from the previous one in that the task can only be at the exit program point, i.e., it is a *finished* task.

In both cases, future variables $y$ and $\bar{x}$ can be $\star$, which is a special symbol indicating that we have no information on the future variable.

Intuitively, the MHP atoms of $\Psi$ represent (local) tasks that are executing in parallel. However, since a variable $y$ cannot be bound to more than one task at the same time, atoms bound to the same variable (it could happen in conditional statements) represent *mutually exclusive tasks*, i.e., cannot be executing at the same time. The same holds for atoms that use *mutually exclusive calling sites* $\ell_1$ and $\ell_2$ (i.e., there is no path from $\ell_1$ to $\ell_2$ and vice versa) cannot be executing at the same time.

The use of multisets allows including the same atom several times to represent different instances of the same method. We let $(a, i) \in \Psi$ indicate that $a$ appears $i$ times in $\Psi$. Note that $i$ can be $\infty$, which happens when the atom corresponds to a calling site inside a loop, this guarantees convergence of the analysis. Note that the MHP atoms of the original MHP analysis do not use the parameters $\bar{x}$ and the calling site $\ell'$, since they do not benefit from such extra information.

**Example 2.2.** *The following are LMHP states for some program points from Figure 2.1:*

| | |
|---|---|
| $L2$: {} | $L9$ : {$x$:3:$\mathtt{T}(f, \mathtt{fin})()$,$z$:5:$\mathtt{T}(g, \mathtt{fin})(x)$} |
| $L3$: {} | $L11$: {} |
| $L4$: {$x$:3:$\mathtt{T}(f, \mathtt{act})()$} | $L12$: {} |
| $L5$: {$x$:3:$\mathtt{T}(f, \mathtt{act})()$} | $L15$: {} |
| $L6$: {$x$:3:$\mathtt{T}(f, \mathtt{act})()$,$z$:5:$\mathtt{T}(g, \mathtt{act})(x)$} | $L16$: {} |
| $L7$: {$x$:3:$\mathtt{T}(f, \mathtt{act})()$,$z$:5:$\mathtt{T}(g, \mathtt{act})(x)$} | $L17$: {} |
| $L8$: {$x$:3:$\mathtt{T}(f, \mathtt{fin})()$,$z$:5:$\mathtt{T}(g, \mathtt{fin})(x)$} | $L18$: {} |

*Let us explain some of the above LMHP states. The state at $L6$ includes $x$:3:$\mathtt{T}(f, \mathtt{act})()$ and $z$:5:$\mathtt{T}(g, \mathtt{act})(x)$ for the active tasks invoked at $L3$ and $L5$.*

The LMHP states are inferred by a *data-flow analysis* which is defined as a solution of a set of LMHP constraints obtained by applying a transfer function to the instructions. Recall that the role of the transfer function in a data-flow analysis is to abstractly execute the different instructions, i.e., transforming one LMHP state to another. The main difference w.r.t. the original MHP analysis of [21] is the treatment of **await** $z$?: while we use an MHF set computed using the inter-procedural MHF analysis of Section 2.1.1 to modify the state of all the tasks that have finished their execution when the task bound to $z$ ends, the original MHP analysis only changes the status of the task bound to future variable z, which is obtained syntactically from the instruction. Our LMHP analysis, as in the original MHP analysis, is defined as a solution of a set of LMHP constraints.

### 2.1.3 Global MHP

The results of the LMHP analysis are used to construct an MHP graph, from which we can compute the desired set of MHP pairs. The construction is exactly as in the original MHP analysis except that we carry the new information in the MHP atoms. However, the process of extracting the MHP pairs from such graphs will be modified.

In what follows, we use $y{:}\ell{:}\mathtt{T}(m, X)(\bar{x})$ to refer to an MHP atom without specifying if it corresponds to an active or finished task, i.e., the symbol $\mathtt{T}(m, X)$ can be matched to $\mathtt{T}(m, \mathtt{act})$ or $\mathtt{T}(m, \mathtt{fin})$. We let $\mathtt{ppoints}(m)$ and $\mathtt{ppoints}(P)$ be the set of program points of method $m$ and program $P$ respectively. As in the original MHP analysis, the nodes of the MHP graph consist of two method nodes $\mathtt{T}(m, \mathtt{act})$ and $\mathtt{T}(m, \mathtt{fin})$ for each method $m$, and a program point node $\ell$ for each $\ell \in \mathtt{ppoints}(P)$. Edges from $\mathtt{T}(m, \mathtt{act})$ to

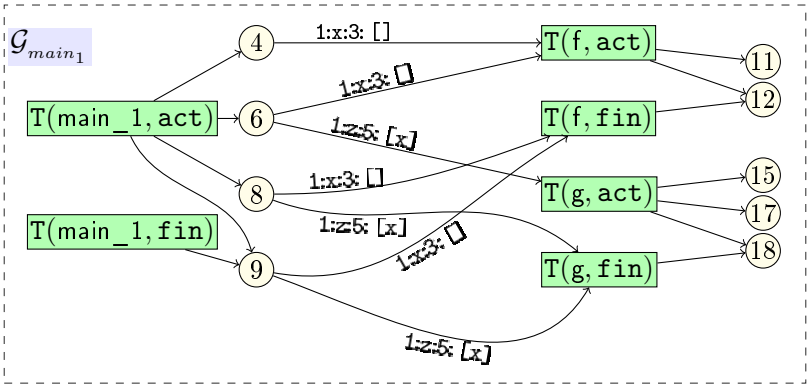



Figure 2.2: MHP graph $\mathcal{G}_{main_1}$ corresponds to analyzing $\mathsf{main}_1$.

each $\ell \in \texttt{ppoints}(m)$ indicate that when $m$ is active, it can be executing at any program point, including the exit, but only one. An edge from $\texttt{T}(m, \texttt{fin})$ to $\ell_m$ indicates that when $m$ is finished it can be only at its exit program point. The out-going edges from a program point node $\ell$ reflect the atoms of the corresponding LMHP state $\Psi_\ell$ as follows: if $(y{:}\ell'{:}\texttt{T}(m, X)(\bar{x}), i) \in \Psi_\ell$, then there is an edge from node $\ell$ to node $\texttt{T}(m, X)$ and it is labeled with $i{:}y{:}\ell'{:}\bar{x}$. These edges simply indicate which tasks might be executing in parallel when reaching $\ell$, exactly as the LMHP set does.

**Example 2.3.** *The MHP graph $\mathcal{G}_{main_1}$ Figure 2.2 corresponds to method $\mathsf{main}_1$, analyzed together with its reachable methods. For simplicity, the graph includes only some program points of interest. Note that the out-going edges of program point nodes coincide with the LMHP states of Example 2.2.*

The procedure of the original MHP analysis for extracting the MHP pairs from the (modified) MHP graph of a program $P$, denoted $\mathcal{G}_P$, is based on the following principle: $(\ell_1, \ell_2)$ is an MHP pair induced by $\mathcal{G}_P$ iff

(i) $\ell_1 \rightsquigarrow \ell_2 \in \mathcal{G}_P$ or $\ell_2 \rightsquigarrow \ell_1 \in \mathcal{G}_P$; or

(ii) there is a program point node $\ell_3$ and paths $\ell_3 \rightsquigarrow \ell_1 \in \mathcal{G}_P$ and $\ell_3 \rightsquigarrow \ell_2 \in \mathcal{G}_P$, such that the first edges of these paths are different and they do not correspond to mutually exclusive MHP atoms, i.e., they use different future variables and do not correspond to mutually exclusive calling sites (see Section 2.1.2). Edges with multiplicity $i > 1$ represent $i$ different edges.

The first (resp. second) case is called direct (resp. indirect) MHP.

**Example 2.4.** *Let us explain some of the MHP pairs induced by $\mathcal{G}_{main_1}$ of Figure 2.2. Since $6 \rightsquigarrow 11$ and $6 \rightsquigarrow 15$, we conclude that (6,11) and (6,15) are direct MHP pairs. Moreover, since these paths originate in the same node 6, and the first edges use different future variables, we conclude that (11,15) is an indirect MHP pair. To see the improvement w.r.t. to the original MHP analysis note that node 8 does not have an edge to $\texttt{T}(\mathsf{f}, \texttt{act})$, since our MHF analysis infers that $\mathsf{x}$ is finished at L8. The original MHP analysis would*

*have an edge to* $\mathtt{T}(f, \mathtt{act})$ *instead of* $\mathtt{T}(f, \mathtt{fin})$*, and thus it produces spurious pairs such as (8,11).*

*Now consider nodes 11 and 17, and note that we have* $6 \rightsquigarrow 11$ *and* $6 \rightsquigarrow 17$*, and moreover these paths use different future variables. Thus, we conclude that (11,17) is an indirect MHP pair. However, carefully looking at the program we can see that this is a spurious pair, because* x *(to which task* f *is bound) is passed to method* g*, as parameter* w*, and* w *is guaranteed to finish when executing* **await** *w? at L*16*.*

The spurious pairs in the above example show that even if we used our improved LMHP analysis when constructing the MHP graph, using the procedure of the original MHP analysis to extract MHP pairs might produce spurious pairs. Next we handle this imprecision, by modifying the process of extracting the MHP pairs to have an extra condition to eliminate such spurious MHP pairs. This condition is based on identifying, for a given path $\mathtt{T}(m, X) \rightsquigarrow \ell$ in the graph of a program under analysis, which of the parameters of $m$ are guaranteed to finish before reaching $\ell$, and thus, any task that is passed to $m$ in those parameters cannot execute in parallel with $\ell$.

Intuitively, a future variable passed as parameter to a task is *not alive* along a path $p$ if the MHF analysis infers that the task bound to this future variable is finished at some point in $p$. Thus, any task bound to the future variable cannot execute in parallel with the corresponding program point.

**Example 2.5.** *Consider* $p \equiv \mathtt{T}(g, \mathtt{act}) \rightsquigarrow 17$*, then* w *is not alive along* p *since it is a path that consists of a single edge and* w *is in the MHF set associated with L17.*

The notion of "*not alive along a path*" can be used to eliminate spurious MHP pairs as follows.

**Example 2.6.** *We reconsider the spurious indirect MHP pairs of Ex. 2.4. Consider first (11,17), which originates from*

$$p_1 \equiv 6 \xrightarrow{1:x:3:[]} \mathtt{T}(f, \mathtt{act}) \rightsquigarrow 11 \text{ and } p_2 \equiv 6 \xrightarrow{1:z:5:[x]} \mathtt{T}(g, \mathtt{act}) \rightsquigarrow 17.$$

*We have that* x *is in the MHF set at L17, thus* $p_1$ *and* $p_2$ *are mutually exclusive and we eliminate this pair.*

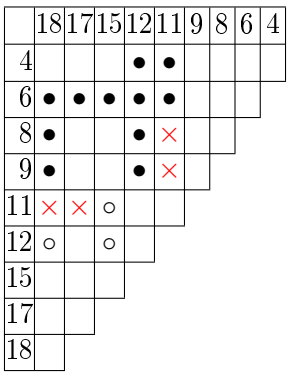



Figure 2.3: MHP pairs from $\mathsf{main}_1$

**Example 2.7.** *In Figure 2.3 all MHP pairs from method* $\mathsf{main}_1$ *are shown. In the table we can distinguish between different types of pairs. If the cell that connects two nodes is marked with* •*, it indicates that the pair is a direct MHP. Cells marked with* ∘ *indicate that the pair is an indirect MHP. Cells marked with* ×*represent spurious pairs that the original MHP analysis will infer.*

## 2.2 MHP Analysis with Returned Futures

In this section we present a MHP analysis with returned future variables [23], which modifies the local and global phases of the original analysis [21] to track the returned futures and backpropagates the new relations between the tasks. These modifications are described in Sections 2.2.1 and 2.2.2 respectively (see Section 4 in [23] for technical details).

```
1  main2() {
2    x=o1!g();
3    skip;
4    await x?;
5    z = x.get;
6    await z?;
7    skip;
8  }

 9  g() {
10    skip;
11    y = o2!f();
12    w = o3!q();
13    skip;
14    return y;
15  }

16  f() {
17    skip;
18  }
19
20
21  q() {
22    skip;
23  }
```

Figure 2.4: Example for MHP analysis with returned futures.

Figure 2.4 shows a simple program that has four methods $\mathsf{main}_2$, g, f and q. As before, the **skip** statement abstracts all instructions that do not affect the concurrency of the program. Method g spawns task f which is bound to future variable y at program point 11. At program point 12, g spawns a new task q, which is running in parallel with f in a different object. Finally, it returns the future variable y at program point 14. Method $\mathsf{main}_2$ is the starting one. It calls g asynchronously at program point 2 and gets blocked at program point 4 until g finishes its execution. At program point 5, the task bound to the future variable returned by the method bound to x is assigned to future z, i.e, task f, that is the one returned by g, is bound to the new future z. Finally, $\mathsf{main}_2$ gets blocked again at program point 6 waiting for the termination of f.

The original MHP analysis infers that task g finishes its execution at program point 4 but it is not able to infer that the task bound to future variable z at program point 5 is f and that f finishes its execution at program point 6. Thus, it infers that program point 6 of $\mathsf{main}_2$ can run in parallel with program point 17 of f though f has already terminated its execution.

### 2.2.1 Local MHP

As explained in Section 2.1.2, the local analysis is based on a transfer function. In this case, it is similar to the one presented on [21] except for the case of **get** statement which is novel to our extension. The LMHP atoms remain the same as those explained in Section 2.1.2.

**Example 2.8.** *In Figure 2.4 we have a method* g *with an instruction "***return** *x", and at the exit program point of* g *we have a LMHP state*

$$\Psi_0 = \{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\},$$

*which means that at the exit program point of* g *we have two active instances of methods* f *and* q*, bound to future variables* y *and* w *respectively. This means that* g *returns a future variable that is bound to an active instance of* f*. Now at program point 5, we have a state*

$$\Psi_1 = \{x{:}\mathtt{T}(g, \mathtt{fin})\},$$

*which means that before reaching the corresponding program point, we have invoked* g *and waited for it to finish (via future variable* x*). Let us now execute the instruction* $z = x.$**get** *in the context of* $\Psi_1$ *and generate a new LMHP state* $\Psi_2$*. Since* x *is bound to a task that is an instance of* g*,* $\Psi_2$ *should include an atom representing that* z *is bound to an active task which is an instance of* f *(which is returned by* g *via a future variable). Having this information in* $\Psi_2$ *allows us to mark* f *as finished when executing* **await** z? *later. We do this as follows:*

- *any MHP atom from* $\Psi_1$ *that does not involve* z *or* x *is copied to* $\Psi_2$*.*
- *any MHP atom from* $\Psi_1$ *that involves* z *is copied to* $\Psi_2$ *but with* z *renamed to* $\star$ *because* z *is overwritten.*
- *we transfer the atom* $x{:}\mathtt{T}(f, \mathtt{act})$ *from* $\Psi_0$ *to* $\Psi_2$*, by adding* $z{:}\mathtt{T}(f, \mathtt{act})$ *to* $\Psi_2$ *since now the corresponding task is bound to* z *as well.*
- *the atom* $x{:}\mathtt{T}(g, \mathtt{fin})$ *must be copied to* $\Psi_2$ *as well, but we first rewrite it to* $x{:}\mathtt{T}(g, \overline{\mathtt{fin}})$ *(in* $\Psi_2$*) to indicate that we have incorporated the information from the exit program point of* g *already. This is important because after executing the* **get***, we will have two instances of* f *in* $\Psi_0$ *and* $\Psi_2$ *that refer to the same task, and we want to avoid considering them as two different ones in the global phase that we will describe in the next section.*

*This results in*

$$\Psi_2 = \{x{:}\mathtt{T}(g, \overline{\mathtt{fin}}), z{:}\mathtt{T}(f, \mathtt{act})\}.$$

To summarize the above example, the local phase of our analysis extends that of [21] in two ways: it introduces a new kind of LMHP atom; and it has to treat the **get** instruction in a special way. Assume that method m is being analyzed. To incorporate this new information to the transfer function, the analysis computes a set of MHP atoms at the exit program point of method m, that are bound to a future variable that is returned by method m.

Due to the new case added, we need to modify the work-flow of the corresponding data-flow analysis in order to backpropagate the information learned from the returned future variables. This is because the LMHP analysis of one method depends on the LMHP states of other methods. This means that a method cannot be analyzed independently from the others as in [21, 22], but rather we have to iterate over their analysis results, in the reverse topological order induced by the corresponding call graph, until their corresponding results stabilize.

**Example 2.9.** *The left column of the table below shows the LMHP states resulting from applying once the transfer function of the local analysis to selected program points, the right column shows the result after one iteration of the transfer function over the results in the left column:*

| | |
|---|---|
| $L2 : \{\}$ | $L2 : \{\}$ |
| $L3 : \{x{:}\mathtt{T}(g, \mathtt{act})\}$ | $L3 : \{x{:}\mathtt{T}(g, \mathtt{act})\}$ |
| $L4 : \{x{:}\mathtt{T}(g, \mathtt{act})\}$ | $L4 : \{x{:}\mathtt{T}(g, \mathtt{act})\}$ |
| $L5 : \{x{:}\mathtt{T}(g, \mathtt{fin})\}$ | $L5 : \{x{:}\mathtt{T}(g, \mathtt{fin})\}$ |
| $L6 : \tau(z = x.\mathbf{get}, L5)$ | $L6 : \{x{:}\mathtt{T}(g, \overline{\mathtt{fin}}), z{:}\mathtt{T}(f, \mathtt{act})\}$ |
| $L7 : \tau(\mathbf{await}\ z?, L6)$ | $L7 : \{x{:}\mathtt{T}(g, \overline{\mathtt{fin}}), z{:}\mathtt{T}(f, \mathtt{fin})\}$ |
| $L8 : L7$ | $L8 : \{x{:}\mathtt{T}(g, \overline{\mathtt{fin}}), z{:}\mathtt{T}(f, \mathtt{fin})\}$ |
| $L11$: $\{\}$ | $L11$: $\{\}$ |
| $L12$: $\{y{:}\mathtt{T}(f, \mathtt{act})\}$ | $L12$: $\{y{:}\mathtt{T}(f, \mathtt{act})\}$ |
| $L13$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ | $L13$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ |
| $L14$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ | $L14$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ |
| $L15$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ | $L15$: $\{y{:}\mathtt{T}(f, \mathtt{act}), w{:}\mathtt{T}(q, \mathtt{act})\}$ |
| $L17$: $\{\}$ | $L17$: $\{\}$ |
| $L18$: $\{\}$ | $L18$: $\{\}$ |
| $L22$: $\{\}$ | $L22$: $\{\}$ |
| $L23$: $\{\}$ | $L23$: $\{\}$ |

*Let us explain some of the above LMHP states. In the left column, $L4$ corresponds to the state when reaching program point L4, i.e., before executing the statement* **await** $x?$*. It includes $x{:}\mathtt{T}(g, \mathtt{act})$ for the active task invoked at L2. The state $L4$ includes the finished task corresponding to the* ***await*** *instruction of the previous program point. $L6$ cannot be solved, as we need the information from state $L15$, which has not been computed yet. Something similar happens with the state $L7$, which cannot be calculated as the state $L6$ has not been totally computed. Atoms $y{:}\mathtt{T}(f, \mathtt{act})$ and $w{:}\mathtt{T}(q, \mathtt{act})$ appear in state $L14$ for the active tasks invoked at L11 and L12.*

*In the right column, after one iteration, we observe that most states are not modified except for $L6$, $L7$ and $L8$. Now that $L15$ is computed, we can calculate the LMHP state $L6$ adding the atom $z{:}\mathtt{T}(f, \mathtt{act})$ and updating the state of $g$. Having $L6$ calculated, $L7$ is computed modifying the state of $f$ to finished.*

### 2.2.2 Global MHP

In this section we describe how to use the LMHP information, inferred by the local phase, in order to construct an MHP graph from which an over-approximation of the set of MHP pairs can be extracted. The construction of the MHP graph is different from the one of the original MHP analysis [21] in that we need to introduce new kind of nodes to reflect the information carried by the new kind of MHP atom $y{:}\mathtt{T}(m, \overline{\mathtt{fin}})$. However, the procedure for computing the MHP pairs from the MHP graph is the same.

The *MHP graph* of a given program $P$ is denoted by $\mathcal{G}_P$. We include the following nodes to the ones explained in Section 2.1.3: (i) the node $\overline{\mathtt{fin}}(m)$ to represent the new MHP atom added to the local phase; (ii) each program point $\ell$ that is an exit program point, of some method $m$, contributes a node $\bar{\ell}$ and (iii) each future variable contributes a node $\ell_y$. Note that nodes $\overline{\mathtt{fin}}(m)$ and $\bar{\ell}$ are particular to our extension and will be used, as we will see later, to avoid duplicating tasks that are returned to some calling context.

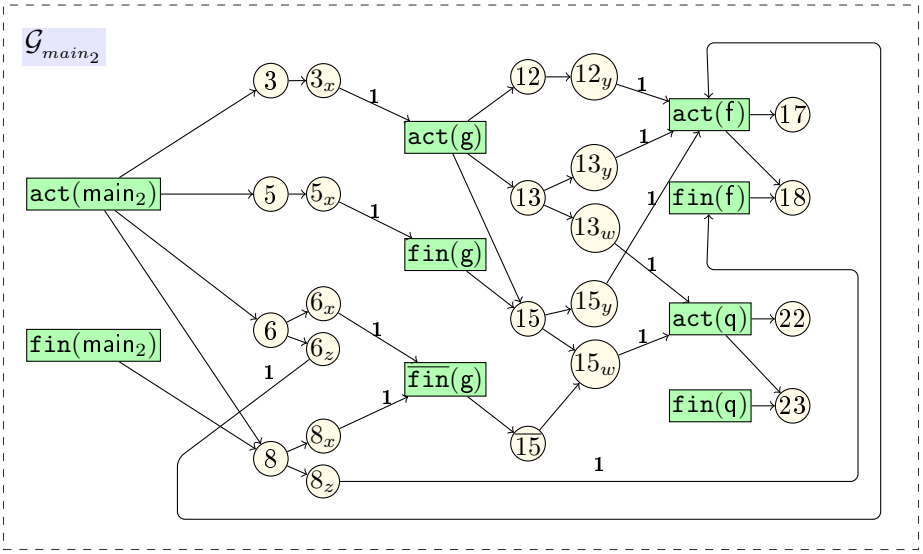



Figure 2.5: MHP graph $\mathcal{G}_{main_2}$ corresponds to analyzing $\mathsf{main}_2$.

The edges of $\mathcal{G}_P$ are constructed as in the original MHP analysis except for the new nodes. There is an edge from $\overline{\texttt{fin}}(m)$ to the corresponding *return node* $\bar{\ell}$, i.e., $\ell$ here is the exit program point of $m$. In addition, if $\ell$ is an exit program point and $y$ is not a returned future we add an edge from node $\bar{\ell}$ to node $\ell_y$. Note that when $\ell$ is an exit program point, the difference between node $\ell$ and $\bar{\ell}$ is that the later ignores tasks that were returned via future variables. As explained in the previous section, when a task returns a future variable, the MHP atom that represents the state of the task returned will appear at least in two LMHP states. Thanks to the new nodes and edges in the graph, we avoid to consider them as different instances of the same method.

**Example 2.10.** *Figure 2.5 shows the MHP graph for some program points of interest for our running example. Note that the out-going edges of program point nodes in* $\mathcal{G}_{main_2}$ *coincide with the LMHP states at these program points depicted in Example 2.9. Note that we have two nodes 15 and* $\overline{15}$ *to represent the exit program point L15, connected to* $\texttt{fin}(g)$ *and* $\overline{\texttt{fin}}(g)$*. The edges that go out from 15 correspond to the atoms in* $L15$*. As L15 is the exit program point of method* g*, we have to build an edge. This edge goes from* $\overline{15}$ *to* $15_w$ *and from there to* $\texttt{act}(q)$ *and corresponds to the atom in* $L15$ *whose future variable is not returned by* g*.*

Given $\mathcal{G}_P$, using the same procedure as in the original MHP analysis, we say that two program points $\ell_1, \ell_2$ may run in parallel if one of the following conditions hold:

1. there is a non-empty path from $\ell_1$ to $\ell_2$ or vice versa; or
2. there is a program point $\ell_3$ and non-empty paths from $\ell_3$ to $\ell_1$ and from $\ell_3$ to $\ell_2$ such that the first edge is different, or they share the first edge but it has weight $i > 1$.

| | 23 | 22 | 18 | 17 | 15 | 13 | 12 | 8 | 6 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | ● | ● | ● | ● | ● | ● | ● | | | | |
| 5 | ● | ● | ● | ● | ● | | | | | | |
| 6 | ● | ● | ● | ● | ● | | | | | | |
| 8 | ● | ● | ● | × | | | | | | | |
| 12 | | | ● | ● | | | | | | | |
| 13 | ● | ● | ● | ● | | | | | | | |
| 15 | ● | ● | ● | ● | | | | | | | |
| 17 | ○ | ○ | × | × | | | | | | | |
| 18 | ○ | ○ | × | | | | | | | | |
| 22 | | | | | | | | | | | |
| 23 | | | | | | | | | | | |

Figure 2.6: MHP pairs from $\mathsf{main}_2$.

**Example 2.11.** *Let us explain some of the MHP pairs induced by* $\mathcal{G}_{main_2}$*. (14,17) and (14,22) are direct MHP pairs as we can find the paths* $14 \rightsquigarrow 17$ *and* $14 \rightsquigarrow 22$ *in* $\mathcal{G}_{main_2}$*. In addition, as the first edge is different, we can conclude that (17,22) is an indirect pair. In contrast to the graph that would be obtained for the original analysis, (7,17) is not an MHP pair.*

*Instead of it, we have the path* $7 \rightsquigarrow 18$ *which indicates that the task* $f$ *is finished.*

**Example 2.12.** *In Figure 2.6 the MHP pairs from method* $\mathsf{main}_2$ *of Figure 2.4 are shown. In this table we can distinguish between different types of pairs. If the cell that connects two nodes is marked with* ●*, it indicates that the pair is a direct MHP, if it is marked with* ○ *indicates that the pair is an indirect MHP and if it is marked with* × *indicates that the pair is spurious and would be inferred by the original MHP analysis if the new nodes are not included.*

## 2.3 Related Work

The closest approach to the original MHP analysis [21] is an analysis developed for X10 in [66, 12, 67, 85]. X10 uses an *async-finish* concurrency model. The *async* construct allows forking a process in a new thread and the *finish* primitive waits until the execution of all the tasks called in this scope finish before the execution continues. Note that the async-finish model simplifies the inference of the status of the task. Namely, it is easier to infer the *escaped* tasks, i.e, those that are still running though the tasks that have spawned them asynchronously finished their execution, since the finish construct ensures that all methods called within its scope terminate before the execution continues to the next instruction. [66] presents a MHP analysis specified with a type system. They define a set of type rules to represent the program. The type inference is done in two steps. First they rephrase the type inference problem as a constraint problem and afterwards they solve the constraint system obtained using algorithms for iterative data-flow analysis. In [12], the authors propose a MHP analysis in two steps. First, it performs a *Never-Execute-in-Parallel* (NEP) analysis (complement of MHP) to discard some pairs. Then,

a *Place-Equivalent* analysis is performed to know if all instances of two statements are guaranteed to execute at the same places. Finally, both analyses are combined to obtain MHP information. To compute that, they use a program structure tree to represent each procedure. In contrast, we use a graph that represents the whole program analyzed. [67] improves the analysis presented in [66]. It presents a MHP analysis of a storeless model of X10. The approach focuses on answering two closely related problems: the MHP decision problem and the MHP computation problem. The first one will be used to answer the second. To solve the first problem they use a reduction to *constrained dynamic pushdown networks* (CDPNs). CDPN models collections of sequential pushdown processes running in parallel. They give a translation from programs in X10 to CDPNs and the MHP decision problem is solved by performing a reachability test. To solve the MHP computation problem a type-based analysis that produces a set of candidate pairs is developed. The type analysis problem is reformulated as a constraint solving problem. Once this set has been created, the CDPN-based decision procedure is used for each of the candidate pairs in order to remove those that cannot happen in parallel. [85] improves the analysis presented in [12] by proposing an incremental MHP analysis based on the tree representation of the program. It starts with a sequential version of the program, removing the concurrent instructions. Afterwards, it re-introduces these elements in a particular order updating the MHP pairs. The main contribution is that the analysis proposed avoids computing the Never-Execute-in-Parallel analysis of [12]. Instead, the analysis computes the MHP relation directly, gaining efficiency.

An MHP analysis for Ada has been presented in [79], and extended later for Java in [80]. These works have been superseded later by [68, 28, 37]. [79] proposes an MHP analysis for Ada. Ada has a *rendezvous* concurrency model based on synchronous message passing, which is very different to the use of future variables. [80] presents a data flow analysis that uses a *parallel execution graph* (PEG). The PEG represents the control-flow graph of each thread created during the execution of the program with additional edges at the synchronization program points. One of the limitations of [79, 80, 68] is that they assume that the number of tasks/threads is bounded and known at compilation time. [28] proposes a non-iterative analysis that uses an intra-procedural control-flow graph to represent the execution of each thread. In this work, Java programs are abstracted to an *abstract thread model* which is then analyzed in two phases. First, it generates the MHP graphs for each thread of the program. Afterwards, it computes a must-join relation and builds a thread creation tree with this new information that contains the global information of the program. A main difference is that our first phase infers local information for each method, while that of [28] infers a thread-level MHP from which it is possible to know which threads might *globally* run in parallel. In addition, unlike our local analysis, it does not consider any synchronization between the threads in the first phase, but rather in the second phase. In [37] an iterative analysis based on a model similar to the previous one is proposed. In this case, they use a graph-based program representation over which they define a set of data flow equations. Then, an iterative process is applied to these equations to obtain the pairs that may run in parallel.

[46] considers a fork-join semantics and operates over code regions rather than individual statements. This analysis uses a control flow graph and a *Happens-Before* analysis to infer the MHP information. It builds a control flow graph to obtain the dependencies between the different threads. This graph is used to perform a Happens-Before analysis

obtaining thread-order properties, which are used to define regions formed by statements that share the same properties. Finally, the analysis constructs a region relation graph, from which it infers the MHP pairs.

In [71, 69], an asynchronous semantics is defined for JavaScript programs. These works formalize an asynchronous execution model to infer fundamental relationships between asynchronous events of the program. They defined a *priority promise* which are promise variables, similar to future variables, that have the same semantic of JavaScript promises and are enriched with a priority value. They defined a *happens-before* relation to infer the order in which the different events are executed in a trace. In [71] the authors compute *event call-graphs* that connect the call sites with the function declarations and include the happens before relations. Event call-graphs enables discovering potential bugs of the program such as unreachable functions or mismatches between asynchronous and synchronous calls. [69] uses the happens-before relation to build a data-race detector. The analysis defines two sets of rules and computes *linking* and *causal* contexts to represent the relations between the promise variables and their synchronization and their continuation. These contexts are represented using trees that are used to find the data-races.

[40] builds a time based model to infer race conditions in high performance systems. The analysis proposed relies on a MHP analysis which is applied in a segment graph. Instead of inferring the pairs of statements that may run in parallel, they suppose initially that all statements can happen in parallel with each other. Iteratively, the analysis eliminates the spurious ones. [40] is extended in [36], using a model checker to solve the MHP decision problem. First, the analysis defines an automaton for each behavior in the system. Then, it creates the properties that are going to be processed by the model checker. In addition, the analysis reduces the number of queries that the model checker has to compute considering only those program points that affect the concurrency, as our MHP analyses do.

Other analyses for more complex properties can greatly benefit from the MHP pairs that our MHP analyses infer. Several proposals for deadlock analysis [77, 49] rely on the MHP pairs to discard unfeasible deadlocks when the instructions involved in a possible deadlock cycle cannot happen in parallel. In [20], the MHP analysis also plays a fundamental role to increase the accuracy of termination and resource analysis.

All the analyses described before do not handle inter-procedural synchronization or are imprecise when future variables or tasks identifiers are passed as parameters to the methods and awaited in an inner scope, or returned by methods and awaited in an outer one.

Our solutions for the forward and backwards inference, described in Section 2.1 and Section 2.2 respectively (formalized in [22] and [23] respectively), are technically different, but fully compatible. Essentially, they only have in common that both the local and global analysis phases need to be changed. For the forward inference, the analysis includes a separated *must-have-finished* (MHF) pre-analysis that allows inferring, for each program point, which tasks (both the tasks spawned locally and the passed as arguments) have finished their execution when reaching that program point.

In contrast, for the backwards inference, the local phase itself has to be extended to propagate backwards the new relations created when a future variable is returned, which requires changing the analysis flow.

In both analyses, the creation of the graph needs to be modified to reflect the new

information inferred by the respective local phases, but in each case is different.

For the forward inference, the way in which the MHP pairs are inferred besides has to be modified as described in Section 2.1.2. However, both extensions are compatible, and together provide a full treatment of future variables in the MHP analysis.

# Chapter 3

# EthIR: A Framework for High-Level Analysis of Ethereum Bytecode

Once a smart contract is deployed in the blockchain, its Ethereum bytecode (EVM) is inmutable. Thus, it is fundamental to find security bugs and unwanted behaviors before they are deployed. Several analysis tools have been developed for analyzing either the original source code or the EVM bytecode of the contract. Analyzing the Solidity may benefit of the high-level representation of the code, as it has the explicit representation of data structures. However it has much more limited applicability because most of the smart contracts do not have their source code stored in the blockchain of Ethereum. Analyzing EVM bytecode is a necessity when: (1) the source code is not available (e.g., the blockchain only stores the bytecode), (2) the information to be gathered in the analysis is only visible at the level of bytecode (e.g., gas consumption is specified at the level of EVM instructions), (3) the analysis results may be affected by optimizations performed by the compiler (thus the analysis should be done ideally *after* compilation).

The purpose of decompilation, as for other bytecode languages [92], is to make explicit in a higher-level representation the *control flow* of the program (by means of rules which indicate the continuation of the execution) and the *data flow* (by means of explicit variables, which represent the data stored in the stack, in contract fields, in local variables, and in the blockchain), so that an analysis can have this information directly available.

This chapter presents EthIR [24, 25], a framework that generates an *intermediate representation* for EVM bytecode. The intermediate representation allows us to analyze the bytecode of a smart contract using a high-level representation instead of the low-level one provided by the bytecode. This intermediate representation is formed by guarded rules that maintain the data- and control-flow of the original bytecode. The translation is done in two main steps (see Section 2 in [24] and Sections 2.1 and 2.2 in [25] for technical details):

1. EthIR takes a smart contract written in Solidity, EVM bytecode or disassembled code and generates the control-flow graph (CFG) of the contract. EthIR uses a modified version of tool Oyente [70, 9] to build the CFG.

2. EthIR translates the CFG into a *Rule-based Representation* (RBR). Each block of the CFG corresponds to a set of rules depending on the type of the block (conditional or unconditional). The rules may contain a guard to define the conditions for their

applicability. The RBR makes the stack explicit and it also has a representation of the contract state variables, memory and blockchain data.

As OYENTE's goal is symbolic execution for the sake of bug detection, rather than the generation of a complete CFG, the method proposed in [24] returns an incomplete CFG. In [25], we improve the generation of the CFG and modified OYENTE to avoid the source of incompleteness.

ETHIR is open-source, it is implemented in Python and is available at `https://github.com/costa-group/EthIR`.

## 3.1 From EVM Bytecode to a Control-Flow Graph

Given some `EVM` code, the OYENTE tool generates a set of blocks that store the information needed to represent the CFG of such `EVM` code. The original version of OYENTE does not generate complete CFGs. In order to fix this issue, the following extensions have been needed:

1. To recover the list of addresses for unconditional blocks with more than one possible jump address (as OYENTE originally only kept the last processed one) due to different calling points.

2. To add more explicit information to the CFG: jump operations are decorated with the jumping address, discovered by OYENTE, and other operations such as `SSTORE`, `MSTORE`, `SLOAD` or `MLOAD` are also decorated with the address they operate: the number of state variable for operations on storage; and the memory location for operations on memory if OYENTE is able to discover it. These annotations cannot be generated when the address is not statically known, (for instance, when we have an array access inside a loop with a variable index). In such cases, we annotate the corresponding instructions with "?".

3. To remove the execution bound (as well as other checks that were only used for their particular applications), and add information to the path under analysis. Namely, every time a new `JUMP` command is found, we check if the jumping point is already present in the path. In such case, an edge to that point is added and the exploration of the trace is stopped. As a side effect, we not only produce a complete CFG, but also avoid much useless exploration for our purposes which results in important efficiency gain.

As an example, Figure 3.1 shows the Solidity source code for a fragment of `EthereumPot` contract explained in Section 1.3 (left), and the corresponding `EVM` disassembled code (right). In Figure 3.2, part of the CFG associated to the `__callback` method is displayed. The call from `__callback` to `findWinner` happens in `block2369`. Observe that, instead of using a `CALL` opcode, there is a `PUSH` of the return address `0x0954` (2388) and a `JUMP` to `block1611` where the code of `findWinner` starts. The decorations mentioned in point (2) above can be observed e.g., in `block1619` or `block1647`, where `SLOAD` and `MSTORE` opcodes are labeled, resp., with a 3, and with 0 and "?".

```
1  contract EthereumPot {
2    address[] public addresses;
3    address public winnerAddress;
4    uint[] public slots;
5    function __callback (bytes32 _queryId, string _result, bytes _proof)
6    oraclize_randomDS_proofVerify (_queryId, _result, _proof) {
7      if (msg.sender != oraclize_cbAddress()) throw;
8      random_number = uint(sha3(_result));
9      winnerAddress = findWinner(random_number);
10     amountWon = this.balance * 98 / 100 ;
11     winnerAnnounced(winnerAddress, amountWon);
12     if (winnerAddress.send(amountWon)) {
13      if (owner.send(this.balance)) {
14        openPot();
15      }
16    }
17   }
18
19   function findWinner (uint random) constant returns (address winner) {
20     for (uint i = 0; i < slots.length; i++) {
21       if (random <= slots[i]) {
22         return addresses[i];
23       }
24     }
25   }
26   // Other functions
27 }
```

```
28 ···
29 DUP1
30 PUSH1 => 0x00
31 SWAP1
32 POP
33 PUSH1 => 0x03
34 DUP1
35 SLOAD
36 SWAP1
37 ···
38 PUSH1 => 0x40
39 MLOAD
40 DUP1
41 SWAP2
42 SUB
43 SWAP1
44 SHA3
45 PUSH1 => 0x01
46 ···
47 JUMPDEST
48 MOD
49 ADD
50 PUSH1 => 0x0a
51 DUP2
52 SWAP1
53 SSTORE
54 POP
55 PUSH2 => 0
       x0954
56 PUSH1 => 0x0a
57 SLOAD
58 PUSH2 => 0
       x064b
59 JUMP
60 ···
```

Figure 3.1: Excerpt of Solidity code for `EthereumPot` contract (left), and fragment of `EVM` code for function `__callback` (right).

Finally, when we have Solidity code available, we are able to retrieve the name of the functions invoked from the hash codes. This allows us to statically know the continuation block and analyze each public function independently.

## 3.2 From Control-Flow Graph to a Rule-based Representation

EthIR provides a Rule-Based Representation (RBR) for the CFG obtained. EthIR has been implemented as a standalone tool to facilitate its integration into other tools in the future. Intuitively, for each block in the CFG it generates a corresponding rule that contains a high-level representation of all bytecode instructions in the block (*e.g.*, load and store operations are represented as assignments) and that has as parameters an explicit representation of the stack, local, state, and blockchain variables. Conditional branching in the CFG is represented by means of guards in the rules. The identifiers given to the rules *–block_ x* or *jump_ x–* use $x$, the PC of the first bytecode in the block being translated. We distinguish among three cases: (1) if the last bytecode in the block is an unconditional jump (`JUMP`), we generate a single rule, with an invocation to the continuation block,

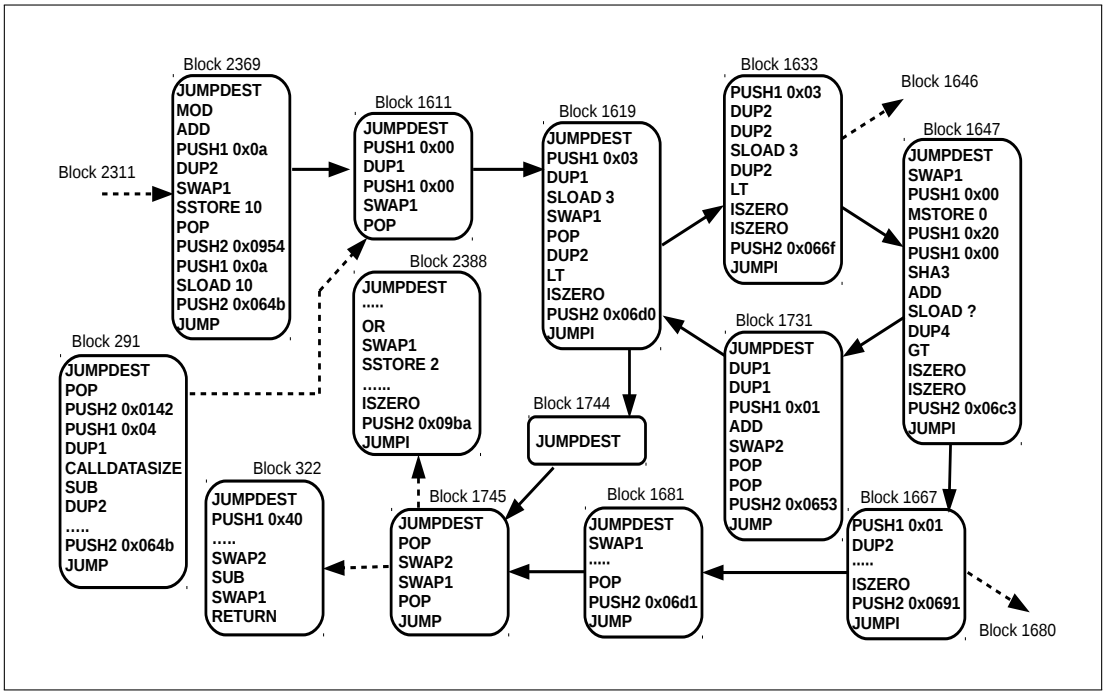



Figure 3.2: Fragment of CFG generated by EthIR for `__callback`

(2) if it is a conditional jump (`JUMPI`) we produce two additional *guarded* rules which represent the continuation when the condition holds, and when it does not, (3) otherwise, we continue the execution in block $x{+}s$ (where $s$ is the size of the `EVM` bytecodes in the block being translated). As regards the variables, we distinguish the following types:

- *Stack variables*: a key ingredient of the translation is that the stack is flattened into variables, *i.e.*, the part of the stack that the block is using is represented, when it is translated into a rule, by the explicit variables $s_0, s_1, \ldots$, where $s_1$ is above $s_0$, and so on. The initial stack variables are obtained as parameters $s_0, s_1, \ldots, s_n$.

- *Local variables*: the content of the local memory in numeric addresses appearing in the code, which are accessed through `MSTORE` and `MLOAD` with the given address $i$, are modeled with variable $l_i$ where $i$ is the address involved in the operation. The set of local variables involved in each rule is passed as parameters. For the translation, we assume we are given a map which associates a different local variable to every numeric address memory used in the code. When the address is not numeric, we represent it using a fresh variable local to the rule to indicate that we do not have information on this memory location.

- *State variables*: we model the contract state variables by means of variables named $g_0, \ldots, g_k$. The set of state variables involved in each rule is passed as parameters. Since these state variables are accessed using `SSTORE` and `SLOAD` using the number of the state variable, we associate $g_i$ to the $i$th state variable. As for the local memory, if the number of the state variable is not numeric because it is unknown (annotated

$block1611(\overline{s_6}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$\ldots$
$call(block1619(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}))$

$block1619(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$\ldots$
$call(jump1619(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}))$

$jump1619(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$geq(s_{10}, s_9),$
$call(block1744(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}))$

$jump1619(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$lt(s_{10}, s_9),$
$call(block1633(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}))$

$block1633(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$s_9 = 3, s_{10} = s_8, s_{11} = s_9, s_{11} = g_3, s_{12} = s_{10},$
$call(jump1633(\overline{s_{12}}, \overline{sv}, \overline{lv}, \overline{bc}))$

$jump1633(\overline{s_{12}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$lt(s_{12}, s_{11}),$
$call(block1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}))$

$block1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$s_{11} = s_9, s_9 = s_{10}, s_{10} = s_{11}, s_{11} = 0, l_2 = s_{10},$
$s_{10} = 32, s_{11} = 0, s_{10} = sha3(s_{11}, s_{10}),$
$s_9 = s_{10} + s_9, gl = s_9, s_9 = fresh_0,$
$s_{10} = s_6, call(jump1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}))$

$jump1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$gt(s_{10}, s_9),$
$call(block1731(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}))$

$block1731(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow$
$(see\ Figure 3.4)$
$call(block1619(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}))$

Figure 3.3: Fragment of the RBR for the call from `__callback` to `findWinner`.

as "?"), we use a fresh local variable to represent it.

- *Blockchain data*: we model this data with variables $\overline{bc}$, which are either indexed with $md_0, \ldots, md_q$ when they represent the message data, or with corresponding names, if they are precise information of the call, like the gas, which is accessed with the opcode `GAS`, or about the blockchain, like the current block number, which is accessed with the opcode `NUMBER`. All this data is accessed through dedicated opcodes, which may consume some offsets of the stack and normally place the result on top of the stack (some of them, like `CALLDATACOPY`, can besides store information in the local memory).

The translation uses two auxiliary functions to translate each bytecode into corresponding high-level instruction (and updates the size of the stack) and to translate the guard of a conditional jump.

**Example 3.1.** *Figure 3.3 shows the RBR for selected blocks in Figure 3.2. As can be seen, for instance, in rule* block1647*, bytecode instructions that load or store information are transformed into assignments on the involved variables. For arithmetic operations, operations on bits, sha, etc., the variables they operate on are made explicit. Since stack variables are always consecutive we denote by $\overline{s_n}$ the decreasing sequence of all $s_i$ from $n$ down to 0. $\overline{lv}$ includes $l_2$ and $l_0$, which is the subset of the local variables that are needed in this rule or in further calls (second extension described above). The unknown location "?" has become a fresh variable* $\text{fresh}_0$ *in* block1647*. For state variables, $\overline{sv}$ includes the needed ones $g_{11}, g_8, g_7, g_6, g_5, g_3, g_2, g_1, g_0$ ($g_i$ is the $i$-th state variable). Finally, $\overline{bc}$ includes the needed blockchain state variables* `address`*,* `balance` *and* `timestamp`*. The first extension described above is required in the example to connect* block2639 *(that contains the* `PUSH`*) with* block2388*, and the same for* block291 *with* block322*. Our solution is to make two clones of the 11 involved blocks in Figure 3.2 for each path. The rules in Figure 3.3 show only (a fragment of) one of the clones.*

Besides, [25] provides two extensions to the initial version of EthIR in [24] which are required for soundness and efficiency:

$$
\begin{aligned}
&block1731(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow \\
&\quad nop(JUMPDEST), s_9 = s_8, nop(DUP1), s_{10} = s_9, nop(DUP1), s_{11} = 1, \\
&\quad nop(PUSH1), s_{10} = s_{11} + s_{10}, nop(ADD), s_{11} = s_8, s_8 = s_{10}, s_{10} = s_{11}, \\
&\quad nop(SWAP2), nop(POP), nop(POP), s_9 = 3943, nop(PUSH2), \\
&\quad call(block1619(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc})), nop(JUMP)
\end{aligned}
$$

Figure 3.4: Selected rule including nop functions.

- The first extension is related to the way function calls are handled in the `EVM`, where instead of an explicit `CALL` opcode, as we have seen in the example before, a call to an internal function is transformed into a `PUSH` of the return address in the stack followed by a `JUMP` to the address where the code of the function starts. If the same function is called from different points of the program, the resulting CFG shares for all these calls the same subgraph (the one representing the code of the function) which ends with different jumping addresses at the end. As described in [52], there is a need to clone parts of the CFG to explicitly link the `PUSH` of the return address with the final `JUMP` to this address. This cloning in our implementation is done at the level of the RBR as follows: Since the jumping addresses are known thanks to the symbolic execution applied by Oyente, we can find the connection between the `PUSH` and the `JUMP` and clone the involved part of the RBR (between the rule of the `PUSH` and of the `JUMP`) using different rule names for each cloning.

- The second extension is a flow analysis intended to reduce the number of parameters of the rules of the RBR. This is crucial for efficiency as the number of involved parameters is a bottleneck for the successive analysis steps that we are applying. Basically, before starting the translation phase, we compute the inverse connected component for each block of the CFG, i.e, the set of its predecessor blocks. During the generation of each rule, we identify the local, state or blockchain variables that are used in the body of the rule. Then, these variables have to be passed as arguments only to those rules built from the blocks of its inverse connected component.

Optionally, EthIR provides in the RBR the original bytecode instructions (from which the higher-level ones are obtained) by simply wrapping them within a nop functor (see Figure 3.4). This is relevant for a gas analyzer to assign the precise gas consumption to the higher-level instruction in which the bytecode was transformed.

## 3.3 Related Work

In the past three years, several approaches tackled the challenge of fully formal reasoning about Ethereum contracts implemented directly in `EVM` bytecode by modeling its rigorous semantics in state-of-the art proof assistants [58, 73, 57, 53]. While those mechanisations enabled formal machine-assisted proofs of various safety and security properties of `EVM` contracts [58, 57, 53], none of them provided means for fully *automated* sound analysis of `EVM` bytecode.

In [58] a formalization of EVM bytecode in Lem [76] is presented. Lem is a language that can be compiled into Coq [42], Isabelle/HOL [82] and HOL4 [87]. This formalization over-approximates the semantics of EVM. First, the rules that represent actions of a smart contract when a transaction starts are defined such as the call environment, delegate calls or how the values are returned. After that, the definition of the rules for a deterministic semantics of the program are given. Finally the approach tests the specification defined using Isabelle/HOL, finding some differences between the specification [95] and the implementation of Ethereum.

[73] defines a semantics for smart contracts based on finite state machines. The authors present FSolidM, a framework for designing contracts as Finite State Machines (FSM). They propose a graphical interface that creates a smart contract as a FMS and generates its source code in Solidity automatically. To model the smart contracts as FMS, they identify states that may be modified due to the effect of actions invoked by other contracts or users through functions. They also use guards to define the applicability of the transitions between the different states of the contract. The paper argues that it is easier to codify a smart contract as a FSM, reducing the potential programming errors thanks to its well-defined semantics.

In [57] the tool KEVM is presented. It introduces a formal specification of EVM in the $\mathbb{K}$ framework [84], a rewriting based tool that defines executable semantics specifications. The aim of the $\mathbb{K}$ framework is to separate the construction of the analyses and the specification of the programming language. It generates a parser given the semantics and the syntax of EVM. The semantics is built defining rewrite rules for the types and bytecode instructions and configuration files for representing the states and the execution model.

Finally, [53, 54] define a formal small-step semantics for Ethereum bytecode using the F* proof assistant [89]. They present a formalization for the global state of the system in order to represent all the accounts available on it. It allows modeling the effect of exceptions. They also define a grammar for *call stacks* and *transaction environments* that defines the result of executing each of the bytecode instructions in the stack as well as transaction effects that collect the changes on the global state after the transaction finishes its execution. The semantics defined is used to find security bugs on smart contracts. The authors define the semantics characterization of some significant security properties on smart contracts to study them using the semantics proposed in the papers.

Concurrently, several other approaches for ensuring correctness and security of Ethereum contracts build an intermediate representation for smart contracts. Some tools instead, implement automated toolchains for detecting bugs by symbolically executing EVM bytecode directly.

[10] describes Rattle, a static analysis framework designed to work on deployed smart contracts. It takes the EVM bytecode of the contract and applies a data-flow analysis to recover a complete CFG of the contract. After that, it converts the CFG into a single-static-assignment (SSA) register form, and finally optimizes the SSA form removing the PUSH, DUP, SWAP and POP instructions. This optimization removes most of the EVM instructions from the original code, making the resulting representation more readable to the user. To generate the CFG, Rattle splits the EVM code into single blocks, as EthIR does, and after that builds the links between the blocks. The process to obtain the SSA register form of the contract is carried out iteratively as each block is translated independently. When the first iteration finishes, the values of non-resolved registers are propagated. Thanks to

the SSA form, Rattle is able to remove EVM bytecode, track variable definitions and uses, recover memory and storage locations and check conditions of external calls. However, some analyses may be affected due to the modifications of the original bytecode.

[75] presents Mythril, a security analysis tool for smart contracts. Mythril [8] decompiles EVM bytecode using the dynamic symbolic execution engine Laser-Ethereum [3]. It produces traces that are used to build an intermediate representation. Laser-Ethereum is a symbolic interpreter for EVM bytecode. It takes one or more contracts as input and returns a set of abstract program states representing the *world state* (a mapping between addresses to accounts), *the machine state* (program counter, memory and stack) and *execution environment* (implicit variables such as caller address or transaction values). Using these abstract states, it generates a CFG with edges labeled with constraints. Thus, Mythril represents the execution of a smart contract as a set of states and path formulas in propositional logic.

[33] presents Vandal, a tool that accepts EVM bytecode and decompiles it into a structured intermediate representation. Vandal consists of an analysis pipeline that converts EVM bytecode to logic relations that reflect the semantics of the contract. The translation is done in four main steps: (1) the *Scraper* traverses all transactions in all blocks of the live Ethereum blockchain searching for the contract creation and, when it is found, it stores the EVM bytecode of the contract that is going to be decompiled locally, (2) the *Disassembler* takes the sequence of hexadecimal numbers that represents the bytecode of the contract and returns a series of readable low-level instructions annotated with program counter addresses, (3) the *Decompiler* translates the disassembled bytecode into a register transfer language using a CFG and a data-flow analysis to expose data- and control-flow structures of the bytecode, and (4) the *Extractor* that translates the register language into logic relations expressed in files that can be read by analyses. [51] improves the decompilation process of [33]. The authors use Datalog [56], a declarative logical language, as specification language for the decompiler. The decompilation can be summarized in five steps: (1) it finds the basic blocks of the bytecode, (2) performs local analysis of stack effects of basic blocks (it introduces variables and performs a first value analysis), (3) analyzes the bytecode and builds a CFG to produce a three-address intermediate representation, (4) infers function boundaries (entry and exit blocks and function calls) heuristically that are used to split the global CFG into several local CFGs and a call graph, and (5) infers function arguments and return arguments for all functions to generate the intermediate representation of the bytecode.

[86] presents Scilla, an intermediate language for verified smart contracts. This work provides a translation from Solidity to Scilla. The authors define an automata-based model to represent smart contracts. It separates the communication aspects and the programming components of the smart contracts. The contract under analysis is translated into a communicating automata that reacts to actions. Each function of the contract leads to an atomic transition on the automata. The transitions are invoked by messages containing a tag to identify the transition and its arguments. Scilla is able to model external calls as *continuations*, a special kind of transitions that represent the code that is executed after an external call. However, it does not support loop constructs although the authors plan to include them via recursive function definitions. The semantics of Scilla can be specified in the Coq proof assistant [42] to study safety and liveness properties on smart contracts.

[93] presents IRSRI, an intermediate representation that is built following similar approaches to [10] and [51]. The underlying principle is to generate a CFG formed by basic blocks, use SSA to represent variables and three-address code (3AC) for statements. Basic blocks of the CFG keep continuations like [86], i.e., each block takes arguments and a pointer to the next block. The author proposes to use attribute grammars in order to build the intermediate representation. First it takes the abstract syntax tree (AST) of the Solidity code using the compiler of Solidity and builds the CFG using the attribute grammars defined. The paper specifies attribute grammar rules to translate the AST nodes into blocks, for handling variables and the statements.

[70] presents Oyente, a popular tool for analysis of Ethereum smart contracts [9]. Oyente takes a smart contract written in Solidity or its `EVM` bytecode and models the stack, the memory and the storage to symbolically execute the contract. However it does not implement all the instructions available in `EVM`. It uses Z3 [45] to decide if an execution path is unfeasible and abort the exploration of the path under analysis. The representation of Oyente is too low-level to implement analyses of high-level properties, *e.g.*, loop complexity or commutative conditions.

Zeus [63], a tool for analyzing Ethereum smart contracts via symbolic execution *wrt.* client-provided *policies* where some conditions to be checked are defined. In contrast to EthIR, it operates directly on Solidity source code. In addition, it is not fully automatic as the users have to define the policies. The authors define an abstract language to translate the source code of the contracts. Finally, it generates LLVM bytecode from the abstract language presented in the paper. Soundness of Zeus depends on the semantics of Solidity, which is not formally defined.

All the tools mentioned above generate an intermediate representation to analyze either the Solidity source code or the `EVM` bytecode of a smart contract. Most of them generate a CFG to represent the control-flow of the contract and use it to build an intermediate representation of it. They build several analyses on top of their intermediate representation to study different properties of the contract and find vulnerabilities. In [24] we propose an intermediate representation of the `EVM` bytecode of a smart contract that makes explicit the stack, blockchain data, the memory and the storage. Our intermediate representation can be used by the different analyses developed with minor changes. The intermediate representation generated by [51] is the most similar to the one built by EthIR. The main difference is that [51] implements a data-flow analysis to propagate constants and expressions. Hence, it is able to generate more complex expressions, rather than binary assignments as EthIR does. However, the expressions are equivalent. A unique feature of EthIR (that none of the intermediate representations has) is that we maintain the original bytecode decompiled code. This information could be essential to develop more complex analyses as we will see in Chapter 5.

# Chapter 4

# SAFEVM: A Verifier for Ethereum Smart Contracts

The Solidity language contains the verification-oriented functions `assert` and `require` to check for safety conditions or requirements. If they do not hold, the execution of the transaction terminates. Namely, when the Solidity code is compiled into `EVM` bytecode, the `require` condition is transformed into a test that checks the condition and invokes a `REVERT` bytecode if it does not hold. `REVERT` aborts the whole execution of the smart contract, reverts the state and all remaining gas is refunded to the caller. On the other hand, `assert` checks the condition and invokes an `INVALID` bytecode if it does not hold. When executing `INVALID`, the state is reverted but *no gas is refunded* and hence, it has more serious consequences than `REVERT`: besides the economic consequences of losing the gas, the only information given to the transaction is an out-of-gas error message.

This chapter presents SAFEVM [17], a verifier for smart contracts that uses verification engines for C programs. The analysis is applied in two steps (see Section 2 in [17] for technical details):

- SAFEVM generates a C program from the RBR built by EthIR. It translates the recursive guarded rules generated by EthIR into an iterative C program that abstracts the contract.

- The program generated is analyzed using a verifier for C programs. The verifiers try to prove the reachability of the bytecode instruction `INVALID` and generate a verification report. SAFEVM has been tested with CPAchecker [29], VeryMax [35] and SeaHorn [62] as backend C verifiers.

SAFEVM has a very large (potential) user base, as Ethereum is currently the most advanced platform for coding and processing smart contracts with more than 50,000 contracts available. We have automatically verified 6,301 real Solidity files, which contain `INVALID` instructions in their `EVM` code, pulled from the Ethereum blockchain. These files contain 24,294 smart contracts with 44,046 public functions that may lead to an `INVALID` instruction and 177,549 `INVALID`-free functions. From these functions, SAFEVM verifies automatically safety of around 20% of the functions (see Section 3 in [17] for more details).

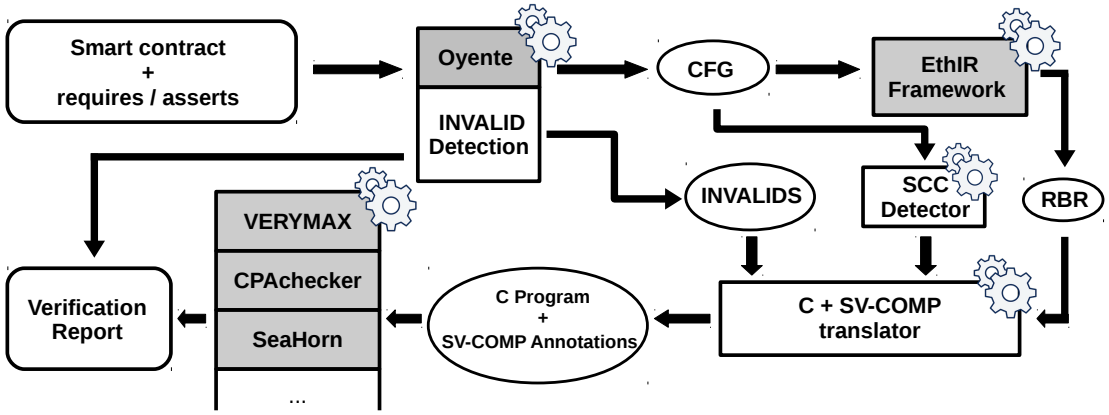



Figure 4.1: SAFEVM's architecture

## 4.1 Architecture of SAFEVM

Figure 4.1 depicts the main components of SAFEVM that are as follows (shaded boxes are off-the-shelf used systems not developed by us):

1. *Input.* SAFEVM takes a smart contract, optionally with `assert` and `require` verification annotations. The smart contract can be given in Solidity source code or in `EVM` compiled code. In the latter case, the annotations have been compiled into bytecode as described above.

2. *CFG.* In either form, the code is given to our CFG generator built on top of Oyente [9]. As Oyente does not handle recursive functions, they are already discarded at this step (Section 3.1).

3. *EthIR.* The decompilation of the `EVM` bytecode into a higher-level *rule-based representation* (RBR) is carried out from the generated CFG by EthIR (Section 3.2).

4. *C+SV-COMP translator.* We have implemented a translator for the recursive RBR representation into an *abstract* Integer C program (i.e., all data is of type Integer) with verification annotations using the `SV-COMP` [11] format. Features of the `EVM` that we cannot handle yet (e.g., bit-wise operations) are abstracted away in the translation. `INVALID` instructions are transformed into `ERROR` annotations in the C program following the `SV-COMP` format.

5. *Verification.* Any verification tool for Integer C programs that uses `SV-COMP` annotations can be used to verify the safety of our C-translated contracts. We have evaluated our approach using three state-of-the-art C verifiers, CPAchecker [29], VeryMax [35], and SeaHorn [62], and the verification report they produce is processed by us to report the results in terms of functions of the smart contract.

```
1  contract SmartBillions {
2    struct Wallet {
3      ...,   uint16 lastDividendPeriod;}
4    uint public dividendPeriod;
5    uint[] public dividends;
6    mapping(address => uint) balances;
7    uint public totalSupply;
8    mapping (address => Wallet) wallets;
9    function commitDividend(address _who) {
10 ⊕  //require(totalSupply > 0);
11 ⊕  //require(dividendPeriod < dividends.length);
12     uint last=wallets[_who].lastDividendPeriod;
13 ⊕  //require(dividendPeriod >= last);
14      ...
15 ⓘ   uint share=balances[_who]*0xffffffff/totalSupply;
16     uint balance = 0;
17     for(; last<dividendPeriod; last++) {
18 ⓘ       balance += share * dividends[last];
19     }
20 ⓘ⊕ assert(last == dividendPeriod);
21     balance = (balance / 0xffffffff);
22       ...
23   }
24 }
```

```
block734(s5,...,s0,g4,g1,g0,l3,l2) ←
  ...
  call(jump734(s7,...,s0,g4,g1,g0,l3,l2))
jump734(s7,...,s0,g4,g1,g0,l3,l2) ←
  // last ≥dividendPeriod
  geq(s7,s6),
  call(block789(s5,...,s0,g4,g0,l3,l2))
jump734(s7,...,s0,g4,g1,g0,l3,l2) ←
  // last <dividendPeriod
  lt(s7,s6),
  call(block745(s5,...,s0,g4,g1,g0,l3,l2))
block745(s5,...,s0,g4,g1,g0,l3,l2) ←
  call(jump745(s9,...,s0,g4,g1,g0,l3,l2))
jump745(s9,...,s0,g4,g1,g0,l3,l2) ←
  // last <dividends.length
  lt(s9,s8),
  call(block759(s7,...,s0,g4,g1,g0,l3,l2))
jump745(s9,...,s0,g4,g1,g0,l3,l2) ←
  // last ≥dividends.length
  geq(s9,s8),
  call(block758(s7,...,s0))
block758(s7,...,s0) ←
  INVALID
block759(s7,...,s0,g4,g1,g0,l3,l2) ←
  ...
  s7 = sha3(s8,s7),      // KECCAK256
  s6 = s7+s6,            // ADD
  s6 = fresh0,           // SLOAD
  s7 = s4,               // DUP3
  s6 = s7*s6,            // MUL
  ...
  call(block734(s5,...,s0,g4,g1,g0,l3,l2))
```

Figure 4.2: Solidity code (left) and excerpt of RBR rules of for loop (lines 21-23)

## 4.2 From RBR to C Program

As motivating example, we use a Solidity contract that implements a lottery system called *SmartBillions* (available at `https://smartbillions.com/`). We illustrate the safety verification of its internal function `commitDividend` (an excerpt of its code appears to the left of Figure 4.2) that commits remaining dividends to the user `wh`. We have shortened the variable names by removing the vowels from the names. Lines marked with ⓘ might lead to executing different sources of `INVALID`: L15 to a division by zero when `ttlSpply` is 0; at L18 when `lst` $\geq$ `dvdnds.length` and thus it is accessing a position out of the bounds of the array; and at L20 when the condition within the assert does not hold. In order to be able to verify its safety (i.e., absence of `INVALID` executions), we add the lines marked with ⊕ that introduce error-handling functions `require` and `assert` in the verification process.

The starting point of our translator is the RBR produced by EthIR (Section 3.2). To the right of Figure 4.2 we show the fragment of the RBR produced by EthIR for the loop of L17-L19.

Observe that the fragment of the RBR contains an `INVALID` instruction within *block*758 and such block can be executed when $geq(s_9, s_8)$ (see rule *jump*745). By tracking variable assignments, we can infer that $s_9$ contains the value of `lst` and $s_8$ the size of `dvdnds`, hence the comparison is checking out-of-bounds array access. The RBR is translated into an abstract Integer C program in four phases:

```
25 // dividendPeriod
26 int g0 = __VERIFIER_nondet_int();
27 // wallets
28 int g4 = __VERIFIER_nondet_int();
29 // last
30 int l0 = __VERIFIER_nondet_int();
31 int l3 = __VERIFIER_nondet_int();
32 // who
33 int who = __VERIFIER_nondet_int();
34 ...
35 void jump734(int s7 ,..., int s0){
36   // last>=dividendPeriod
37   if(s7>=s6) block789(s5 ,..., s0);
38   else       block745(s5 ,..., s0);
39 }
40 void jump745(int s9 ,..., int s0){
41   // last < dividens.length
42   if(s9<s8) block759(s7 ,..., s0);
43   else      block758(s7 ,..., s0);
44 }
46 void block734(int s5 ,...,  int s0){
47    ...
48   jump734(s7 ,...,  s0);
49 }
50 void block745(int s5 ,..., int s0){
51    ...
52   jump745(s9 ,..., s0);
53 }
54 void block759(int s7 ,..., int s0){
55    ...
56   s7 = __VERIFIER_nondet_int();
57   s6 = s7 + s6;
58   s6 = __VERIFIER_nondet_int();
59    ...
60   block734(s5 ,..., s0);
61 }
62 void block758(int s7 ,..., int s0) {
63   ERROR: __VERIFIER_error();
64 }
```

Figure 4.3: C translated code with SV-COMP annotations

1. C functions: Our translation produces, for each non-recursive rule definition in the RBR, a C function without parameters that returns `void`. Recursive rules produced by loops are translated into iterative code. For this part of the translation, we compute the *strongly connected component* (SCC) from the CFG (see Figure 4.1) and model the detected loops by means of `goto` instructions. Figure 4.3 shows the obtained C functions from the RBR program of Figure 4.2. Note that *jump* rules are translated into an *if-then-else* structure.

2. Types of variables: Solidity basic, signed and unsigned data types are stored into untyped 256-bit words in the `EVM` bytecode, and the bytecode does not include information about the actual types of the variables. Moreover, most `EVM` operations do not distinguish among them except for few specific signed operations (`SLT`, `SGT`, `SIGNEXTEND`, `SDIV` and `SMOD`). As verifiers behave differently w.r.t. overflow [29, 35, 62], our translation allows the user to choose (by means of a flag) if all variables are declared with type `int` in the C program, or of type `unsigned int` with casting to `int` for sign-specific operations. The code in Figure 4.3 uses the default `int` transformation. Thus, although in `EVM` integers have overflow, the interpretation of them as unbounded integers or with overflow will be determined by the available options in the C verification tool (e.g., VeryMax only handles unbounded integers). Besides, instructions that contain `fresh` variables or that are not handled (like `SLOAD`) are translated into a call to function `__VERIFIER_nondet_int` in order to model the lack of information for them during verification. Arrays or maps are not visible in the `EVM` (nor in the RBR). The only information that is trackable about arrays corresponds to their sizes as it is stored in a stack variable that in the C program is stored in an integer variable.

3. Variable definitions: In order to enable reasoning on them (within their scopes) dur-

ing verification, SAFEVM translates them in the C program as follows: (i) as we flattened the execution stack, we declare the stack variables as global C variables to make them accessible to all C functions. These variables do not need to be initialized as they take values in the program code; (ii) local variables are defined as global C variables (L30-L31) because a function of the contract might be translated into several C functions, and all of them need to access the local data. They are initialized at the beginning of the function corresponding to the block in which they are firstly used; (iii) state variables are also translated into global variables accessible by all functions and, as their values when functions are verified are unknown, they are initialized using `__VERIFIER_nondet_int` (L26-L28); and (iv) function input parameters are also defined as global variables (for the same reason as (ii)), whose initial values are not determined (L33).

4. SV-COMP annotations: The verification of Ethereum smart contracts is done in SAFEVM by guaranteeing the unreachability of the `INVALID` operations in the C-translated code. Following the `SV-COMP` rules, we translate `INVALID` operations into calls to the `__VERIFIER_error` function so that its unreachability can be proven by any verification tool compatible with the `SV-COMP` annotations. An example of an `INVALID` operation can be seen in L63. Verification tools return that the program in Figure 4.2 cannot be verified as the `INVALID` instruction could be executed. This is due to the fact that contract state values are unknown, that is: `ttlSpply` is not guaranteed to be different from 0 at L15 and the size of the array `dvdnds` is not guaranteed to be greater than the value of `lst` at L18. Lines L10 and L11 contain the Solidity instructions needed to guarantee that L15 and L18, respectively, will never execute an `INVALID` instruction. The `assert` at L20 can be verified by using the `require` at L13. The inclusion of the `require` annotation also improves the contract as, if it is violated, a `REVERT` rather than an `INVALID` bytecode will be executed, not causing a loss of gas of the transaction (the gas needed to check it is negligible).

## 4.3 Related Work

Verification of Ethereum smart contracts for potential safety and security vulnerabilities is becoming a popular research topic with numerous tools being developed. Among them, we have tools based on symbolic execution [70, 55, 81, 65, 63, 91], tools based on SMT solving [72, 64, 75], runtime verification tools [60, 88] and tools based on certified programming [30, 53, 26]. There are also some tools that aim at detecting, analyzing and verifying non-functional properties of smart contracts, e.g., those focused on reasoning about the gas consumption [25, 38, 52, 72] that will be discussed in further detail in the next chapter.

As mentioned before, Oyente does not build a complete CFG and thus, the analysis presented in [70] is unsound. The paper identifies four vulnerabilities: (1) transaction-ordering dependence happens when the final state of the contract depends on the order in which two different transactions are executed, (2) timestamp dependence appears on contracts that use the timestamp as a triggering condition to start an event while the timestamp can be varied by the miners in 900s, (3) exceptions dependence occurs when the contract does not check the return value of a call to an external contract explicitly, and

(4) reentrancy vulnerability appears when the recipient of the call makes use of the current state of the caller. The work proposes programming patterns to avoid the vulnerabilities. The analysis is based on searching traces that meet different conditions. However, as the CFG generated is not complete, it is not able to analyze all possible traces and the results are not sound. It also raises false alarms. After publishing the paper, new vulnerabilities have been included in the tool such as integer overflow.

In [55], an algorithm to study reentrancy vulnerabilities is introduced. It defines a safety property called Effectively Callback Free (ECF), for objects and executions in order to allow modular reasoning on smart contracts. An object is ECF in a given trace if there exists an equivalent execution trace without callbacks that leads to the same state. The idea behind it is to check if the callback may affect to the atomic behavior of the transaction or not. Thus, the objects that satisfy the ECF property can be trusted. They also present an online algorithm for checking if the execution under analysis is ECF. It is based on detecting conflicting memory accesses and uses a relation similar to a happens-before, in an effective way, i.e., it is not computed for all possible permutations of the traces.

MAIAN [81] is a tool for precisely specifying and reasoning about trace properties on smart contracts. It employs inter-procedural symbolic analysis and concrete validation. The approach focuses on three trace properties that lead to vulnerabilities: (1) those that lock the funds of a contract indefinitely, (2) those that transfer the funds to arbitrary users, and (3) those that can be killed by any user. The tool is able to find both safety (if there exists a trace from a specified state that causes a violation of the condition under analysis) and liveness (if some action cannot be taken in any execution starting from a specified state) properties over traces. It is based on a modified version of the semantics defined in [70] and in the notion of contract trace according to the semantics. [81] also characterizes the safety and liveness properties formally in terms of contract traces. In a first step MAIAN executes a symbolic analysis to recover the possible traces of the public functions of the contract. When it finds a state where the desired property does not hold, it emits a warning classifying the contract as potentially vulnerable. It returns constraint paths whose feasibility is proven using Z3 [45]. Finally, in a second step they check the vulnerabilities with a concrete validator that forks a private copy of the original Ethereum blockchain and runs the concrete trace found.

In [65], teEther is presented, a tool that allows creating and exploiting smart contracts given their `EVM` bytecode. As the previous tools, it is based on searching execution traces. The tool identifies paths that lead to a critical instruction. Critical instructions are those that cause a direct transfer of gas (`CALL` and `SELFDESTRUCT`) or that allow arbitrary code to be executed in the context of the main contract that starts the transaction (`CALLCODE` and `DELEGATECALL`). The tool recovers the CFG of the contract and then searches for critical paths. Once a critical one is found, it creates a set of path constraints through symbolic execution. Finally, the constraints of critical paths are solved using Z3 [45] to check if the path is feasible and reproduce the vulnerability. teEther is able to deal with inter-procedural communications between different contracts.

In [91], the authors present Securify, a security analyzer for smart contracts. It is able to prove contract behavior as safe or unsafe with respect to a given property. First the analysis extracts semantic information from the code. Then, it proves if a property holds through compliance or violation patterns. The analysis of Securify is applied in

three steps: (1) SECURIFY decompiles the EVM bytecode of the contract into a SSA form similarly to [10] and constructs a CFG, (2) it infers the semantic information needed for the analysis using inference rules specified in Datalog [56], as [52] does, and (3) it checks if the predefined security patterns are violated using the Datalog solver. If the analysis discovers a violation pattern that matches the inferred semantic facts, SECURIFY returns the vulnerable instructions. Otherwise the tool reports a warning to indicate that the contract may or may not be vulnerable.

[64] investigates vulnerabilities defined as *event-ordering* (EO) bugs. These vulnerabilities are related to the dynamic order in which the contract events are executed. They present ETHRACER, an automatic analyzer that works on the EVM bytecode of the contract to find EO bugs. Intuitively, the analysis constructs inputs and tries all the possible function orderings until an EO bug or a timeout is reached. It finds out if reordering the function calls of a contract produces different final states. Note that this can lead to an intractable analysis because of the state explosion. Thus, the analysis implements partial-order-reduction techniques to reduce the search space based on the classical notion of happens-before relation. ETHRACER uses a symbolic analysis to recover all the public functions of the contract under analysis and dynamic symbolic execution techniques to reason about the output of each public function separately. After that, similarly to other state-of-the-art tools, the constraints generated by each execution path are solved by a SMT solver to infer the happens-before relations and transform the symbolic values into concrete ones in each path.

[47] shows how standard techniques from runtime verification can be applied to smart contracts. It proposes a new stake-based instrumentation technique to infer the violating parties of a contract. [47] presents CONTRACTLARVA, a proof-of-concept tool that implements these techniques. The verification framework proposed enables the combination of the contract and its specification. The contract behaves as the original one but can identify the specification violated. To define properties they use an automaton-based approach, a subset of Dynamic Automata with Timers and Events (DATE) [41]. DATE monitors for events on the contract and allows specifying the behavior of vulnerable event traces. CONTRACTLARVA monitors control- and data-flow events expressed as a symbolic automata that listens to contract events. The tool parses the source code of the contract and, thanks to DATE specification is able to generate a safe contract including the monitoring logic in the code as Solidity modifiers.

ZEUS [63] translates Solidity code of a smart contract into LLVM and uses SeaHorn [62] to classify six types of vulnerabilities: (1) reentrancy, (2) unchecked send, (3) failed send, (4) integer overflow/underflow, (5) block/transaction state dependence, and (6) transaction order dependence. To check vulnerabilities (2) and (3) ZEUS introduces a condition to validate if a `send` instruction has been executed correctly into an `assert` statement. To test (5), which is detected when block variables such as `timestamp` affect to `send` or `call` instructions, ZEUS implements a taint analysis and uses a symbolic model checker to eliminate unfeasible paths. However the transformation from Solidity to LLVM is not complete.

[88] presents Solitor, a tool that applies runtime verification techniques to make smart contracts more secure. As proposed in [47], it allows the user to specify the behavior of the contracts using annotations. The tool parses and translates the annotations into Solidity and checks them at runtime. Annotations are logical expressions that can reference

contract variables or blockchain identifiers to define contract invariants or pre and post conditions for functions. First the contract code has to be annotated according to a grammar generated by the tool ANTLR. ANTLR [83] generates a parser that is used to transform Solidity code and annotations into a parse tree. The parse tree allows analyzing different parts of the contract. The next step is type checking the annotations to prove that they are valid and generating parse tree objects that correspond to type-checked annotations. Finally the information generated by the type checker is used together with the Solidity source code to obtain the runtime monitored contract.

[60] presents ContractFuzzer, a generator of fuzzing inputs based on the ABI specification of smart contracts. It defines test oracles to detect security vulnerabilities, instruments `EVM` code to generate logs with information about the behavior of the contract and analyzes these log files to report security vulnerabilities. ABI (application binary interface) defines the interface between two program modules. Data is encoded according to its type. The tool takes an instrumented `EVM` code as input and the ABI specification of the deployed contract. It obtains the signature of the functions and the types of its arguments. Knowing the signature of the functions, the tool is able of analyzing the interaction between different contracts. Thanks to the analysis of the instrumented `EVM` and the ABI specification, ContractFuzzer generates valid fuzzing inputs and starts to execute functions randomly. Finally it analyzes the log reports created by the executions in order to find seven types (all of them explained before) of vulnerabilities: (1) unchecked send, (2) unhandled exception, (3) reentrancy, (4) timestamp dependence, (5) block number dependence, (6) dangerous `DELEGATECALL`, and (7) freezing contracts.

The tools proposed in [63] and [75] are the most similar to SAFEVM. Zeus [63] and SAFEVM have the same target of verifiers because Zeus generates LLVM code and we produce C programs. However we cannot compare both approaches because the code of Zeus is not public. Mythril [75, 8] obtains the constraint paths generated by its intermediate representation and solves them using Z3 [45] as SMT solver. The difference between Mythril and SAFEVM relies on the way in which the assert statement is treated. Both tools study the reachability of `INVALID` bytecode instructions but Mythril obtains the corresponding set of constraints from its intermediate representation and SAFEVM leaves this task to the verifiers.

To conclude, to the best of our knowledge, SAFEVM is the first tool that uses existing verification engines developed for C programs to verify low-level `EVM` code. This opens the door to the applicability of advanced techniques developed for the verification of C programs to the new languages used to code smart contracts.

# Chapter 5

# Gastap: A Gas Analyzer for Smart Contracts

This chapter presents GASTAP [25], a *Gas-Aware Smart contracT Analysis Platform*. GASTAP infers gas upper bounds for all public functions of a contract, i.e., it infers upper bounds of the gas that each function of the contract needs to be executed without aborting the execution due to run-out-gas exceptions. The upper gas bounds inferred by GASTAP can be parametric in terms of the size of the arguments of the functions of the contract, the state of the contract or blockchain data. The inference of gas requires complex transformation and analysis processes on the code (see Section 2 in [25] for technical details): (1) construction of CFGs, (2) decompilation from low-level code to a higher-level representation, (3) inference of size relations, (4) generation of gas equations, and (5) solving the equations into gas bounds.

We have used GASTAP to analyze more than 29,000 real smart contracts pulled from Ethereum blockchain. In total GASTAP has inspected 258,541 public functions (and all auxiliary functions that are used from them). For the analyzed contracts, a large number of functions, 86.37% and 87.36% have a constant opcode and memory gas consumption respectively. This is as expected because of the nature of smart contracts. There is a relevant number of functions (5.48% and 5.15%) for which we obtain an opcode and memory gas bound that is not constant and hence are potentially vulnerable (see Section 3.2 in [25] for details).

GASTAP is available online and can be tried at `https://costa.fdi.ucm.es/gastap`.

## 5.1 Architecture of Gastap

Figure 5.1 shows the main components of GASTAP that are as follows:

1. *Input.* GASTAP takes as input a smart contract that can be written in Solidity, `EVM` code or disassembled code.

2. *CFG.* Independently of the input, it recovers the CFG of the smart contract under analysis.

3. *RBR.* It uses ETHIR (Section 3.2) to decompile the low-level code represented in the CFG to a higher-level representation based on guarded rules.

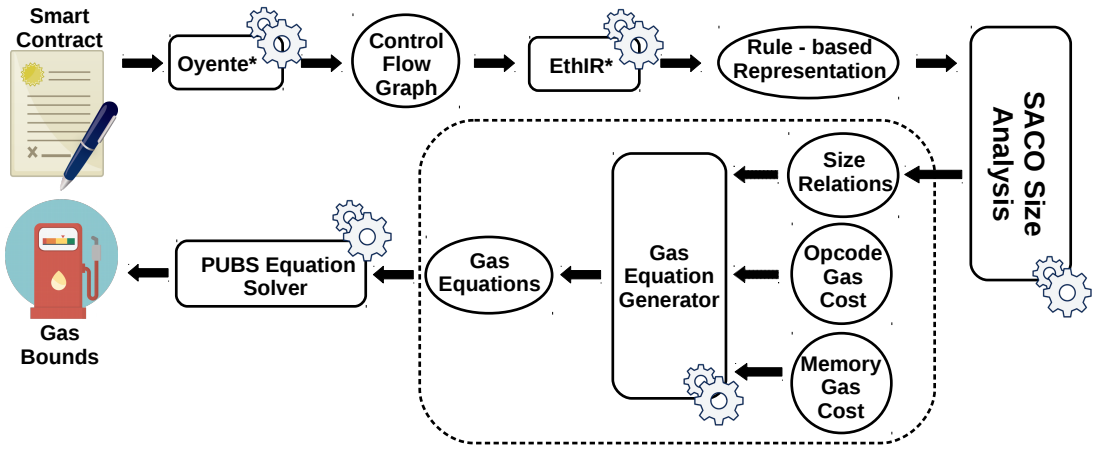



Figure 5.1: Architecture of GASTAP.

4. *Size relations.* It uses the RBR to infer size relations (SR) that show the relation between the input and output data and how the data are modified in each rule.

5. *Gas equations.* It generates a gas equation for each rule of the RBR taking into account the SR inferred in the previous step. They depend on the opcodes executed and the memory used during the transaction.

6. *Gas upper-bounds.* The gas equations are solved to infer closed-form gas upper bounds.

Therefore, building an automatic gas analyzer from EVM code requires a daunting implementation effort that has been possible thanks to the availability of a number of existing open-source tools that we have succeeded to extend and put together in the GASTAP system. In particular, the modified version of the tool OYENTE is used for point (1) (see Section 3.1), the intermediate representation of ETHIR [24] is used for point (2) (see Section 3.2 for details), an adaptation of the size analyzer of SACO [15] is used to infer the size relations, and the PUBS [16] solver for point (5).

GASTAP takes as a starting point the RBR introduced in Chapter 3. In what follows we discuss the components related to points 4, 5 and 6.

## 5.2 Size Relations

Assuming that we have the intermediate representation generated by ETHIR, the next step is to generate *size relations* (SR) from the RBR using the SACO tool [15]. SR are equalities and inequalities that state how the sizes of data change in the rule [43]. This information is obtained by analyzing how each instruction of the rules modifies the sizes of the data it uses, and propagating this information as usual in dataflow analysis. SR are needed to build the gas equations and then generate gas bounds in the last step of the process. The size analysis of SACO has been slightly modified to ignore the *nop* instructions. Besides,

$$
\begin{array}{l}
\mathit{block1611}(\overline{s_6}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad 15 + \mathit{block1619}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_8 = 0, s_7 = 0\} \\
\mathit{block1619}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad 234 + \mathit{jump1619}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_{10} = s_8, s_9 = g_3\} \\
\mathit{jump1619}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad \mathit{block1744}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_{10} \geq s_9\} \\
\mathit{jump1619}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad \mathit{block1633}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_{10} < s_9\}
\end{array}
\qquad
\begin{array}{l}
\mathit{jump1633}(\overline{s_{12}}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad \mathit{block1647}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_{11} > s_{12}\} \\
\mathit{block1647}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad 244 + \mathit{jump1647}(s'_{10}, s'_9, \overline{s_8}, \overline{sv}, l'_2, l_0, \overline{bc}) \\
\quad \{s'_{10} = s_6, l'_2 = s_9\} \\
\mathit{jump1647}(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad \mathit{block1731}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s_{10} > s_9\} \\
\mathit{block1731}(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) = \\
\quad 41 + \mathit{block1619}(s'_8, \overline{s_7}, \overline{sv}, \overline{lv}, \overline{bc}) \\
\quad \{s'_8 = 1 + s_8\}
\end{array}
$$

Figure 5.2: Gas equations for opcode gas model obtained by GASTAP.

before sending the rules to SACO, we replace the instructions that cannot be handled (e.g., bit-wise operations, hashes) by assignments with fresh variables (to represent an unknown value). Apart from this, we are able to adjust our representation to make use of the approach followed by SACO, which is based on abstracting data (structures) to their sizes. For integer variables, the size abstraction corresponds to their values and thus it works directly. However, a language specific aspect of this step is the handling of data structures like arrays, strings or bytes (an array whose elements are of type byte). In the case of array state variables, SACO's size analysis works directly as in `EVM` the slot assigned to the variable contains indeed its length (and the address where the array content starts is obtained with the hash of the slot address).

**Example 5.1.** *In Figure 5.2, we can see, inside the braces, the generated SR for every rule of the RBR depicted in Figure 3.3. Note that it corresponds to the decompilation of the code showed in Figure 3.1. For instance, the size relations for the* jump1619 *function involve the* `slots` *array length ($g_3$ stored in $s_9$) and the local variable* `i` *(in $s_8$ and copied to $s_{10}$). It corresponds to the guard of the* `for`*-loop in function* `findWinner` *that compares* `i` *and* `slots.length` *and either exits the loop or iterates (and hence consume different amount of gas). The size relation on $s_8$ for* block1731 *corresponds to the increase in the loop counter.*

## 5.3 Generation of Gas Equations

This section describes the process of generating gas equations, given the RBR in Section 3.2 and the SR produced in Section 5.2. The generation is split into two steps, the generation of the equations for opcode gas cost and for memory gas cost. Both types of equations can then be solved using an off-the-shelf cost relation solver as described in Section 5.4.

In order to generate gas equations (GE), we need to define an `EVM` gas model, which over-approximates the specification of the gas consumption for each `EVM` instruction as

provided in [95]. The EVM gas model is complex and unconventional, it has two components, one which is related to the memory consumption, and another one that depends on the bytecode executed. The first component is computed separately as will be explained below. To compute the gas attributed to the opcodes, we define a function $C_{opcode} : s \mapsto g$ which, for an EVM opcode, takes a stack $s$ and returns a gas $g$ associated to it. We distinguish three types of instructions:

1. Most bytecode instructions have a *fixed* constant gas consumption that we encode precisely in the cost model $C_{opcode}$, *i.e.*, $g$ is a constant.

2. Bytecode instructions that have different *constant* gas consumption $g_1$ or $g_2$ depending on some given condition. This is the case of SSTORE that costs $g_1 = 20000$ if the storage value is set from zero to non-zero (first assignment), and $g_2 = 5000$ otherwise. But it is also the case for CALL and SELFDESTRUCT. In these cases, for soundness, we use $g = max(g_1, g_2)$ in $C_{opcode}$.

3. Bytecode instructions with a non-constant (*parametric*) gas consumption that depends on the value of some stack location. For instance, the gas consumption of EXP is defined as $10 + 10 \cdot (1 + \lfloor log_{256}(\mu_s[1]) \rfloor)$ if $\mu_s[1] \neq 0$ where $\mu_s[0]$ is the top of the stack. Therefore, we have to define $g$ in $C_{opcode}$ as a parametric function that uses the involved location. Other bytecode instructions with parametric cost are CALLDATACOPY, CODECOPY, RETURNDATACOPY, CALL, SHA3, LOG*, and EXTCODECOPY.

Given the RBR annotated with the nop information, the size relations, and the cost model $C_{opcode}$, we can generate GE that define the gas consumption of the corresponding code applying the classical approach to cost analysis [94] which consists of the following basic steps:

- Each rule is transformed into a corresponding cost equation that defines its cost. Figure 5.2 displays the GE obtained for the rules of Figure 3.3.

- The nop instructions determine the gas that the rule consumes according to the gas cost model $C_{opcode}$ explained above. Those are the accumulated costs in the equations, e.g., *block1611* accumulates *15* units that correspond to the gas consumed by the instructions in this block. In the example, the accumulated cost is always constant, however, there are equations whose accumulated cost is parametric with respect to some of the entries of the function analyzed, e.g., the equation for *block561*, that is generated from the RBR of method __callback (omitting rule arguments and constraints) is: $447 + 3 \cdot result/32 + 3 \cdot proof/32 + block2187$. The parametric accumulated cost comes from the two CALLDATACOPY opcodes that appear in this block.

- Calls to other rules are replaced by calls to the corresponding cost equations. See for instance the call to *block1619* from rule *block1611* that is transformed into a call to the cost function *block1619*.

- Size relations are attached to rules to define their applicability conditions and how the sizes of data change when the equation is applied. See for instance the size relations attached to *jump1619* that have been explained in Example 5.1.

As said before, the gas cost model includes a cost that comes from the memory consumption. Intuitively, the memory gas cost depends on the highest memory address accessed during the execution of the function under analysis. It is obtained by computing the highest slot of memory accessed for each bytecode executed and paying a fee every time that the memory is extended. Besides `MLOAD` or `MSTORE`, instructions like `SHA3` or `CALL`, among others, make use of the local memory, and hence can increase the memory gas cost.

This is not a standard memory consumption analysis in which one obtains the total amount of memory allocated by the function. Instead, in this case, we need to infer the actual value of the highest slot accessed by any operation executed in the function.

In order to estimate the cost associated to all `EVM` instructions in the code of the function, we first make the following observation: Computing the sum of all the memory gas cost amounts to computing the memory cost function for the highest memory slot accessed by the instructions of the function under analysis.

One possible way to infer this information is by means of a size analysis [34] that computes upper bounds on sizes of all expressions used to access the memory (note that they can be parametric). After this, in a second step, a maximization of all these upper bound expressions is required.

Our approach to solve this problem is to view it as an instance of the peak resource analysis problem [18, 59]. This analysis, rather than accumulating all costs as in standard resource analysis, computes the peak (*i.e.*, the maximal) of the resource consumption of the whole execution. In order to use the peak analysis to our end, for each instruction that accesses a memory location $l$ we count as it allocates $l$ resources.

**Example 5.2.** *Let us show how we obtain the memory gas cost for* block1647. *In this case, the two instructions in this block that cost memory correspond to a* `MSTORE` *and* `SHA3` *bytecodes. In this block, both bytecodes operate on slot 0 of the memory, and they cost 3 units of gas because they only activate up to slot 1 of the memory.*

## 5.4 From Equations to Close-Form Bounds

The last step of the gas bounds inference is the generation of a *closed-form gas upper bound*, i.e., a solution for the GE as a non-recursive expression. As the GE we have generated have the standard form of cost relations systems, they can be solved using off-the-shelf solvers, such as Pubs [16] or Cofloco [50] (Pubs can find logarithmic bounds too), without requiring any modification. These systems are able to find polynomial, logarithmic, and exponential solutions for the input cost relations in a fully automated way.

The gas bounds computed for `__callback` and `findWinner` from the equations in Figure 3.3 using Pubs are $229380+3\cdot(proof/32)+103\cdot(result/32)+50\cdot(32-result)+5836\cdot g3+5057\cdot g1$ and $1555+779\cdot g3$ respectively. Note that they are parametric on different state variables, input and blockchain data. For instance, the upper bound inferred for the function `findWinner` is linear on the size of the third state variables $g3$, that represents the size of the array `slots` and is the bound of the for-loop that `findWinner` contains.

## 5.5 Related Work

As mentioned in the previous chapter, the analysis of Ethereum smart contracts to identify vulnerabilities before the contract is deployed is a topic that is gaining popularity in the last years. The state-of-the art tools are based on symbolic execution [70, 55, 81, 65, 63, 91], SMT solving [72, 64], and certified programming [30, 53, 26]. However, most of them ignore resource usage, focusing on non-gas-related safety, security, and temporal properties. As GASTAP is not meant to be an *all-in-one* smart contract analyzer and focuses exclusively on gas consumption, in this section we only review to the tools and approaches that are concerned with gas usage and the corresponding *out-of-gas* vulnerabilities.

The GASPER tool identifies gas-costly programming patterns [38], which can be optimized to consume less. For doing so, it relies on matching specific control-flow patterns, SMT solvers and symbolic computation, which makes their analysis neither sound, nor complete for determining gas usage bounds. The authors classify the patterns in two groups: useless-code related patterns and loop-related patterns. Although they define seven different patterns, GASPER is able to identify only three of them, that covers the two categories mentioned above. The aim of the first pattern depicted in [38] is to detect dead code (code whose execution depends on a condition that has to be tested at execution time) on the smart contract, the second detects opaque predicates (code that depends on the value of a condition that is set at compilation time) and the third one finds expensive operations in a loop (recall that the operations over storage are more expensive than those that operate in memory). GASPER takes the bytecode of a contract and generates a CFG through symbolic execution using Z3 solver to check if the guards of conditional branches are feasible. When the CFG is constructed, it analyzes the code to recognize the patterns. To detect dead code, GASPER compares the blocks of the CFG (which are known statically) with those executed during the symbolic execution. Opaque code is detected by testing if there is any branch that is never executed. Finally, the expensive operations in loops are analyzed searching the loops inside the CFG and identifying if any of the blocks involved in the loop contains the bytecode instructions `SSTORE`, `SLOAD` or `BALANCE`. These techniques focus on detecting code patterns to improve the development of the contract rather than inferring gas consumption bounds. In fact the gas information does not take part in the analysis presented.

The previous work is extended in [39], where they present GASREDUCER. GASREDUCER also works at `EVM` bytecode level. GASREDUCER takes the bytecode of a smart contract and outputs an optimized version of the original one that consumes less gas. The authors increase the number of defined patterns from seven to twenty-four. In this case the patterns are defined as a sequence of `EVM` instructions that can be replaced by another one that consumes less gas but has the same semantics as the original sequence. To identify the patterns, the authors inspect several instances from the execution traces of deployed smart contracts. GASREDUCER transforms the `EVM` bytecode into the disassembled one. Then, it analyzes the disassembled code iteratively until no pattern is found. When the tool identifies a pattern, it generates a report containing the location of the pattern and the corresponding optimized code. After replacing the code of the pattern with the efficient one, GASREDUCER reconstructs the disassembled code of the new contract executing it symbolically as the jump address may have changed.

In a similar vein, the work [52] identifies several classes of gas-focused vulnerabilities,

and provides MadMax, a static analysis, also working on decompiled EVM bytecode, that combines techniques from flow analysis together with control-flow analysis (CFA), context-sensitive analysis and modeling of memory layout. MadMax uses Vandal [33] as its decompiler in order to get an intermediate representation of the contract. The analysis is implemented in Datalog [56]. They define a set of Datalog rules in order to model memory and storage layouts. In a first step, MadMax infers loop and data-flow information. From loops, it infers information related to the exit condition of the loop or induction variables, i.e., those that are incremented by a concrete value inside the loop. The data-flow analysis provides information such as aliasing or dependencies between variables. It also implements a constant propagation. However, the data-flow analysis is neither sound nor complete. Using the basic loop and data-flow analysis, MadMax is able to infer high-level concepts such as array iterators or if the storage is increased on public functions. MadMax differs from Gastap as it focuses on identifying control- and data-flow patterns inherent for the gas-related vulnerabilities, thus, working as a bug-finder, rather than complexity analyzer.

Since deriving accurate worst-case complexity boundaries is not a goal of any of GASPER, GasReducer and MadMax, they are unsuitable for inferring gas upper bounds.

Other tools based on proof assistants [58, 73, 57, 53] may reason about out-of-gas exceptions. They can model how gas is updated along the execution of the trace and encode it as constraint formulas. However these tools are not able to infer loop invariants. They have to be specified manually. Thus, they are not able no infer gas bounds for programs that involve loops.

In a concurrent work [72], the authors identify three cases in which computing gas consumption can help in making Ethereum more efficient: (1) preventing contracts getting stuck with *out-of-gas* exception, (2) placing the right price on the gas unit, and (3) recognizing semantically-equivalent smart contracts and optimize the code. They propose a methodology, based on the notion of the so-called *gas consumption paths* (GCPs), to estimate the worst-case gas consumption using techniques from symbolic bounded model checking [31]. Their approach is based on symbolically enumerating all execution paths and unwinding loops to a limit. The authors suggest two algorithms for studying GCPs. Instead of working in EVM bytecode, the analysis is based on Solidity. However, the analysis computes EVM gas bound using concrete execution paths that cover all Solidity GCPs. Intuitively, the analysis extends the CFG to GCP, and returns the GCP that maximizes the gas consumption. In the first algorithm presented, Gas Consumption Path Enumeration, the authors translate the Solidity code into an unwound SSA (USSA) form. This form consists of a sequence of guarded assignments. Then all the constraints that affect the gas consumption of the USSA form are sent to a SMT solver. From the results of the satisfiable queries the algorithm simulates the transaction precisely and gets the highest gas estimation. In the second algorithm, Function-Oriented Gas Consumption Path Enumeration, the algorithm constructs GCP for each function as cost-equivalence classes. In that case the paths are computed gradually starting from the low-level instructions and increasing the set of GCPs built recursively. It relies on the notion of cost equivalence classes: set of conditions under which the behavior of a function or a block of instructions changes with respect to the gas consumption. Cost equivalence classes correspond to the GCPs of the function that is being analyzed. Once the classes are computed, the

algorithm infers the exact cost of each class and returns the maximum. Instead of using resource analysis, GASTAP infers the maximal number of iterations for loops and generates accurate gas bounds which are sound for any possible execution of the function and not only for the unwound paths. Therefore, our results are sound for any possible execution of the smart contract (where loops can be executed any number of times). As we have seen, using a resource analysis approach, in addition to inferring precise cost expressions for constant gas consumption as in [72], we can go beyond that, and generate parametric gas bounds. Besides, to the best of our knowledge, the approach proposed in [72] has not been implemented in the context of `EVM` and has not been evaluated on real-world smart contracts as ours.

GASTAP is to the best of our knowledge, the first automatic gas analyzer for smart contracts.

# Chapter 6

# Conclusions and Future Work

In this thesis we have presented two MHP analyses (Section 2.1 and Section 2.2) for asynchronous programs based on the actor model. In this model, the objects (actors) are the concurrency units. An MHP analysis infers the pairs of program points that may interleave their execution. The termination of asynchronous calls can be awaited using specific instructions and future variables [48, 44]. The MHP analysis learns from the future variables used in the synchronization instructions when tasks are terminated, so that the analysis can accurately eliminate unfeasible MHP pairs that would be inferred otherwise.

There are several MHP analysis developed for multiples languages and concurrency models [79, 12, 67, 85, 46, 28, 37, 40, 36]. In [12, 67, 85], the problem of developing a MHP analysis is studied for the concurrency model with "async-finish parallelism". In [79], an MHP analysis for Ada is developed. Ada has a rendezvous concurrency model based on synchronous message passing. [28] and [37] infer a thread-level MHP analysis for programs in Java and its concurrency model based on threads. The analysis presented in [40, 36] builds a time based model to infer race conditions in high performance systems. The analysis relies on an MHP analysis which is applied in a segment graph. On one hand, some of these analyses [79, 12, 67, 85] do not support that the synchronization of asynchronous calls is carried out in a different scope from that in which they were created. On the other hand, existent analyses [28, 37, 40, 36] allow synchronization of asynchronous tasks in different scopes but they are not precise enough.

The analyses presented [22, 23] extend the MHP analysis developed in [21]. The information is inferred in two phases: (i) the local phase infers the information local to each method, and (ii) the global phase where all the information of the previous step is composed in order to infer transitive relations. This original analysis only supports intra-procedural synchronization, i.e., the tasks can only be awaited in the same scope in which they were spawned. Thus, we have enhanced the original MHP analysis [21] in order to analyze inter-procedural synchronization:

- In Section 2.1, we present a MHP analysis [22] where the tasks can be synchronized in inner scopes so the future variables can be passed to the methods as parameters. It proposes a MHF analysis, a novel pre-analysis that infers, for each program point, which tasks must have finished their execution. This information is incorporated to both phases of the analysis.

- In Section 2.2, we present a MHP analysis [23] that allows awaiting the tasks in

outer scopes from those that spawn them as the tasks can return future variables. It modifies both phases in order to back-propagate the new information generated by the returned future variables.

These extensions improve significantly the precision of the initial MHP analysis [21]. The MHP information has revealed to be fundamental to build more complex analysis of safety and liveness properties such as deadlock, termination or resource analysis. Improving the precision of the MHP analysis, we increase also the accuracy of these analysis.

As future work we highlight the following directions:

- Improving the precision of the MHP analysis when future variables are declared as global variables, i.e., fields of classes. It introduces new challenges to the analysis as it would require tracking all the fields along all the analysis because they can be updated at any program point. Our current approach is to apply the technique proposed in [14] for handling fields that consists in transforming fields to local variables and pass them in arguments whenever it is sound to do so (i.e., when the tasks that interleave with the current one will not modify them). If fields then become arguments that are returned, we can apply our approach to handle returned future variables.

- Although the MHP analysis has been studied for many languages and concurrency models, there are a few publications related to decidability of the analysis. [90] proves that for a rendezvous based concurrency model the problem is undecidable. In contrast, the problem has been proven to be, under some conditions, decidable in lineal time for concurrency models based on X10 [12, 67]. An interesting direction is to study the computational complexity of deciding MHP, for our language, in a similar way to the one presented in [32] for the problem of state reachability.

- The two MHP analyses presented in this thesis are fully compatible. The next step would be the integration of both approaches in a unique framework having the MHF analysis as a precomputation and redefining the local and the global phases.

- Studying how the previous analyses can be combined, their possible applications and their use as part of more complex analyses. The enhancement of precision of the analyses affects directly to other analyses, increasing their complexity. Sometimes the analyses have to be adapted in order to support the improvement of the precision.

- An important aspect related to the applicability of the analyses is the possibility of adapting and generalizing them to different languages and concurrency models. Other concurrency models are less restrictive than the actor model which makes its analysis more difficult. The MHP analyses can be adapted to other concurrent programming patterns.

This thesis also presents a method to decompile Ethereum smart contracts. EthIR [24, 25] is a framework for high-level analysis of smart contracts that takes the bytecode of the smart contract as input and builds an intermediate representation based on guarded rules. The aim of the intermediate representation is to make explicit the control- and data-flow of the contract. The analysis is applied in two main steps:

1. EthIR takes a smart contract written in Solidity, EVM bytecode or disassembled code and generates a CFG. The CFG generator has been built on top of the tool Oyente. It uses the parser of Oyente to simulate the effects of each EVM instruction on the stack. The logic to infer all the possible jump addresses and recover the connections between the blocks of the CFG has been reimplemented in order to obtain a complete CFG.

2. Each block is translated into a set of guarded rules. These rules have explicit variables to represent the stack, memory, storage and blockchain data involved in the contract. Optionally, the rules keep the original bytecodes in the code using nop instructions.

Current state-of-the art tools follow two different approaches: analyzing the source code [47, 63, 88, 72] or analyzing the EVM bytecode [55, 81, 65, 91] of the contract. Analyzing Solidity source code has advantages over the analysis of EVM bytecode as it contains the high-level representation of the contract. The control-flow of the contract is available in the code as well as the data structures. However, the semantics of Solidity is not formally defined and the number of smart contracts whose source code is stored in the Ethereum blockchain is very low (less than 1%). In addition, some properties can be only analyzed at EVM level because they depend on the bytecode instructions of the smart contract.

In the past years, multiple tools for decompiling contracts and building intermediate representations have been published. Some of them have tried to formally reason about Ethereum contracts modeling its semantics [58, 73, 57, 53]. Others symbolically execute the EVM bytecode to generate an intermediate representation [70, 10, 75].

Regarding decompilation of EVM bytecode, we plan to continue the development of the tool in the following directions:

- As mentioned before, the data structures are only explicit at the source code level. In Solidity, the user can define arrays, maps, and structs as data structures. However, when the contract is compiled into EVM code these structures are converted into separated variables and are accessed through hash codes. We aim at making the data structures explicit in our intermediate representation and being able to distinguish them.

- Although there exist types in Solidity, EVM is an untyped language. However, some EVM instructions such as SMOD, SDIV, SLT, SGT or SIGNEXTEND allow detecting the type of the data on which they operate. These EVM instructions differ from the original ones (MOD,DIV,etc.) in that the numbers on which they operate are known to be signed integers represented in two's complement. We plan to use this information to infer the types of the variables. Tools like SAFEVM will benefit from this information.

EthIR constitutes the basis for the two analyzers for smart contracts presented in this thesis: SAFEVM [17] and Gastap [25]. They consider the RBR generated by EthIR as their starting point.

SAFEVM uses C verification tools to prove safety on Ethereum smart contracts. The idea behind the verifier is to study the reachability of the bytecode instruction INVALID. The execution of this opcode has economical consequences: the transaction that is currently running is aborted, the state is reverted to the initial one but the gas spent until

this point is not refunded. Hence, SAFEVM is able to ensure if there exists a possible execution on the smart contract that leads to an `INVALID`. This is done in two steps:

1. By transforming the recursive RBR into an iterative goto-based C program with `ERROR` annotations following the `SV-COMP` format.

2. The C program generated is analyzed by a C verifier. The verifier proves if the `ERROR` annotations are reachable by a possible executions and generates a report. Although we have tested SAFEVM using CPAchecker, VeryMax and SeaHorn as verifiers, the C program that represents the contract can be analyzed by any other verifier that supports `SV-COMP` annotations.

The analysis of properties related to the security and safety of smart contracts has become an interesting research topic. Ethereum smart contracts cannot be modified once they are deployed. A vulnerability in the code may have a high economical impact. There are several tools whose aim is to verify properties of smart contracts. Each tool tries to find different types of vulnerabilities being the most common bugs those related to reentrancy, integer overflows, frozen funds or unhandled exceptions. Some of the tools [70, 55, 81, 65, 63, 91] developed in the recent years find the vulnerabilities using symbolic execution of the contract. Other tools [72, 64, 75] employ SMT solvers to obtain the results of certain constraint systems built during the generation of the intermediate representation of the `EVM` code. There are also analyzers [60, 88] that apply runtime verification techniques to find potential vulnerabilities. Finally, some of the state-of-the art tools are based on certified programming [30, 53, 26].

Despite the high number of existing analyzers, SAFEVM is to the best of our knowledge the first verifier of Ethereum smart contracts that uses C engines. Although SAFEVM is still in a prototypical stage, it provides a proof-of-concept of the transformational approach, and we argue that it constitutes a promising basis to build verification tools for `EVM` smart contracts.

Some of the aspects that we aim at improving in future work are:

- We have identified smart contracts that we are not able to verify due to the memory abstraction performed by EthIR. The development of a memory analysis for `EVM` smart contracts can be crucial for the accuracy of verification.

- Handling bit-wise operations in the future that are extensively used in the `EVM` bytecode. In the current version they have to be abstracted as most of the verifiers do not support them.

- A more advanced reasoning for arrays and maps (the only data structures available in Ethereum smart contracts) can be added to the framework to gain further accuracy.

- Improving the tool to be able to reason about properties which are global to the contract. To verify some properties the analyzer needs to ensure, for instance, that the state variables involved in the property are not modified by any other function of the contract or that two data structures always have the same size.

The last tool presented in this thesis is Gastap [25]. Gastap is a resource analyzer to infer gas upper bounds on smart contracts. In order to execute a transaction on Ethereum

blockchain, an amount of gas must be provided. If the gas provided is less than the gas that the transaction needs to be executed, the transaction will be aborted. GASTAP allows calculating a gas upper bound for each public function of the contract. Assuming that the analysis has the RBR generated by ETHIR, it is implemented in three phases:

1. It computes the size relations that represent how the variables are modified along the rule.

2. It generates gas equations for each rule of the RBR. The gas can be separated in two parts: opcode gas that depends on the `EVM` instruction and memory gas that depends on the memory consumed in the execution of the transaction.

3. The PUBS solver [16] takes the gas equations and the size relations to generate closed-form gas upper bounds.

Automated sound static reasoning about resource consumption is critical for developing safe and secure blockchain-based replicated computations, managing billions of dollars worth of virtual currency. We have adapted and extended state-of-the art techniques in resource analysis, showing that such reasoning is feasible for Ethereum, where it can be used for preventing vulnerabilities and verification or certification of existing smart contracts.

None of the state-of-the art tools presented is able to compute gas bounds for smart contracts. In fact, only a small fraction of tools [38, 39, 52, 72] are focused on analyzing gas consumption. GASTAP is to the best of our knowledge, the first automatic gas analyzer for smart contracts.

As future work, we plan:

- The only difference between our cost model and the specification of the gas consumption given in [95] is that we are not including the cost associated to the gas-cap of the opcodes `CALLCODE`, `CALL` and `DELEGATECALL`, denoted in [95] by $C_{\text{GASCAP}}$. This cost is not real as it is just a cap on the gas consumption of the call and it can be, for instance, all the available gas. Adding the gas-cap would give an unacceptably high estimate. Therefore, our cost analysis only includes the instructions of the code we are actually analyzing and does not include the cost of the code of the external functions if we do not know such code. As future work, we plan to improve the analysis by searching for the code of external calls and, if available, infer its gas consumption and add it.

- GASTAP over-approximates the gas consumption of the public functions of the contract. Some `EVM` opcodes consume a constant gas fee that depends on some condition. Our current approach, to make the analyzer sound, is to keep the more expensive fee. We plan to improve the gas model in order to infer more accurate bounds, e.g., in the case of `SSTORE` that consumes 20,000 units of gas if we are writing in a position that previously contained a 0 or 5,000 otherwise.

- Finally, as in the Solidity compiler, we are measuring the gas consumption of the analyzed instructions and not adding to our analysis the so called *intrinsic gas* cost of the execution. The intrinsic gas includes a transaction fee of 21,000 plus an amount that depends on the input data of the transaction and another amount that

might be added for contract-creating transactions. We plan to add a flag to our tool to optionally include this cost as well.

# Bibliography


[1] Bamboo. https://github.com/pirapira/bamboo.

[2] DAO smart contract. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[3] Laser-Ethereum. https://github.com/b-mueller/laser-ethereum.

[4] Serpent. https://github.com/ethereum/wiki/wiki/Serpent.

[5] Solidity. https://solidity.readthedocs.io/en/develop.

[6] Vyper. https://github.com/ethereum/vyper.

[7] The EthereumPot contract, 2017. https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7.

[8] Mythril, 2018. https://github.com/b-mueller/mythril.

[9] Oyente: An Analysis Tool for Smart Contracts, 2018. https://github.com/melonproject/oyente.

[10] Rattle - an evm binary static analysis framework, 2018. https://github.com/crytic/rattle.

[11] Competition on Software Verification Rules, 2019. https://sv-comp.sosy-lab.org/2019/rules.php.

[12] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.

[13] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[14] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla an d Guillermo Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.

[15] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.

[16] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[17] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19)*, ACM, pages 386–389, 2019.

[18] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Peak Cost Analysis of Distributed Systems. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014.

[19] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2012.

[20] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2013.

[21] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Transactions on Computational Logic*, 17(2):11:1–11:39, March 2016.

[22] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In Sandrine Blazy and Thomas P. Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2015.

[23] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-Happen-in-Parallel Analysis with Returned Futures. In Deepak D'Souza and K.Narayan Kumar, editors, *15th International Symposium on Automated Technology for Verification and Analysis, ATVA 2017*, volume 10482 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2017.

[24] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A Framework for High-Level Analysis of Ethereum Bytecode. In Shuvendu Lahiri and Chao Wang, editors, *16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018. Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 513–520. Springer, 2018.

[25] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas vulnerabilitires in Ethereum Smart Contracts using Static Resource Analysis. In *13th International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS 2019. Proceedings*, volume 11847 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2019.

[26] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 66–77. ACM, 2018.

[27] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[28] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2005.

[29] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A Tool for Configurable Software Verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[30] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In Toby C. Murray and Deian Stefan, editors, *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for*

*Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 91–96. ACM, 2016.

[31] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[32] A. Bouajjani and M. Emmi. Analysis of Recursively Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(3):10, 2013.

[33] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.

[34] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016.

[35] Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Compositional Safety Verification with Max-SMT. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 33–40. IEEE, 2015.

[36] Che-Wei Chang and Rainer Dömer. May-happen-in-parallel analysis of ESL models using UPPAAL model checking. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 1567–1570. ACM, 2015.

[37] Congming Chen, Wei Huo, Lung Li, Xiaobing Feng, and Kai Xing. Can We Make It Faster? Efficient May-Happen-in-Parallel Analysis Revisited. In Hong Shen, Yingpeng Sang, Yidong Li, Depei Qian, and Albert Y. Zomaya, editors, *13th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012*, pages 59–64. IEEE, 2012.

[38] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 442–446. IEEE Computer Society, 2017.

[39] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In Andrea Zisman and Sven Apel, editors, *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 81–84. ACM, 2018.

[40] Weiwei Chen, Xu Han, and Rainer Dömer. May-happen-in-parallel analysis based on segment graphs for safe ESL models. In Gerhard P. Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[41] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Safe Runtime Verification of Real-Time Properties. In Joël Ouaknine and Frits W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, volume 5813 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2009.

[42] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.7*, 2018.

[43] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[44] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In Rocco de Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, March 2007.

[45] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[46] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. Region-Based May-Happen-in-Parallel Analysis for C Programs. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*, pages 889–898. IEEE Computer Society, 2015.

[47] Joshua Ellul and Gordon J. Pace. Runtime verification of ethereum smart contracts. In *14th European Dependable Computing Conference, EDCC 2018, Iaşi, Romania, September 10-14, 2018*, pages 158–163. IEEE Computer Society, 2018.

[48] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[49] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, June 2013.

[50] Antonio Flores-Montoya and Reiner Hähnle. Resource Analysis of Complex Programs with Cost Equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[51] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1176–1186. IEEE / ACM, 2019.

[52] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.

[53] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 243–269. Springer, 2018.

[54] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. EtherTrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, 2018.

[55] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL):48:1–48:28, 2018.

[56] Carole D. Hafner and Kurt Godden. Portability of syntax and semantics in datalog. *ACM Trans. Inf. Syst.*, 3(2):141–164, 1985.

[57] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.

[58] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. 2017.

[59] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM*

*SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 357–370. ACM, 2011.

[60] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[61] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[62] Temesghen Kahsai, Jorge A. Navas, Arie Gurfinkel, and Anvesh Komuravelli. The SeaHorn Verification Framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[63] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[64] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.*, pages 363–373. ACM, 2019.

[65] Johannes Krupp and Christian Rossow. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1317–1333. USENIX Association, 2018.

[66] Jonathan K. Lee and Jens Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Principles and Practice of Parallel Programming (PPoPP'10)*, pages 25–36, New York, NY, USA, 2010. ACM.

[67] Jonathan K. Lee, Jens Palsberg, Rupak Majumdar, and Hong Hong. Efficient may happen in parallel analysis for async-finish parallelism. In Antoine Miné and David Schmidt, editors, *19th International Symposium on Static Analysis (SAS 2012)*, volume 7460, pages 5–23. Springer, 2012.

[68] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors,

*Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*, volume 3602 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2004.

[69] Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In Davide Ancona, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 51–62. ACM, 2017.

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[71] Magnus Madsen, Ondrej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017.

[72] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 450–465. Springer, 2018.

[73] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. *CoRR*, abs/1711.09327, 2017.

[74] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases on Inf. Techn.*, 21(1):19–32, 2019.

[75] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit.(2018). In *The 9th annual HITB Security Conference*, 2018.

[76] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014.

[77] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[78] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[79] Gleb Naumovich and George S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statement That May Happen in Parallel. In *SIGSOFT '98, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena Vista, Florida, USA, November 3-5, 1998*, pages 24–34. ACM, 1998.

[80] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An Efficient Algorithm for Computing *MHP* Information for Concurrent Java Programs. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 1999.

[81] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 653–663. ACM, 2018.

[82] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[83] Terence John Parr and Russell W. Quong. ANTLR: A Predicated- *LL(k)* Parser Generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[84] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[85] Aravind Sankar, Soham Chakraborty, and V. Krishna Nandivada. Improved MHP Analysis. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 207–217. ACM, 2016.

[86] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[87] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[88] Lars Stegeman. Solitor: runtime verification of smart contracts on the ethereum network. Master's thesis, University of Twente, 2018.

[89] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub,

Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[90] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, 1983.

[91] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical Security Analysis of Smart Contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.

[92] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999.

[93] Frédéric Henri Vogel. IRSRI: An intermediate representation for smart contracts. Master's thesis, Department of Computer Science, ETH Zurich, 2019.

[94] Ben Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.

[95] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

# Part II

# Papers of the Thesis

# Chapter 7

# Publications

**List of Papers (in chronological order):**

1. Elvira Albert, Samir Genaim and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015*. Proceedings, volume 9291 of Lecture Notes in Computer Science, pages 72-89. Springer, September 2015.

2. Elvira Albert, Samir Genaim and Pablo Gordillo. May-Happen-in-Parallel Analysis with Returned Futures. In *15th International Symposium on Automated Technology for Verification and Analysis, ATVA 2017*. Proceedings, volume 10482 of Lecture Notes in Computer Science, pages 42-58. Springer, October 2017.

3. Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio and Ilya Sergey. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018*. Proceedings, volume 11138 of Lecture Notes in Computer Science, pages 513-520. Springer, October 2018.

4. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*. ACM, pages 386–389, July 2019.

5. Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on Fumes: Preventing Out-Of-Gas Vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In *13th International Conference on Verification and Evaluation of Computer and Communication Systems 2019, VECoS 2019*. Proceedings, volume 11847 of Lecture Notes in Computer Science, pages 63-78. Springer, October 2019.

# May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization

Elvira Albert, Samir Genaim, and Pablo Gordillo( )

Complutense University of Madrid (UCM), Madrid, Spain
pabgordi@ucm.es

**Abstract.** A may-happen-in-parallel (MHP) analysis computes pairs of program points that may execute *in parallel* across different distributed components. This information has been proven to be essential to infer both safety properties (e.g., deadlock freedom) and liveness properties (e.g., termination and resource boundedness) of asynchronous programs. Existing MHP analyses take advantage of the synchronization points to learn that one task has finished and thus will not happen in parallel with other tasks that are still active. Our starting point is an existing MHP analysis developed for *intra-procedural* synchronization, i.e., it only allows synchronizing with tasks that have been spawned inside the current task. This paper leverages such MHP analysis to handle *inter-procedural* synchronization, i.e., a task spawned by one task can be awaited within a different task. This is challenging because task synchronization goes beyond the boundaries of methods, and thus the inference of MHP relations requires novel extensions to capture inter-procedural dependencies. The analysis has been implemented and it can be tried online.

## 1 Introduction

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which *asynchronous* tasks can execute concurrently with their caller tasks, and their callers can explicitly wait for their completion. Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables for synchronization [7,8]. In this model, a method call m on some parameters $\overline{x}$, written as f=m($\overline{x}$) , spawns an asynchronous task. Here, f is a *future variable* which allows synchronizing with the termination of the task executing m. The instruction **await** f? allows checking whether m has finished, and blocks the execution of the current task if m is still running. As concurrently-executing tasks interleave their accesses to shared memory, asynchronous

This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.


S. Blazy and T. Jensen (Eds.): SAS 2015, LNCS 9291, pp. 72–89, 2015.
DOI: 10.1007/978-3-662-48288-9_5

programs are prone to concurrency-related errors [6]. Automatically proving safety and liveness properties still remains a challenging endeavor today.

MHP is an analysis of utmost importance to ensure both liveness and safety properties of concurrent programs. The analysis computes MHP *pairs*, which are pairs of program points whose execution might happen in parallel across different distributed components. In this fragment of code f=m(..);...; **await** f?; the execution of the instructions of the asynchronous task m may happen in parallel with the instructions between the asynchronous call and the **await**. However, due to the **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. This piece of information is fundamental to prove more complex properties: in [9], MHP pairs are used to discard unfeasible deadlock cycles; in [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. As a simple example, consider a procedure g that contains as unique instruction y=-1, where y is a global variable. The following loop y=1; **while**(i>0){i=i-y;} might not terminate if g runs in parallel with it, since g can modify y to a negative value and the loop counter will keep on increasing. However, if we can guarantee that g will not run in parallel with this code, we can ensure termination and resource-boundedness for the loop.

This paper leverages an existing MHP analysis [3] developed for intra-procedural synchronization to the more general setting of inter-procedural synchronization. This is a fundamental extension because it allows synchronizing with the termination of a task outside the scope in which the task is spawned, as it is available in most concurrent languages. In the above example, if task g is awaited outside the boundary of the method that has spawned it, the analysis of [3] assumes that it may run in parallel with the loop and hence it fails to prove termination and resource boundedness. The enhancement to inter-procedural synchronization requires the following relevant extensions to the analysis:

1. *Must-Have-Finished Analysis* (MHF): the development of a novel MHF analysis which infers *inter-procedural dependencies* among the tasks. Such dependencies allow us to determine that, when a task finishes, those that are awaited for on it must have finished as well. The analysis is based on using Boolean logic to represent abstract states and simulate corresponding operations. The key contribution is the use of logical implication to delay the incorporation of procedure summaries until synchronization points are reached. This addresses a challenge in the analysis of asynchronous programs.
2. *Local MHP Phase*: the integration of the above MHF information in the local phase of the original MHP analysis in which methods are analyzed locally, i.e., without taking indirect calls into account. This will require the use of richer analysis information in order to consider the inter-procedural dependencies inferred in point 1 above.
3. *Global MHP phase*: the refinement of the global phase of the MHP analysis – where the information of the local MHP analysis in point 2 is composed– in order to eliminate spurious MHP pairs which appear when inter-procedural dependencies are not tracked. This will require to refine the way in which MHP pairs are computed.

We have implemented our approach in SACO [2], a static analyzer for concurrent objects which is able to infer the aforementioned liveness and safety properties. The system can be used online at http://costa.ls.fi.upm.es/saco/web, where the examples used in the paper are also available.

## 2 Language

Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables [7,8]. It also includes conditional and loop constructs, however, conditions in these constructs are simply non-deterministic choices. Developing the analysis at such abstract level is convenient [11], since the actual computations are simply ignored in the analysis and what is actually tracked is the control flow that originates from asynchronously calling methods and synchronizing with their termination. Our implementation, however, is done for the full concurrent object-oriented language ABS [10] (see Sect. 6).

A program $P$ is a set of methods that adhere to the following grammar:

$$
\begin{array}{lll}
M & ::= & m(\bar{x})\ \{s\} \\
s & ::= & \epsilon \mid b; s \\
b & ::= & \textbf{if}\ (*)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{while}\ (*)\ \textbf{do}\ s \mid y = m(\bar{x}) \mid \textbf{await}\ x? \mid \textbf{skip}
\end{array}
$$

Here all variables are future variables, which are used to synchronize with the termination of the called methods. Those future variables that are used in a method but are not in its parameters are the *local future variables* of the method (thus we do not need any special instruction for declaring them). In loops and conditions, the symbol $*$ stands for non-deterministic choice (*true* or *false*). The instruction $y = m(\bar{x})$ creates a new task which executes method $m$, and binds the future variable $y$ with this new task so we can synchronize with its termination later. Inter-procedural synchronization is realized in the language by passing future variables as parameters, since the method that receives the future variable can await for the termination of the associated task (created outside its scope). For simplifying the presentation, we assume that *method parameters are not modified inside each method*. For a method $m$, we let $P_m$ be the set of its parameters, $L_m$ the set of its local variables, and $V_m = P_m \cup L_m$.

The instruction **await** $x?$ blocks the execution of the current task until the task associated with $x$ terminates. Instruction **skip** has no effect, it is simply used when abstracting from a richer language, e.g., ABS in our case, to abstract instructions such as assignments. Programs should include a method main from which the execution (and the analysis) starts. We assume that instructions are labeled with unique identifiers that we call program points. For **if** and **while** the identifier refers to the corresponding condition. We also assume that each method has an exit program point $\ell_m$. We let $\texttt{ppoints}(m)$ and $\texttt{ppoints}(P)$ be the sets of program points of method $m$ and program $P$, resp., $I_\ell$ be the instruction at program point $\ell$, and $\texttt{pre}(\ell)$ be the set of program points preceding $\ell$.

Next we define a formal (interleaving) operational semantics for our language. A task is of the form $tsk(tid, l, s)$ where $tid$ is a unique identifier, $l$ is a mapping

$$(\textsc{skip}) \quad \frac{}{tsk(tid, l, \textbf{skip}; s) \rightsquigarrow tsk(tid, l, s)}$$

$$(\textsc{if}) \quad \frac{b \equiv \textbf{if } (*) \textbf{ then } s_1 \textbf{ else } s_2, \text{ set } s' \text{ non-deterministically to } s_1; s \text{ or } s_2; s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{loop}) \quad \frac{b \equiv \textbf{while } (*) \textbf{ do } s_1, \text{ set } s' \text{ non-deterministically to } s_1; b; s \text{ or } s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{call}) \quad \frac{\bar{z} \text{ are the formal parameters of } m,\ tid' \text{ is a fresh id, } l' = \{z_i \mapsto l(x_i)\}}{tsk(tid, l, y = m(\bar{x}); s) \rightsquigarrow tsk(tid, l[y \mapsto tid'], s), tsk(tid', l', body(m))}$$

$$(\textsc{await}) \quad \frac{l(x) = tid'}{tsk(tid, l, \textbf{await } x?; s), tsk(tid', l', \epsilon) \rightsquigarrow tsk(tid, l, s), tsk(tid', l', \epsilon)}$$

**Fig. 1.** Derivation rules

from local variables and parameters to task identifiers, and $s$ is a sequence of instructions. Local futures are initialized to the special value $\perp$ which is the default value for future variable (i.e., $\perp$ like **null** for reference variables in Java). A state $S$ is a set of tasks that are executing in parallel. From a state $S$ we can reach a state $S'$ in one execution step, denoted $S \rightsquigarrow S'$, if $S$ can be rewritten using one of the derivation rules of Fig. 1 as follows: if the conclusion of the rule is $A \rightsquigarrow B$ such that $A \subseteq S$ and the premise holds, then $S' = (S \setminus A) \cup B$. The meaning of the derivation rules is quite straightforward: (SKIP) advances the execution of the corresponding task to the next instruction; (IF) nondeterministically chooses between one of the branches; (LOOP) nondeterministically chooses between executing the loop body or advancing to the instruction after the loop; (CALL) creates a new task with a fresh identifier $tid'$, initializes the formal parameters $\bar{z}$ of $m$ to those of the actual parameters $\bar{x}$, sets future variable $y$ in the calling task to $tid'$, so one can synchronize with its termination later (other local futures of $m$ are assumed to have the special value $\perp$); and (AWAIT) advances to the next instruction if the task associated to $x$ has terminated already. Note that when a task terminates, it does not disappear from the state but rather its sequence of instructions remains empty.

An execution is a sequence of states $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_n$, sometimes denoted as $S_0 \rightsquigarrow^* S_n$, where $S_0 = \{tsk(0, l, body(\mathsf{main}))\}$ is an initial state which includes a single task that corresponds to method main, and $l$ is an empty mapping. At each step there might be several ways to move to the next state depending on the task selected, and thus executions are nondeterministic.

In what follows, given a task $tsk(tid, l, s)$, we let $pp(s)$ be the program point of the first instruction in $s$. When $s$ is an empty sequence, $pp(s)$ refers to the exit program point of the corresponding method. Given a state $S$, we define its set of MHP pairs, i.e., the set of program points that execute in parallel in $S$, as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, l_1, s_1), tsk(tid_2, l_2, s_2) \in S, tid_1 \neq tid_2\}$. The set of MHP pairs for a program $P$ is then defined as the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \rightsquigarrow^* S_n\}$.

*Example 1.* Figure 2 shows some examples in our language, where $\mathsf{m}_1$, $\mathsf{m}_2$ and $\mathsf{m}_3$ are main methods. The following are some steps in a possible derivation for $\mathsf{m}_2$:

$$\mathbf{S_0} \equiv tsk(0, \emptyset, body(\mathsf{m}_2)) \rightsquigarrow^* \mathbf{S_1} \equiv tsk(0, [x \mapsto 1], \{16, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})) \rightsquigarrow^*$$
$$\mathbf{S_2} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{18, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})), tsk(2, [w \mapsto 1], body(\mathsf{g})) \rightsquigarrow^*$$
$$\mathbf{S_3} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{19, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], body(\mathsf{g})) \rightsquigarrow^*$$
$$\mathbf{S_4} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{20, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], \epsilon) \rightsquigarrow \ldots$$

In $S_1$ we execute until the asynchronous call to f which creates a new task identified as 1 and binds x to this new task. In $S_2$ we have executed the skip and the asynchronous invocation to g that adds in the new task the binding of the formal parameter w to the task identified as 1. In $S_3$ we proceed with the execution of the instructions in $\mathsf{m}_2$ until reaching the **await** that blocks this task until g terminates. Also, in $S_3$ we have executed entirely f (denoted by $\epsilon$). $S_4$ proceeds with the execution of g whose **await** can be executed since task 1 is at its exit point $\epsilon$. We have the following MHP pairs in this fragment of the derivation, among many others: from $S_1$ we have (16,35) that captures that the first instruction of f executes in parallel with the instruction 16 of m2, from $S_2$ we have (18,35) and (18,38). The important point is that we have no pair (20,35) since when the **await** at L19 executes at $S_4$, it is guaranteed that f has finished. This is due to the inter-procedural dependency at L39 of g where the task f is awaited: variable x is passed as argument to g, which allows g to synchronize with the termination of f at L39 even if f was called in a different method.

## 3 An Informal Account of Our Method

In this section, we provide an overview of our method by explaining the analysis of $\mathsf{m}_2$. Our goal is to infer precise MHP information that describes, among others, the following representative cases: (1) any program point of g cannot run in parallel with L20, because at L19 method $\mathsf{m}_2$ awaits for g to terminate; (2) L35 cannot run in parallel with L20, since when waiting for the termination of g at L19 we know that f *must-have-finished* as well due to the *dependency* relation that arises when $\mathsf{m}_2$ implicitly waits for the termination of f; and (3) L35 cannot run in parallel with L40, because f *must-have-finished* due to the synchronization on the local future variable w at L39 that refers to future variable x of $\mathsf{m}_2$.

Let us first informally explain which MHP information the analysis of [3] is able to infer for $\mathsf{m}_2$, and identify the reasons why it fails to infer some of the desired information. The analysis of [3] is carried out in two phases: (1) each method is *analyzed separately* to infer local MHP information; and (2) the local information is used to construct a global MHP graph from which MHP pairs are extracted by checking reachability conditions among the nodes.

The local analysis infers, for each program point, a *multiset* of MHP atoms where each atom describes a task that might be executing in parallel when reaching that program point, but only considering tasks that have been invoked directly in the analyzed method. An atom of the form $x{:}\tilde{m}$ indicates that there might be an *active* instance of $m$ executing at any of its program points, and is

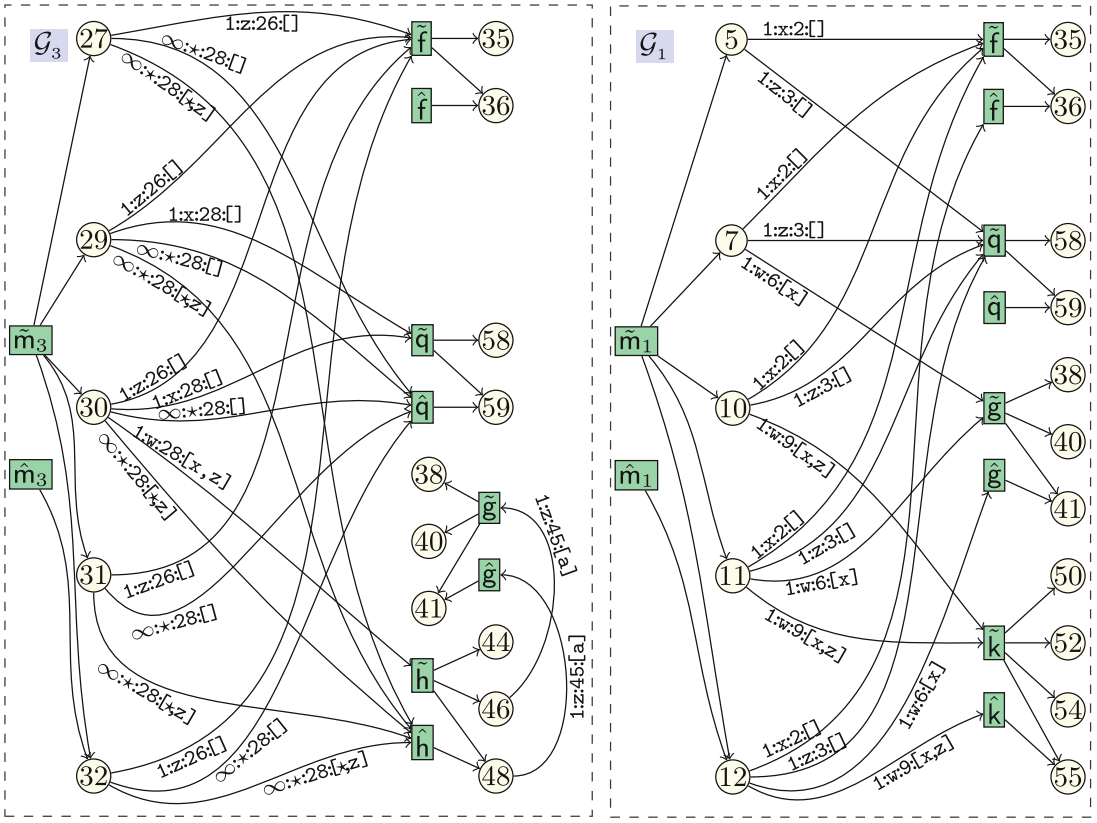
```
1  m1() {
2    x=f();
3    z=q();
4    skip
5    if (*) then
6      w=g(x);
7      skip;
8    else
9      w=k(x,z);
10     skip;
11   await w?;
12 }
```

```
13 m2() {
14   skip;
15   x=f();
16   skip;
17   z=g(x);
18   skip;
19   await z?;
20   skip;
21 }
22
23
24
```

```
25 m3() {
26   z=f();
27   while (*)
28     x=q();
29     w=h(x,z);
30     await w?;
31     skip;
32 }
33
34 f() {
35   skip;
36 }
```

```
37 g(w) {
38   skip;
39   await w?
40   skip;
41 }
42
43 h(a,b) {
44   skip;
45   z=g(a);
46   skip;
47   await z?;
48 }
```

```
49 k(a,b) {
50   skip;
51   await a?;
52   skip;
53   await b?;
54   skip;
55 }
56
57 q() {
58   skip;
59 }
60
```



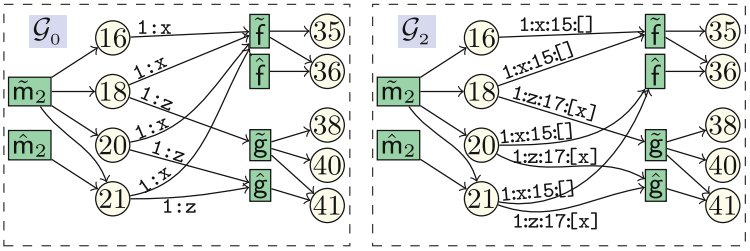




**Fig. 2.** (TOP) Examples for MHP analysis ($\mathsf{m}_1$, $\mathsf{m}_2$, $\mathsf{m}_3$ are main methods). (BOTTOM) MHP graph $\mathcal{G}_i$ corresponds to analyzing $\mathsf{m}_i$, and $\mathcal{G}_0$ to analyzing $\mathsf{m}_2$ as in [3].

bound to the future variable $x$. An atom of the form $x{:}\hat{m}$ differs from the previous one in that $m$ must be at its exit program point, i.e., has finished executing already. For method $\mathsf{m}_2$, the local MHP analysis infers, among others, $\{\mathsf{x}{:}\tilde{\mathsf{f}}\}$ for L16, $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\tilde{\mathsf{g}}\}$ for L18, and $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\hat{\mathsf{g}}\}$ for L20 and L21, because $\mathsf{g}$ has been awaited locally. Observe that the sets of L20 and L21 include $\mathsf{x}{:}\tilde{\mathsf{f}}$ and not $\mathsf{x}{:}\hat{\mathsf{f}}$, although method $\mathsf{f}$ has finished already when reaching L20 and L21 (since $\mathsf{g}$ has finished). This information cannot be inferred by the local analysis of [3] since it is applied to each method separately, ignoring ($a$) indirect (non-local) calls and ($b$) inter-procedural synchronizations. In the sequel we let $\Psi_\ell$ be the result of the local MHP analysis for program point $\ell$.

In the second phase, the analysis of [3] builds an MHP graph whose purpose is to capture MHP relations due to indirect calls (point ($a$) above). The graph $\mathcal{G}_0$ depicted in Fig. 2 for $\mathsf{m}_2$ is constructed as follows: (1) every program point $\ell$ contributes a node labeled with $\ell$ – for simplicity we include only program points of interest; (2) every method $m$ contributes two nodes $\tilde{m}$ and $\hat{m}$, where $\tilde{m}$ is connected to all program point nodes of $m$ to indicate that when $m$ is active, it can be executing at any of its program points, and $\hat{m}$ is connected only to the exit program point of $m$; and (3) if $x{:}\tilde{m}$ (resp. $x{:}\hat{m}$) is an atom of $\Psi_\ell$ with multiplicity $i$, i.e., it appears $i$ times in the multiset $\Psi_\ell$, we create an edge from $\ell$ to $\tilde{m}$ (resp. $\hat{m}$) and label it with $i{:}x$. Note such edge actually represents $i$ identical edges, i.e., we could copy the edge $i$ times and omit the label $i$.

Roughly, the MHP pairs are obtained from $\mathcal{G}_0$ using the following principle: program points $(\ell_1, \ell_2)$ might execute in parallel if there is a path from $\ell_1$ to $\ell_2$ or vice versa (direct MHP pair); or if there is a program point $\ell_3$ such that there are paths from $\ell_3$ to $\ell_1$ and to $\ell_2$ (indirect MHP pair), and the first edge of both paths is labeled with two different future variables. When two paths are labeled with the same future variable, it is because there is a disjunction (e.g., from an if-then-else) and only one of the paths might actually occur. Applying this principle to $\mathcal{G}_0$, we can conclude that L20 cannot execute in parallel with any program point of $\mathsf{g}$, which is precise as expected, and that L20 can execute in parallel with L35 which is imprecise. This imprecision is attributed to the fact that the MHP analysis of [3] does not track inter-method synchronizations.

In order to overcome the imprecision, we develop a must-have-finished analysis that captures inter-method synchronizations, and use it to improve the two phases of [3]. This analysis would infer, for example, that "*when reaching* L40, *it is guaranteed that whatever task bound to* $\mathsf{w}$ *has finished already*", and that "*when reaching* L20, *it is guaranteed that whatever tasks bound to* $\mathsf{x}$ *and* $\mathsf{z}$ *have finished already*". By having this information at hand, the first phase of [3] can be improved as follows: when analyzing the effect of **await** $\mathsf{z}$? at L20, we change the status of both $\mathsf{g}$ and $\mathsf{f}$ to finished, because we know that any task bound $\mathsf{z}$ and $\mathsf{x}$ has finished already. In addition, we modify the MHP atoms as follows: an MHP atom will be of the form $y{:}\ell{:}\tilde{m}(\bar{x})$ or $y{:}\ell{:}\hat{m}(\bar{x})$, where the new information $\ell$ and $\bar{x}$ are the calling site and the parameters passed to $m$. The need for this extra information will become clear later in this section. In summary, the modified first phase will infer $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}()\}$ for L16, $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}(), \mathsf{z}{:}17{:}\tilde{\mathsf{g}}(\mathsf{x})\}$ for L18, and $\{\mathsf{x}{:}15{:}\hat{\mathsf{f}}(), \mathsf{z}{:}17{:}\hat{\mathsf{g}}(\mathsf{x})\}$ for L20 and L21.

In the second phase of the analysis: (i) the construction of the MHP graph is modified to use the new local MHP information; and (ii) the principle used to extract MHP pairs is modified to make use to the must-have-finished information. The new MHP graph constructed for $\mathsf{m}_2$ is depicted in Fig. 2 as $\mathcal{G}_2$. Observe that the labels on the edges include the new information available in the MHP atoms. Importantly, the spurious MHP information that is inferred by [3] is not included in this graph: (1) in contrast to $\mathcal{G}_0$, $\mathcal{G}_2$ does not include edges from nodes 20 and 21 to $\tilde{\mathsf{f}}$, but to $\hat{\mathsf{f}}$. This implies that L35 cannot run in parallel with L20 or L21; (2) in $\mathcal{G}_2$, we still have paths from 18 to 35 and 40, which means, if the old principle for extracting MHP pairs is used, that L35 and L40 might happen in parallel. The main point is that, using the labels on the edges, we know that the first path uses a call to f that is bound to x, and that this same x is passed to g, using the parameter w, in the first edge of the second path. Now since the must-have-finished analysis tell us that at L40 any task bound w is finished already, we conclude that f must be at its exit program point when the execution reaches L40, and thus the MHP pair (35,40) is spurious because L35 is not an exit program point of f. This last point explains why the MHP atoms are designed to include the actual parameters of method calls.

## 4 Must-Have-Finished Analysis

In this section we present a novel inter-procedural Must-Have-Finished (MHF) analysis that can be used to compute, for each program point $\ell$, a set of *finished future variables*, i.e., whenever $\ell$ is reached those variables are either not bound to any task (i.e., have the default value $\perp$) or their bound tasks are guaranteed to have terminated. We refer to such sets as MHF sets.

*Example 2.* The following are MHF sets for the program points of Fig. 2:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| L2: {x,w,z} | L9 : {w} | L16: {z} | L26: {x,z,w} | L32: {x,w} | L41: {w} | L50: {} | L58: {} |
| L3: {z,w} | L10: {} | L17: {z} | L27: {x,w} | L35: {} | L44: {z} | L51: {} | L59: {} |
| L4: {w} | L11: {} | L18: {} | L28: {x,w} | L36: {} | L45: {z} | L52: {a} | |
| L5: {w} | L12: {x,w} | L19: {} | L29: {w} | L38: {} | L46: {} | L53: {a} | |
| L6: {w} | L14: {x,z} | L20: {x,z} | L30: {} | L39: {} | L47: {} | L54: {a,b} | |
| L7: {} | L15: {x,z} | L21: {x,z} | L31: {x,w} | L40: {w} | L48: {a,z} | L55: {a,b} | |

Here, at program points that correspond to method entries, all local variables (but not the parameters) are finished since they point to no task. For g: at L38 and L39 no task is guaranteed to have finished, because the task bound to w might be still executing; at L40 and L41, since we passed through **await** w? already, it is guaranteed that w is finished. For k: at L50 and L51 no task is guaranteed to have finished; at L52 and L53 a is finished since we already passed through **await** a?; and at L54 and L55 both a and b are finished. For $\mathsf{m}_1$: at L12 both w and x are finished. Note that w is finished because of **await** w? , and x is finished due to the implicit dependency between the termination of x and w.

### 4.1 Definition of MHF

By carefully examining the MHF sets of Example 2, we can see that an analysis that simply tracks MHF sets would be imprecise. For example, since the MHF set at L11 is empty, the only information we can deduce for L12 is that w is finished. To deduce that x is finished we must track the implicit dependency between w and x. Next we define a more general MHF property that captures such dependencies, and from which we can easily compute the MHF sets.

**Definition 1.** *Given a program point* $\ell \in \texttt{ppoints}(P)$*, we let* $\mathcal{F}(\ell) = \{f(S_i, l) \mid S_0 \rightsquigarrow^* S_i, tsk(tid, l, s) \in S_i, pp(s) = \ell\}$ *where* $f(S, l) = \{x \mid x \in dom(l),\ l(x) = \bot \vee (l(x) = tid' \wedge tsk(tid', l', \epsilon) \in S)\}$.

Intuitively, $f(S, l)$ is the set of all future variables, from those defined in $l$, whose corresponding tasks are finished in $S$. The set $\mathcal{F}(\ell)$ considers all possible ways of reaching $\ell$, and for each one it computes a corresponding set $f(S, l)$ of finished future variables. Thus, $\mathcal{F}(\ell)$ describes all possible sets of finished future variables when reaching $\ell$. The set of *all* finished future variables at $\ell$ is then defined as $\texttt{mhf}(\ell) = \cap\{F \mid F \in \mathcal{F}(\ell)\}$, i.e., the intersection of all sets in $\mathcal{F}(\ell)$.

*Example 3.* The values of $\mathcal{F}(\ell)$ for selected program points from Fig. 2 are:

| | | |
|---|---|---|
| L5 : {{w,x,z},{w,z},{w,x},{w}} | L31: {{w,x,z},{w,x}} | L46: {{a,z},{a},{}, |
| L11: {{w,x,z},{w,x},{x,z},{z},{x},{}} | L32: {{w,x,z},{w,x}} | {a,b,z},{a,b},{b}} |
| L12: {{w,x,z},{w,x}} | L35: {{}} | L48: {{a,z},{a,b,z}} |
| L20: {{x,z}} | L38: {{w},{}} | L52: {{a},{a,b}} |
| L27: {{w,x,z},{w,x}} | L40: {{w}} | L54: {{a,b}} |
| L30: {{w,x,z},{w,x},{x,z},{x},{z},{}} | | L58: {{}} |

In L5 different sets arise by considering all possible orderings in the execution of tasks f, q and $\texttt{m}_1$, but $\texttt{mhf}(L5) = \{\texttt{w}\}$. Note that for any $F \in \mathcal{F}(11)$, if $\texttt{w} \in F$ then $\texttt{x} \in F$, which means that if w is finished at L11, then x must have finished.

### 4.2 An Analysis to Infer MHF Sets

Our goal is to infer $\texttt{mhf}(\ell)$, or a subset of it, for each $\ell \in \texttt{ppoints}(P)$. Note that any set $X$ that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq X$, can be used to compute a subset of $\texttt{mhf}(\ell)$, because $\cap\{F \mid F \in X\} \subseteq \cap\{F \mid F \in \mathcal{F}(\ell)\}$. In the rest of this section we develop an analysis to over-approximate $\mathcal{F}(\ell)$. We will use Boolean formulas, whose models naturally represent MHF sets, and Boolean connectives to smoothly model the abstract execution of the different instructions.

An MHF state for the program points of a method $m$ is a propositional formula $\Phi : V_m \mapsto \{true, false\}$ of the form $\vee_i \wedge_j c_{ij}$, where an atomic proposition $c_{ij}$ is either $x$ or $y \rightarrow x$ such that $x \in V_m \cup \{true, false\}$ and $y \in L_m$. Intuitively, an atomic proposition $x$ states that $x$ is finished, and $y \rightarrow x$ states that if $y$ is finished then $x$ is finished as well. Note that we do not allow the parameters of $m$ to appear in the premise of an implication (we require $y \in L_m$). When $\Phi$

is *false* or of the form $\vee_j \wedge_j x_{ij}$ where $x_{ij}$ is a propositional variable, we call it monotone. Recall that $\sigma \subseteq V_m$ is a *model* of $\Phi$, iff an assignment that maps variables from $\sigma$ to *true* and other variables to *false* is a satisfying assignment for $\Phi$. The set of all models of $\Phi$ is denoted $[\![\Phi]\!]$. The set of all MHF states for $m$, together with the formulas *true* and *false*, is denoted $\mathcal{A}_m$.

*Example 4.* Assume $V_m = \{x, y, z\}$. The Boolean formula $x \vee y$ states that either $x$ or $y$ or both are finished, and that $z$ can be in any status. This information is precisely captured by the models $[\![x \vee y]\!] = \{\{x\},\{y\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\}\}$. The Boolean formula $z \wedge (x \rightarrow y)$ states that $z$ is finished, and if $x$ is finished then $y$ is finished. This is reflected in $[\![z \wedge (x \rightarrow y)]\!] = \{\{z\}, \{z, y\}, \{z, x, y\}\}$ since $z$ belongs to all models, and any model that includes $x$ includes $y$ as well. The formula *false* means that the corresponding program point is not reachable. The following MHF states correspond to some selected program points from Fig. 2:

| | | | | | | |
|---|---|---|---|---|---|---|
| $\Phi_5$ : w | $\Phi_{12}$: w ∧ x | $\Phi_{27}$: w ∧ x | $\Phi_{31}$: w ∧ x | $\Phi_{35}$: *true* | $\Phi_{40}$: w | $\Phi_{48}$: a ∧ z | $\Phi_{54}$: a ∧ b |
| $\Phi_{11}$: w→x | $\Phi_{20}$: x ∧ z | $\Phi_{30}$: w→x | $\Phi_{32}$: w ∧ x | $\Phi_{38}$: *true* | $\Phi_{46}$: z→a | $\Phi_{52}$: a | $\Phi_{58}$: *true* |

Note that the models $[\![\Phi_\ell]\!]$ coincide with $\mathcal{F}(\ell)$ from Example 3.

Now, we proceed to explain how the execution of the different instructions can be modeled with Boolean formulas. Let us first define some auxiliary operations. Given a variable $x$ and an MHF state $\Phi \in \mathcal{A}_m$, we let $\exists x.\Phi = \Phi[x \mapsto \mathit{true}] \vee \Phi[x \mapsto \mathit{false}]$, i.e., this operation eliminates variable $x$ from (the domain of) $\Phi$. Note that $\exists x.\Phi \in \mathcal{A}_m$ and that $[\![\Phi]\!] \models [\![\exists x.\Phi]\!]$. For a tuple of variables $\bar{x}$ we let $\exists \bar{x}.\Phi$ be $\exists x_1.\exists x_2. \ldots .\exists x_n.\Phi$, i.e., eliminate all variables $\bar{x}$ from $\Phi$. We also let $\bar{\exists} \bar{x}.\Phi$ stand for eliminating all variables but $\bar{x}$ from $\Phi$. Note that if $\Phi \in \mathcal{A}_m$ is monotone, and $x \in L_m$, then $x \rightarrow \Phi$ is a formula in $\mathcal{A}_m$ as well.

Given a program point $\ell$, an MHF state $\Phi_\ell$, and an instruction to execute $I_\ell$, our aim is to compute a new MHF state, denoted $\mu(I_\ell)$, that represents the effect of executing $I_\ell$ within $\Phi_\ell$. If $I_\ell$ is **skip**, then clearly $\mu(I_\ell) \equiv \Phi_\ell$. If $I_\ell$ is an **await** $x$? instruction, then $\mu(I_\ell)$ is $x \wedge \Phi_\ell$, which restricts the MHF state of $\Phi_\ell$ to those cases (i.e., models) in which $x$ is finished. If $I_\ell$ is a call $y = m(\bar{x})$, where $m$ is a method with parameters named $\bar{z}$, and, at the exit program point of $m$ we know that the MHF state $\Phi_{\ell_m}$ holds, then $\mu(I_\ell)$ is computed as follows:

- We compute an MHF state $\Phi_m$ that describes "what happens to tasks bound to $\bar{x}$ when $m$ terminates". This is done by projecting $\Phi_{\ell_m}$ onto the method parameters, and then renaming the formal parameters $\bar{z}$ to the actual parameters $\bar{x}$, i.e., $\Phi_m = (\bar{\exists} \bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]$ , where $[\bar{z}/\bar{x}]$ denotes the renaming.
- Now assume that $\xi$ is a new (future) variable to which $m$ is bound. Then $\xi \rightarrow \Phi_m$ states that "when $m$ terminates, $\Phi_m$ must hold". Note that it says nothing about $\bar{x}$ if $m$ has not terminated yet. It is also important to note that $\Phi_m$ is monotone and thus $\xi \rightarrow \Phi_m$ is a valid MHF state.
- Next we add $\xi \rightarrow \Phi_m$ to $\Phi_\ell$, eliminate (old) $y$ since the variable is rewritten, and rename $\xi$ to (new) $y$. Note that we use $\xi$ as a temporary variable just not to conflict with the old value of $y$.

The above reasoning is equivalent to $(\exists y.(\Phi_\ell \wedge (\xi \rightarrow (\bar{\exists}\bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]))[\xi/y]$, and is denoted by $\oplus(\Phi_\ell, y, \Phi_{\ell_m}, \bar{x}, \bar{z})$. Note that the use of logical implication $\rightarrow$, to abstractly simulate method calls, allows delaying the incorporation of the method summary $\Phi_m$ until corresponding synchronization point is reached.

*Example 5.* Let $\Phi_{11} = \mathsf{x} \rightarrow \mathsf{w}$ be the MHF state at L11. The effect of executing $I_{11}$, i.e., **await** w?, within $\Phi_{11}$ should eliminate all models that do not include w. This is done using $\mathsf{w} \wedge \Phi_{11}$ which results in $\Phi_{12} = \mathsf{w} \wedge \mathsf{x}$. Now let $\Phi_{29} = \mathsf{w}$ be the MHF state at L29. The effect of executing the instruction at L29, i.e., w=h(x,z), within $\Phi_{29}$ is defined as $\oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle)$ and computed as follows: (1) we restrict $\Phi_{48} = \mathsf{a} \wedge \mathsf{z}$ to the method parameters $\langle \mathsf{a}, \mathsf{b} \rangle$, which results in a; (2) we rename the formal parameters $\langle \mathsf{a}, \mathsf{b} \rangle$ to the actual ones $\langle \mathsf{x}, \mathsf{z} \rangle$ which results in $\Phi_{\mathsf{h}} = \mathsf{x}$; (3) we compute $\exists \mathsf{w}.(\Phi_{29} \wedge (\xi \rightarrow \Phi_{\mathsf{h}}))$, which results in $\xi \rightarrow \mathsf{x}$; and finally (4) we rename $\xi$ to w which results in $\Phi_{30} = \mathsf{w} \rightarrow \mathsf{x}$.

Next we describe how to generate a set of data-flow equations whose solutions associate to each $\ell \in \texttt{ppoints}(P)$ an MHF state $\Phi_\ell$ that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$. Each $\ell \in \texttt{ppoints}(P)$ contributes one equation as follows:

- if $\ell$ is not a method entry, we generate $\Phi_\ell = \vee\{\mu(\ell') \mid \ell' \in \texttt{pre}(\ell)\}$. This considers each program point $\ell'$ that immediately precedes $\ell$, computes the effect $\mu(\ell')$ of executing $I_{\ell'}$ within $\Phi_{\ell'}$, and the takes their disjunction;
- if $\ell$ is an entry of method $m$, we generate $\Phi_\ell = \wedge\{x \mid x \in L_m\}$, i.e., all local variables point to finished tasks (since they are mapped to $\bot$ when entering a method), and we do not know anything about the parameters.

The set of all equations for a program $P$ is denoted by $\mathcal{H}_P$.

*Example 6.* The following are the equations for the program points of $\mathsf{m}_3$:

$$\begin{array}{l|l|l|l} \Phi_{27} = \oplus(\Phi_{26}, \mathsf{z}, \Phi_{36}, \langle\rangle, \langle\rangle) \vee \Phi_{31} & \Phi_{28} = \Phi_{27} & \Phi_{29} = \oplus(\Phi_{28}, \mathsf{x}, \Phi_{59}, \langle\rangle, \langle\rangle) & \Phi_{31} = \mathsf{w} \wedge \Phi_{30} \\ \Phi_{30} = \oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle) & & \Phi_{26} = \mathsf{w} \wedge \mathsf{x} \wedge \mathsf{z} & \Phi_{32} = \Phi_{27} \end{array}$$

Note the circular dependency of $\Phi_{27}$ and $\Phi_{31}$ which originates from the corresponding while loop. Recall that $\mathsf{m}_3$ is a main method.

The next step is to solve $\mathcal{H}_P$, i.e., compute an MHF state $\Phi_\ell$, for each $\ell \in \texttt{ppoints}(P)$, such that $\mathcal{H}_P$ is satisfiable. This can be done iteratively as follows. We start from an initial solution where $\Phi_\ell = \mathit{false}$ for each $\ell \in \texttt{ppoints}(P)$. Then repeat the following until a fixed-point is reached: (1) substitute the current solution in the right hand side of the equations, and obtain new values for each $\Phi_\ell$; and (2) merge the new and old values of each $\Phi_\ell$ using $\vee$. E.g., solving the equation of Example 6, among other equations that were omitted, results in a

solution that includes, among others, the MHF states of Example 4. In what follows we assume that $\mathcal{H}_P$ has been solved, and let $\Phi_\ell$ be the MHF state at $\ell$ in such solution.

**Theorem 1.** *For any program point* $\ell \in \texttt{ppoints}(P)$*, we have* $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$.

In the rest of this article we let $\texttt{mhf}_\alpha(\ell) = \{x \mid x \in V_m,\ \Phi_\ell \models x\}$, i.e., the set of finished future variables at $\ell$ that is induced by $\Phi_\ell$. Theorem 1 implies $\texttt{mhf}_\alpha(\ell) \subseteq \texttt{mhf}(\ell)$. Computing $\texttt{mhf}_\alpha(\ell)$ using the MHF states of Example 4, among others that are omitted, results exactly in the MHF sets of Example 2.

## 5 MHP Analysis

In this section we present our MHP analysis, which is based on incorporating the MHF sets of Sect. 4 into the MHP analysis of [3]. In Sects. 5.1 and 5.2 we describe how we modify the two phases of the original analysis, and describe the gain of precision with respect to [3] in each phase.

### 5.1 Local MHP

The local MHP analysis (LMHP) considers each method $m$ separately, and for each $\ell \in \texttt{ppoints}(m)$ it infers an LMHP state that describes the tasks that might be executing when reaching $\ell$ (considering only tasks invoked in $m$). An LMHP state $\Psi$ is a *multiset* of MHP atoms, where each atom represents a task and can be: (1) $y{:}\ell'{:}\tilde{m}(\bar{x})$, which represents an *active* task that might be at any of its program points, including the exit one, and is bound to future variable $y$. Moreover, this task is an instance of method $m$ that was called at program point $\ell'$ (the *calling site*) with future parameters $\bar{x}$; or (2) $y{:}\ell'{:}\hat{m}(\bar{x})$, which differs from the previous one in that the task can only be at the exit program point, i.e., it is a *finished* task. In both cases, future variables $y$ and $\bar{x}$ can be $\star$, which is a special symbol indicating that we have no information on the future variable.

Intuitively, the MHP atoms of $\Psi$ represent (local) tasks that are executing in parallel. However, since a variable $y$ cannot be bound to more than one task at the same time, atoms bound to the same variable represent mutually exclusive tasks, i.e., cannot be executing at the same time. The same holds for atoms that use *mutually exclusive calling sites* $\ell_1$ and $\ell_2$ (i.e., there is no path from $\ell_1$ and $\ell_2$ and vice versa). The use of multisets allows including the same atom several times to represent different instances of the same method. We let $(a, i) \in \Psi$ indicate that $a$ appears $i$ times in $\Psi$. Note that $i$ can be $\infty$, which happens when the atom corresponds to a calling site inside a loop, this guarantees convergence of the analysis. Recall that the MHP atoms of [3] do not use the parameters $\bar{x}$ and the calling site $\ell'$, since they do not benefit from such extra information.

*Example 7.* The following are LMHP states for some program points from Fig. 2:

L5 : {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$()}
L7 : {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\tilde{\mathsf{g}}$(x)}
L10: {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:9:$\tilde{\mathsf{k}}$(x,z)}
L11: {x:2:$\tilde{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\tilde{\mathsf{g}}$(x),w:9:$\tilde{\mathsf{k}}$(x,z)}
L12: {x:2:$\hat{\mathsf{f}}$(),z:3:$\tilde{\mathsf{q}}$(),w:6:$\hat{\mathsf{g}}$(x),w:9:$\hat{\mathsf{k}}$(x,z)}
L16: {x:15:$\tilde{\mathsf{f}}$()}
L18: {x:15:$\tilde{\mathsf{f}}$(),z:17:$\tilde{\mathsf{g}}$(x)}
L20: {x:15:$\hat{\mathsf{f}}$(),z:17:$\hat{\mathsf{g}}$(x)}
L21: {x:15:$\hat{\mathsf{f}}$(),z:17:$\hat{\mathsf{g}}$(x)}
L27: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\tilde{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L29: L27 ∪ {x:28:$\tilde{\mathsf{q}}$()}
L30: L29 ∪ {w:29:$\tilde{\mathsf{h}}$(x,z)}
L31: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\hat{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L32: {z:26:$\tilde{\mathsf{f}}$(),(⋆:28:$\hat{\mathsf{q}}$(),∞),(⋆:29:$\hat{\mathsf{h}}$(⋆,z),∞)}
L44: {}
L46: {z:45:$\tilde{\mathsf{g}}$(a)}
L48: {z:45:$\hat{\mathsf{g}}$(a)}

Let us explain some of the above LMHP states. The state at L5 includes x:2:$\tilde{\mathsf{f}}$() and z:3:$\tilde{\mathsf{q}}$() for the active tasks invoked at L2 and L3. The state at L11 includes an atom for each task invoked in m1. Note that those of g and h are bound to the same future variable w, which means that only one of them might be executing at L11, depending on which branch of the **if** statement is taken. The state at L12 includes z:3:$\tilde{\mathsf{q}}$() since q might be active at L12 if we take the **then** branch of the **if** statement, and the other atoms correspond to tasks that are finished. The state at L27 includes z:26:$\tilde{\mathsf{f}}$() for the active task invoked at L26, and ⋆:28:$\hat{\mathsf{q}}$() and ⋆:29:$\hat{\mathsf{h}}$(⋆,z) with ∞ multiplicity for the tasks created inside the loop. Note that the first parameter of h is ⋆ since x is rewritten at each iteration.

The LMHP states are inferred by a *data-flow analysis* which is defined as a solution of a set of LMHP constraints obtained by applying the following transfer function $\tau$ to the instructions. Given an LMHP state $\Psi_\ell$, the effect of executing instruction $I_\ell$ within $\Psi_\ell$, denoted by $\tau(I_\ell)$, is defined as follows:

- if $I_\ell$ is a call $y = m(\bar{x})$, then $\tau(I_\ell) = \Psi_\ell[y/\star] \cup \{y{:}\ell'{:}\tilde{m}(\bar{x})\}$, which replaces each occurrence of $y$ by $\star$, since it is rewritten, and then adds a new atom $y{:}\ell{:}\tilde{m}(\bar{x})$ for the newly created task. E.g., the LMHP state of L30 in Example 7 is obtained from the one of L29 by adding w:29:$\tilde{\mathsf{h}}$(x,z) for the call at L29;
- if $I_\ell$ is **await** $y$?, and $\ell'$ is the program point after $\ell$, then we mark all tasks that are bound to a finished future variable as finished, i.e., $\tau(I_\ell)$ is obtained by turning each $z{:}\ell''{:}\tilde{m}(\bar{x}) \in \Psi_\ell$ to $z{:}\ell''{:}\hat{m}(\bar{x})$ for each $z \in \mathtt{mhf}_\alpha(\ell')$. E.g., the LMHP state of L12 in Example 7 is obtained from the one of L11 by turning the status of g, k, and f to finished (since w and x are finished at L12);
- otherwise, $\tau(I_\ell) = \Psi_\ell$.

The main difference w.r.t. the analysis of [3] is the treatment of **await** $y$?: while we use an MHF set computed using the inter-procedural MHF analysis of Sect. 4, in [3] the MHF set $\{y\}$ is used, which is obtained syntactically from the instruction. Our LMHP analysis, as [3], is defined as a solution of a set of LMHP constraints. In what follows we assume that the results of the LMHP analysis are available, and we will refer to the LMHP state of program point $\ell$ as $\Psi_\ell$.

### 5.2 Global MHP

The results of the LMHP analysis are used to construct an MHP graph, from which we can compute the desired set of MHP pairs. The construction is exactly as in [3] except that we carry the new information in the MHP atoms. However, the process of extracting the MHP pairs from such graphs will be modified.

In what follows, we use $y{:}\ell{:}\breve{m}(\bar{x})$ to refer to an MHP atom without specifying if it corresponds to an active or finished task, i.e., the symbol $\breve{m}$ can be matched to $\tilde{m}$ or $\hat{m}$. As in [3], the nodes of the MHP graph consist of two method nodes $\tilde{m}$ and $\hat{m}$ for each method $m$, and a program point node $\ell$ for each $\ell \in \texttt{ppoints}(P)$. Edges from $\tilde{m}$ to each $\ell \in \texttt{ppoints}(m)$ indicate that when $m$ is active, it can be executing at any program point, including the exit, but only one. An edge from $\hat{m}$ to $\ell_m$ indicates that when $m$ is finished it can be only at its exit program point. The out-going edges from a program point node $\ell$ reflect the atoms of the LMHP state $\Psi_\ell$ as follows: if $(y{:}\ell'{:}\breve{m}(\bar{x}), i) \in \Psi_\ell$, then there is an edge from node $\ell$ to node $\breve{m}$ and it is labeled with $i{:}y{:}\ell'{:}\bar{x}$. These edges simply indicate which tasks might be executing in parallel when reaching $\ell$, exactly as $\Psi_\ell$ does.

*Example 8.* The MHP graphs $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ in Fig. 2, correspond to methods $\mathsf{m}_1$, $\mathsf{m}_2$, and $\mathsf{m}_3$, each analyzed together with its reachable methods. For simplicity, the graphs include only some program points of interest. Note that the out-going edges of program point nodes coincide with the LMHP states of Example 7.

The procedure of [3] for extracting the MHP pairs from the MHP graph of a program $P$, denoted $\mathcal{G}_P$, is based on the following principle: $(\ell_1, \ell_2)$ is an MHP pair induced by $\mathcal{G}_P$ iff (i) $\ell_1 \rightsquigarrow \ell_2 \in \mathcal{G}_P$ or $\ell_2 \rightsquigarrow \ell_1 \in \mathcal{G}_P$; or (ii) there is a program point node $\ell_3$ and paths $\ell_3 \rightsquigarrow \ell_1 \in \mathcal{G}_P$ and $\ell_3 \rightsquigarrow \ell_2 \in \mathcal{G}_P$, such that the first edges of these paths are different and they do not correspond to mutually exclusive MHP atoms, i.e., they use different future variables and do not correspond to mutually exclusive calling sites (see Sect. 5.1). Edges with multiplicity $i > 1$ represent $i$ different edges. The first (resp. second) case is called direct (resp. indirect) MHP, see Sect. 3.

*Example 9.* Let us explain some of the MHP pairs induced by $\mathcal{G}_1$ of Fig. 2. Since $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 58 \in \mathcal{G}_1$, we conclude that (11,58) and (11,35) are direct MHP pairs. Moreover, since these paths originate in the same node 11, and the first edges use different future variables, we conclude that (58,35) is an indirect MHP pair. Similarly, since $11 \rightsquigarrow 38 \in \mathcal{G}_1$ and $11 \rightsquigarrow 50 \in \mathcal{G}_1$ we conclude that (11,38) and (11,50) are direct MHP pairs. However, in this case (38,50) is not an (indirect) MHP pair because the first edges of these paths use the same future variable $\mathsf{w}$. Indeed, the calls to $\mathsf{g}$ and $\mathsf{k}$ appear in different branches of an **if** statement. To see the improvement w.r.t. to [3] note that node 12 does not have an edge to $\tilde{\mathsf{f}}$, since our MHF analysis infers that $\mathsf{x}$ is finished at that L12. The analysis of [3] would have an edge to $\tilde{\mathsf{f}}$ instead of $\hat{\mathsf{f}}$, and thus it produces spurious pairs such as (12,35). Similar improvements occur also in $\mathcal{G}_2$ and $\mathcal{G}_3$.

Now consider nodes 35 and 40, and note that we have $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 40 \in \mathcal{G}_1$, and moreover these paths use different future variables. Thus,

we conclude that (35,40) is an indirect MHP pair. However, carefully looking at the program we can see that this is a spurious pair, because x (to which task f is bound) is passed to method g, as parameter w, and w is guaranteed to finish when executing **await** w? at L39. A similar behavior occurs also in $\mathcal{G}_2$ and $\mathcal{G}_3$. For example, the paths $30 \rightsquigarrow 58 \in \mathcal{G}_3$ and $30 \rightsquigarrow 40 \in \mathcal{G}_3$ induce the indirect MHP pair (58,40), which is spurious since x is passed to h at L45, as parameter a, which in turn is passed to g at L45, as parameter w, and w is guaranteed to finish when executing **await** w? at L39.

The spurious pairs in the above example show that even if we used our improved LMHP analysis when constructing the MHP graph, using the procedure of [3] to extract MHP pairs might produce spurious pairs. Next, we address this imprecision by modifying the process of extracting the MHP pairs to have an extra condition to eliminate such spurious MHP pairs. This condition is based on identifying, for a given path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, which of the parameters of $m$ are guaranteed to finish before reaching $\ell$, and thus, any task that is passed to $m$ in those parameters cannot execute in parallel with $\ell$.

**Definition 2.** *Let $p$ be a path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, $\bar{z}$ be the formal parameter of $m$, and $I$ a set of parameter indices of method $m$. We say that $I$ is* not alive along *$p$ if (i) $p$ has a single edge, and for some $i \in I$ the parameter $z_i$ is in $\texttt{mhf}_\alpha(\ell)$; or (ii) $p$ is of the form $\breve{m} \longrightarrow \ell_1 \overset{k:y:\ell':\bar{x}}{\longrightarrow} \breve{m}_1 \rightsquigarrow \ell$, and for some $i \in I$ the parameter $z_i$ is in $\texttt{mhf}_\alpha(\ell_1)$ or $I' = \{j \mid i \in I, z_i = x_j\}$ is not alive along $\breve{m}_1 \rightsquigarrow \ell$.*

Intuitively, $I$ is not alive along $p$ if some parameter $z_i$, with $i \in I$, is finished at some point in $p$. Thus, any task bound to $z_i$ cannot execute in parallel with $\ell$.

*Example 10.* Consider $p \equiv \tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, and let $I = \{1\}$, then $I$ is not alive along $p$ since it is a path that consists of a single edge and $\texttt{w} \in \texttt{mhf}_\alpha(40)$. Now consider $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, and let $I = \{1\}$, then $I$ is not alive along $p$ since $I' = \{1\}$ is not alive along $\tilde{g} \rightsquigarrow 40$.

The notion of "*not alive along a path*" can be used to eliminate spurious MHP pairs as follows. Consider two paths

$$p_1 \equiv \ell_3 \overset{i_1:y_1:\ell_1':\bar{w}}{\longrightarrow} \tilde{m}_1 \rightsquigarrow \ell_1 \in \mathcal{G}_P \quad \text{and} \quad p_2 \equiv \ell_3 \overset{i_2:y_2:\ell_2':\bar{x}}{\longrightarrow} \breve{m}_2 \rightsquigarrow \ell_2 \in \mathcal{G}_P$$

such that $y_1 \neq \star$, and the first node after $\tilde{m}_1$ does not correspond to the exit program point of $m_1$, i.e., $m_1$ might be executing and bound to $y_1$. Define

- $F = \{y_1\} \cup \{y \mid \Phi_{\ell_3} \models y \rightarrow y_1\}$, i.e., the set of future variables at $\ell_3$ such that when any of them is finished, $y_1$ is finished as well; and
- $I = \{i \mid y \in F,\ x_i = y\}$, i.e., the indices of the parameters of $m_2$ to which we pass variables from $F$ (in $p_2$).

We claim that if $I$ is not alive along $p_2$, then the MHP pair $(\ell_1, \ell_2)$ is spurious. This is because before reaching $\ell_2$, some task from $F$ is guaranteed to terminate, and hence the one bound to $y_1$, which contradicts the assumption that $m_1$ is not finished. In such case $p_1$ and $p_2$ are called mutually exclusive paths.

*Example 11.* We reconsider the spurious indirect MHP pairs of Example 9. Consider first (35,40), which originates from

$$p_1 \equiv 11 \overset{1:\mathsf{x}:2:[]}{\longrightarrow} \tilde{f} \rightsquigarrow 35 \in \mathcal{G}_1 \text{ and } p_2 \equiv 11 \overset{1:\mathsf{w}:6:[\mathsf{x}]}{\longrightarrow} \tilde{g} \rightsquigarrow 40.$$

We have $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Example 10 that $I$ is not alive along $\tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, thus $p_1$ and $p_2$ are mutually exclusive and we eliminate this pair. Similarly, consider (58,40) which originates from

$$p_1 \equiv 30 \overset{1:\mathsf{x}:28:[]}{\longrightarrow} \tilde{q} \rightsquigarrow 58 \in \mathcal{G}_3 \text{ and } p_2 \equiv 30 \overset{1:\mathsf{w}:29:[\mathsf{x},\mathsf{z}]}{\longrightarrow} \tilde{h} \rightsquigarrow 40.$$

Again $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Example 10 that $I$ is not alive along $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, thus $p_1$ and $p_2$ are mutually exclusive and we eliminate this pair.

Recall that $\mathcal{E}_P$ is the set of all concrete MHP pairs. Let $\tilde{\mathcal{E}}_P$ be the set of all MHP pairs obtained by applying the process of [3], modified to eliminate indirect pairs that correspond to mutually exclusive paths.

**Theorem 2.** $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

## 6 Conclusions, Implementation and Related Work

The main contribution of this work has been the enhancement of an MHP analysis that could only handle a restricted form of intra-procedural synchronization to the more general inter-procedural setting, as available in today's concurrent languages. Our analysis has a wide application scope on the inference of the main properties of concurrent programs, namely the new MHP relations are essential to infer (among others) the properties of the termination, resource usage and deadlock freedom of programs that use inter-procedural synchronization.

The analysis has been implemented in SACO [2], a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination and resource boundedness of ABS programs [10] that follow the concurrent objects paradigm. Concurrent objects are based on the notion of concurrently running objects, similar to the actor-based and active-objects approaches [12,13]. These models take advantage of the concurrency implicit in the notion of object to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. Concurrent objects communicate via *asynchronous* method calls and use **await** instructions to synchronize with the termination of the asynchronous tasks. Therefore, the abstract model used in Sect. 2 fully captures the MHP relations arising in ABS programs.

The implementation has been built on top of the original MHP analysis in SACO. The MHF analysis has been implemented and its output has been used within the local and global phases of the MHP analysis, which have been adapted

to this new input as described in the technical sections. The remaining analyses in SACO did not require any modification and now they work for inter-procedural synchronization as well. Our method can be tried online at: http://costa.ls.fi.upm.es/saco/web by enabling the option `Inter-Procedural Synchronization` of the `MHP` analysis in the `Settings` section. One can then apply the MHP analysis by selecting it from the menu for the types of analyses and then clicking on `Apply`. All examples used in the paper are available in the folder `SAS15` adapted to the syntax of the ABS language. In the near future, we plan to apply our analysis to industrial case studies that are being developed in ABS but that are not ready for experimentation yet.

There is an increasing interest in asynchronous programming and in concurrent objects, and in the development of program analyses that reason on safety and liveness properties [6]. Existing MHP analyses for asynchronous programs [1,3,11] lose all information when future variables are used as parameters, as they do not handle inter-procedural synchronization. As a consequence, existing analysis for more advanced properties [4,9] that rely on the MHP relations do all lose the associated analysis information on such futures. In future work we plan to study the complexity of our analysis, which we conjuncture to be in the same complexity order as [3]. In addition, we plan to study the computational complexity of deciding MHP, for our abstract models, with and without inter-procedural synchronizations in a similar way to what has been done in [5] for the problem of state reachability.

## References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of X10 programs. In: Yelick, K.A., Mellor-Crummey, J.M. (ed.), Proceedings of PPOPP 2007, pp. 183–193. ACM (2007)
2. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014)
3. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of may-happen-in-parallel in concurrent objects. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012)
4. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: Termination and cost analysis of loops with concurrent interleavings. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 349–364. Springer, Heidelberg (2013)
5. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. ACM Trans. Program. Lang. Syst. **35**(3), 10 (2013)
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Proceedings of POPL 2015, pp. 651–662. ACM (2015)
7. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
8. Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1995)

9. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
11. Lee, J.K., Palsberg, J., Majumdar, R., Hong, H.: Efficient may happen in parallel analysis for async-finish parallelism. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 5–23. Springer, Heidelberg (2012)
12. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
13. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)

## Proofs

In this we provide a sketch of a proof for Theorem 1. Note that a proof for Theorem 2 is not given since it is basically as the one of [19] but (1) using the MHF information, for the case of **await** y? in the local analysis, which is straightforward; and (2) using the *mutually exclusive* path condition whose correctness was intuitively argued in Section 5.1.

### Sketch of a proof for Theorem 1

The proof sketch follows the next general lines:

1. We first define a concrete *collecting semantics* that basically collects all reachable states when starting from some initial state;
2. We reformulate the MHF analysis (equivalently) as an abstract collecting semantics, which is easier to relate to the concrete one; and
3. We show that the abstract collecting semantics correctly approximates the concrete one (with respect to the MHF property).

Next we define the concrete collecting semantics. Recall that a program state $S$ is a set of tasks (see Section 2). A concrete state in the collecting semantics, or simply *concrete state* or *state*, is a set of program states. We let $\mathcal{C}_0 = \{S_0\}$ where $S_0 = \{tsk(0, l, body(\mathsf{main}))\}$, i.e., an initial state that includes a single program state $S_0$ with a single task corresponding to method main. Let the function $T$ be defined as follows

$$T = \lambda\mathcal{C}.\ \mathcal{C}_0 \cup \{S' \mid S \in \mathcal{C}, S \rightsquigarrow S'\}.$$

Then, the concrete collecting semantics is defined by

$$\mathcal{X} = lfp\, T.$$

Intuitively, $\mathcal{X}$ is the set of all reachable states when starting the execution from the initial state $S_0$.

Next we revisit our MHF analysis and reformulate it as an abstract collecting semantics. An abstract state $\Phi$ is a mapping from $\texttt{ppoints}(P)$ to Boolean formulas (as those defined in Section 4.2). The value to which $\ell$ is mapped in $\Phi$ is denoted by $\Phi_\ell$. Recall that our analysis is based on generating and solving a set of data-flow equations $\mathcal{H}_P$. Given an abstract state $\Phi$, we let $\mathcal{H}_P(\Phi)$ be the result of substituting the different formulas of $\Phi$ in the right-hand-side of $\mathcal{H}_P$, and in this way obtain new values for each $\Phi_\ell$. Given two abstract states $\Phi$ and $\Phi'$, we let $\Phi'' = \Phi \vee \Phi'$ be an abstract state such that $\Phi''_\ell = \Phi_\ell \vee \Phi'_\ell$ for any $\ell \in \texttt{ppoints}(P)$. We let $\Phi_0$ be an initial abstract state such that $\Phi_\ell = \mathit{false}$ for each $\ell \in \texttt{ppoints}(P)$. Let the function $\bar{T}$ be defined as follows

$$\bar{T} = \lambda\Phi.\ \Phi_0 \vee \mathcal{H}_P(\Phi).$$

Then, the abstract collecting semantics, which is the result of our analysis, is defined by

$$\bar{\mathcal{X}} = lfp\, \bar{T}.$$

Next we define when an abstract state $\Phi$ correctly approximates a concrete one $\mathcal{C}$. First we recall the definition $f(S,l) = \{x \mid x \in dom(l),\ l(x) = \bot \vee (l(x) = tk' \wedge tsk(tk', l', \epsilon) \in S)\}$ from Section 4.1, which is used to obtain the set of finished tasks, in a state $S$, for the future variables defined in $l$.

**Definition 1.** *An abstract state $\Phi$ correctly approximates a concrete state $\mathcal{C}$, denoted $\Phi \approx \mathcal{C}$, if for any $tsk(tk, l, s) \in S \in \mathcal{C}$, where $pp(s) = \ell$, it holds that $f(S,l) \in [\![\Phi_\ell]\!]$.*

The rest of this section shows that the abstract collecting semantics correctly approximates the concrete one, namely $\bar{\mathcal{X}} \approx \mathcal{X}$.

**Lemma 1.** $\forall n \geq 1.\ \exists k \geq n.\ \bar{T}^k(\Phi_0) \approx T^n(\emptyset)$.

*Proof.* The proof of the above Lemma is by induction on $n$.

For $n = 1$, we take $k = 1$, then $\mathcal{C}_0 = T(\emptyset)$ and $\Phi = \bar{T}(\Phi_0)$ is a state in which $\Phi_\ell = \mathit{false}$ for all $\ell \in \texttt{ppoints}(P)$ except for the entry point of main which is mapped to $\wedge\{x \mid x \in L_{\mathsf{main}}\}$. Clearly, $\Phi \approx \mathcal{C}_0$.

For $n > 1$, let $\mathcal{C} = T^{n-1}(\emptyset)$ and $\mathcal{C}' = T^n(\emptyset)$. Note that $\mathcal{C}' = T(\mathcal{C})$. By the induction hypothesis, for there is $k \geq n-1$ such that $\Phi = \bar{T}^k(\Phi_0)$ correctly approximates $\mathcal{C}$. Let $S' \in \mathcal{C}'$ but $S' \notin \mathcal{C}$, and note that it must have been generated using some $S \in \mathcal{C}$ in one execution step, i.e., $S \rightsquigarrow S'$. In particular, by one execution step of a task $t = tsk(tk, l, b; s) \in S$, i.e., $b$ was executed. Next we reason on all possible cases of $b$. First let us assume that $s \neq \epsilon$, i.e., this execution step does not introduce any new finished task, and later we come back to the case in which $s = \epsilon$. Assume that $b$ corresponds to program point $\ell$ and that the first instruction in $s$ corresponds to program point $\ell'$, i.e., $\ell \in \texttt{pre}(\ell')$.

**Case 1:** $b \equiv$ **skip.** In this case, the execution rewrites task $t$ into $t' = tsk(tk, l, s)$ as well. Also in this case the status of each task $l(y)$, for any $y \in dom(l)$, in $S'$ is the same as in $S$, since no finished task was introduced, i.e., $f(S', l) = f(S, l)$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$, we claim that $f(S', l) \in [\![\Phi'_{\ell'}]\!]$. This is obvious since $f(S, l) \in [\![\Phi_\ell]\!]$ and $\Phi'_{\ell'}$ was obtained from an equation whose right-hand-side is a disjunction that includes $\mu(\Phi_\ell) = \Phi_\ell$.

**Case 2:** $b \equiv$ **await** $y?$**.** In this case, the execution rewrites task $t$ into $t' = tsk(tk, l, s)$. Note that the status of each task $l(y)$, for any $y \in dom(l)$, in $S'$ is the same as in $S$, since no finished task was introduced ($l(y)$ must be finished in $S$ to be able to execute this instruction), i.e., $f(S', l) = f(S, l)$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$, we claim that $f(S', l) \in [\![\Phi'_{\ell'}]\!]$. This is obvious since $y \in f(S', l) = f(S, l) \in [\![\Phi_\ell]\!]$ and $\Phi'_{\ell'}$ was obtained from an equation whose right-hand-side is a disjunction that includes $\mu(\Phi_\ell) = \Phi_\ell \wedge y$. Adding $y$ to $\Phi_\ell$ does not eliminate models that include $y$, in particular $f(S', l)$.

**Case 3:** $b \equiv y = m(\bar{x})$**.** In this case, the execution rewrites task $t$ into $t' = tsk(tk, l', s)$, and adds a new task $t'' = tsk(tk', l'', s')$ such that $l'(y) = tk'$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$. From the same consideration as above it is easy to see that $f(S', l') \in [\![\Phi'_{\ell'}]\!]$ because the corresponding equation eliminates the old value of $y$ from $\Phi_\ell$, and adds $y \rightarrow \Phi_m$. This implication has no effect when $y$ is false. Let $\ell''$ be the first program point of $s'$, clearly $f(S', l'') \in [\![\Phi'_{\ell''}]\!]$ because of the way we generate the equation for entry program points (all local variables points to finished tasks, and parameters can be finished or not finished).

Now we go back to comment on the case that $s = \epsilon$. I.e., $b$ was the last instruction executed in task $t$. The problem here is that any other task that has a reference to $t'$ must now consider the possibility that this task as finished. If we start from $\Phi' = \bar{T}^{k+1}(\Phi_0)$, and apply $\bar{T}$ one more time, we will reconsider the place where that method was called and use the new method summary $\Phi'_m$, i.e., the method call will now add $y \to \Phi'_m$. This can be repeated a finite number of times until the information is propagated to all corresponding program points. □

**Corollary 1.** $\bar{\mathcal{X}} \approx \mathcal{X}$.

*Proof.* Immediate from Lemma 1 and Kleene fixed-point theorem. □

# May-Happen-in-Parallel Analysis with Returned Futures

Elvira Albert( ), Samir Genaim, and Pablo Gordillo

Complutense University of Madrid (UCM), Madrid, Spain
elvira@sip.ucm.es

**Abstract.** May-Happen-in-Parallel (MHP) is a fundamental analysis to reason about concurrent programs. It infers the pairs of program points that may execute in parallel, or interleave their execution. This information is essential to prove, among other things, absence of data races, deadlock freeness, termination, and resource usage. This paper presents an MHP analysis for asynchronous programs that use *futures* as synchronization mechanism. Future variables are available in most concurrent languages (e.g., in the library `concurrent` of Java, in the standard thread library of C++, and in Scala and Python). The novelty of our analysis is that it is able to infer MHP relations that involve future variables that are *returned* by asynchronous tasks. Futures are returned when a task needs to await for another task created in an *inner* scope, e.g., task $t$ needs to await for the termination of task $p$ that is spawned by task $q$ that is spawned during the execution of $t$ (not necessarily by $t$). Thus, task $p$ is awaited by task $t$ which is in an outer scope. The challenge for the analysis is to (back)propagate the synchronization of tasks through future variables from inner to outer scopes.

## 1 Introduction

MHP is an analysis of utmost importance to ensure both liveness and safety properties of concurrent programs. The analysis computes *MHP pairs*, which are pairs of program points whose execution might happen, in an (concurrent) interleaved way within one processor, or in parallel across different processors. This information is fundamental to prove absence of data races as well as more complex properties: In [13], MHP pairs are used to discard unfeasible deadlock cycles; namely if a deadlock cycle inferred by the deadlock analyzer includes pairs of program points that are proven not to happen in parallel by our MHP analysis, the cycle is spurious and the program is deadlock free. In [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. For instance, consider a loop whose termination cannot be proven because of a potential execution in parallel of the loop with a

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, by the CM project S2013/ICE-3006 and by the UCM CT27/16-CT28/16 grant.


D. D'Souza and K. Narayan Kumar (Eds.): ATVA 2017, LNCS 10482, pp. 42–58, 2017.
DOI: 10.1007/978-3-319-68167-2_3

task that modifies the variables that control the loop guard (and thus threatens its termination). If our MHP analysis proves the unfeasibility of such parallelism, then termination of the loop can be guaranteed.

For simplicity, we develop our analysis on a small asynchronous language which uses *future variables* [10,12] for task synchronization. A method call `m` on some parameters $\bar{x}$, written as `f` = `m`($\bar{x}$), spawns an asynchronous task, and the future variable `f` allows synchronizing with the termination of such task by means of the instruction **await** `f?`; which delays the execution until the asynchronous task has finished. In this fragment of code `f` = `m`(..) ;...; **await** `f?`; the execution of the instructions of the asynchronous task `m` may happen in parallel with the instructions between the asynchronous call and the `await`. However, due to the future variable in the `await` instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the `await`. Therefore, future variables play an essential role within an MHP analysis and it is essential for its precision to track them accurately. Future variables are available in most concurrent languages: Java, Scala and Python allow creating pools of threads. The users can submit tasks to the pool, which are executed when a thread of the pool is idle, and may return future variables to synchronize with the tasks termination. C++ includes the components async, future and promise in its standard library, which allow programmers to create tasks (instead of threads) and return future variables in the same way as we do.

In this paper, we present to the best of our knowledge the first MHP analysis that captures MHP relations that involve tasks that are awaited in an outer scope from the scope in which they were created. This happens when future variables are returned by the asynchronous tasks, as it can be performed in all programming languages that have future variables. Our analysis builds on top of an existing MHP analysis [3] that was extended to track information of future variables passed through method parameters in [5], but it is not able to track information propagated through future variables that are returned by tasks. The original MHP analysis [3] involves two phases: (1) a local analysis which consists in analyzing the instructions of the individual tasks to detect the tasks that it spawns and awaits, and (2) a global analysis which propagates the local information compositionally. Accurately handling returned future variables requires non-trivial extensions in both phases:

1. The local phase needs to be modified to backpropagate the additional inter-procedural relations that arise from the returned futures variables. Back-propagation is achieved by modifying the data-flow of the analysis so that it iterates to propagate the new dependencies.
2. The global phase has to be modified by reflecting in the analysis graph the additional information provided by the local phase. A main achievement has been to generate the necessary information at the local phase so that the process of inferring the MHP pairs remains as in the original analysis.

Our analysis has been implemented within the SACO static analyzer [2], which is able to infer the safety and liveness properties mentioned above. The system can be used online at http://costa.ls.fi.upm.es/saco/web/, where the benchmarks

used in the paper are available. Our experiments show that our analysis improves the accuracy over the previous analysis with basically no overhead.

## 2 Language

We present the syntax and semantics of the asynchronous language on which we develop our analysis. A program $P$ is composed by a set of classes. Each class contains a set of fields and a set of methods. A (concurrent) object of a class represents a processor with a queue of tasks which (concurrently) execute the class methods, and access a shared-memory made up by the object fields. One of the tasks will be active (executing) and the others pending to be executed. The notation $\overline{M}$ is used to abbreviate $M_1, ..., M_n$. Each field and method has a type $T$. The set of types includes class identifiers $C$ and future variable types $fut\langle T\rangle$. A method receives a set of variables as arguments $\bar{x}$, contains local variables $\bar{x}'$, a returned variable, and a sequence of instructions $s$.

$$
\begin{array}{lll}
CL & ::= & class\ C\ \{\overline{T}\ \overline{f}; \overline{M}\} \\
M & ::= & T\ m(\overline{T}\ \bar{x})\ \{\overline{T}\ \bar{x}'; s\} \\
s & ::= & \epsilon \mid b; s \\
b & ::= & o = \textbf{new}\ C(\bar{x}) \mid \textbf{if}\ (*)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{while}\ (*)\ \textbf{do}\ s \mid y = o.m(\bar{x}) \mid \\
 & & \mid \textbf{await}\ y? \mid z = y.\textbf{get} \mid \textbf{return}\ y \mid \textbf{skip}
\end{array}
$$

$y$ and $z$ represent variables of type $fut\langle T\rangle$ and $x$ represents a variable of type $T$. Arithmetic expressions are omitted for simplicity and are represented by the instruction **skip**. This instruction has no effect on the analysis of the program. The loop and conditional statements are non-deterministic and the symbol $*$ represents *true* or *false*. The instruction $y = o.m(\bar{x})$ corresponds to an asynchronous call. It spawns a new instance of the task $m$ in the object $o$ and binds the task to the future variable $y$. Instruction **await** $y?$ is used to synchronize with the task $y = o.m(\bar{x})$, and blocks the execution in object $o$ until task $m$ finishes its execution. $z = y.\textbf{get}$ retrieves the value returned by the method bound to $y$ and associates it with $z$. W.l.o.g., we make the following assumptions: each **get** instruction is preceded by an **await**, i.e., the task associated to the **get** statement has to be finished to access its returned value; the program has a method call `main` without parameters from which the execution will start; future variables can be used once and they cannot be reused after they are bound to a task; the **get** instruction can be applied once over each future variable; we restrict the values returned by a method to future variables; each method can only have a **return** statement in its body, and it has to be the last instruction of the sequence. We let $\texttt{ppoints}(m)$ and $\texttt{ppoints}(P)$ be the set of program points of method $m$ and program $P$ respectively, $\texttt{methods}(P)$ be the set of method names of program $P$ and $\texttt{futures}(P)$ be the set of all future variables defined in program $P$.

Let us define the operational semantics for the language. A *program state* $S$ is a tuple $S = \langle O, T\rangle$ where $O$ is the set of objects and $T$ is the set of tasks. Only one task can be active in each object. An *object* is a term $obj(o, a, lk)$ where $o$ is

$$(1)\ \frac{l' = l[o \to bid_1], O' = O \cup \{obj(bid_1, a, \bot)\}, a = init\_atts(C, \overline{x}), bid_1 \text{ is a fresh id}}{\langle O, \{tsk(tid, m, l, bid, \top, o = \textbf{new } C(\overline{x}); s) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l', bid, \top, s) \parallel T\}\rangle}$$

$$(2)\ \frac{l(o) = bid_1 \neq \textbf{null}, l' = l[y \to tid_1], l_1 = buildLocals(\overline{x}, m)), tid_1 \text{is a fresh id}}{\begin{array}{c}\langle O, \{tsk(tid, m, l, bid, \top, y{=}o.m_1(\overline{x}); s) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid, m, l', bid, \top, s), tsk(tid_1, m_1, l_1, bid_1, \bot, body(m_1)) \parallel T\}\rangle\end{array}}$$

$$(3)\ \frac{l_1(y) = tid_2}{\begin{array}{c}\langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, \textbf{await } y?; s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle\end{array}}$$

$$(4)\ \frac{l_1(y) = tid_2, l_1' = l_1[z \to v]}{\begin{array}{c}\langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, z{=}y.\textbf{get}; s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid_1, m_1, l_1', bid_1, \top, s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle\end{array}}$$

$$(5)\ \frac{obj(bid, a, \top) \in O, O' = O[obj(bid, a, \top)/obj(bid, a, \bot)], v = l(y)}{\langle O, \{tsk(tid, m, l, bid, \top, \textbf{return } y) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l, bid, \bot, \epsilon(v)) \parallel T\}\rangle}$$

$$(6)\ \frac{\begin{array}{c}(l', s') = eval(instr, O, l) \\ instr \in \{\textbf{skip}, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2, \textbf{while } b \textbf{ do } s_3\}\end{array}}{\langle O, \{tsk(tid, m, l, bid, \top, instr; s) \parallel T\}\rangle \rightsquigarrow \langle O, \{tsk(tid, m, l', bid, \top, s') \parallel T\}\rangle}$$

$$(7)\ \frac{obj(bid, a, \bot) \in O, O' = O[obj(bid, a, \bot)/obj(bid, a, \top)], s \neq \epsilon(v)}{\langle O, \{tsk(tid, m, l, bid, \bot, s) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l, bid, \top, s) \parallel T\}\rangle}$$

**Fig. 1.** Summarized semantics

the identifier of the object, $a$ is a mapping from the object fields to their values and $lk \in \{\top, \bot\}$ indicates whether the object contains an active task executing ($\top$) or not ($\bot$). A task is a term $tsk(t, m, l, o, lk, s)$ where $t$ is a unique task identifier, $m$ is the method name that is being executed, $l$ is a mapping from the variables of the task to their values, $o$ is the identifier of the object in which the task is executing, $lk \in \{\top, \bot\}$ indicates if the task has the object's lock or not and $s$ is a sequence of instructions that the task will execute or $s = \epsilon(v)$ if the task has finished and the return value $v$ is available. The execution of a program starts from the initial state $S_0 = \langle obj(0, a, \top), tsk(0, \mathsf{main}, l, 0, \top, body(main)\rangle$ where $a$ is an empty mapping, and $l$ maps future variables to **null**.

The execution starts from $S_0$ applying *non-deterministically* the semantic rules from Fig. 1. We use the notation $\{t \parallel T\}$ to represent that task $t$ is the one selected non-deterministically for the execution. At each step, a subset of the state $S$ is rewritten according to the rules of Fig. 1 as follows: (1) creates a new object with an empty queue, free lock and initializes its fields (*init_atts*). (2) corresponds to an asynchronous call. It gets the identifier of the object which is going to execute the task, initializes the parameters and variables of the task (*buildLocals*), and creates the new task with a new identifier that is associated with the corresponding future variable. (3) An **await** $y$? statement waits until the

task bound to $y$ finishes its execution. (4) checks if the task bound to the future variable involved in the **get** statement is finished. If so, it retrieves the value associated with the future variable. (5) After executing the **return** statement, the retrieved value is stored in $v$ so that it can be obtained by the future variable bound to this task. Then, the object's lock is released ($O[o/o']$ means that the object $o$ is replaced by $o'$ in $O$) and the task is finished ($\epsilon(v)$ is added to the sequence of instructions). (6) covers sequential instructions that do not affect synchronization by moving the execution of the corresponding task to the next instruction and possibly changing the state (represented by $eval$). Finally, (7) is used to get the object's lock by an unfinished task and start its execution.

In what follows, given a task $tsk(t, m, l, o, lk, s)$, $pp(s)$ denotes the program point of the first instruction of $s$. If $s$ is empty, $pp(s)$ returns the exit program point of the corresponding method, denoted $exit(m)$. Given a state $S = \langle O, T\rangle$, we define its set of MHP pairs, i.e., the set of program points that can run in parallel as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, m_1, l_1, o_1, lk_1, s_1), tsk(tid_2, m_2, l_2, o_2, lk_2, s_2) \in T, tid_1 \neq tid_2\}$. The set of MHP pairs for a program $P$ is defined as the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \rightsquigarrow^* S_n\}$.

## 3 Motivation: Using MHP Pairs in Deadlock Analysis

Let us motivate our work by showing its application in the context of deadlock analysis. Consider the example in Fig. 2 that models a typical client-server application with two delegate entities to handle the requests. The execution starts from the main block by creating four concurrent objects, the client c, the server s, and their delegates dc and ds, respectively. The call start at Line 6 (L6) spawns an asynchronous task on the client object c that sends as arguments references to the other objects. When this task is scheduled for execution on the client, we can observe that it will spawn an asynchronous task on the server (L10) and another one on the delegate-client (L14). The request task on the server in turn posts two asynchronous tasks on the delegate-server (L19) and delegate-client objects (L20). Such delegates communicate directly with each other as we have passed as arguments the references to them.

The most challenging aspect for the analysis of this model is due to the synchronization through returned future variables. For instance at L12 the instruction x.**get** retrieves the future variable returned by request at L21. Thus, we would like to infer that after L13 the task executing result at the object ds has terminated. The inference needs to backpropagate this synchronization information from the inner scope where the task has been created (L19) to the outer scope where it is awaited (L13). This backpropagation is necessary in order to prove that the execution of this application is deadlock free. Otherwise, an MHP-based deadlock analyzer will spot an unfeasible deadlock. Figure 3 shows a fragment of the graph that a deadlock analyzer [13] constructs: the concurrent objects are in circles, the asynchronous tasks in boxes, and labelled arrows contain the program lines at which tasks post new tasks on the destiny objects. In the bold arrows of the graph, we can observe the cycle detected by the analyzer due to the task

```
1  main() {
2    Client c = new Client();
3    Server s = new Server();
4    DS ds = new DS();
5    DC dc = new DC();
6    c.start(s,ds,dc);
7  }
8  class Client {
9   Unit start(Server s, DS ds, DC dc){
10     x=s.request(ds,dc);
11     await x?;
12     z=x.get;
13     await z?;
14     dc.sendMessage(ds);
15   }
16 }
17 class Server {
18   Fut<Unit> request(DS ds, DC dc){
19     y=ds.result(dc);
20     p = dc.inform();
21     return y;
22   }
23 }
25 class DS {
26   Unit result(DC dc){
27     w = dc.myClientId();
28     await w?;
29   }
30   Unit myServerId() {
31     skip;
32   }
33 }
34 class DC {
35   Unit sendMessage(DS ds){
36     r = ds.myServerId();
37     await r?;
38   }
39   Unit myClientId() {
40     skip;
41   }
42   Unit inform() {
43     skip;
44   }
45 }
```

**Fig. 2.** Example of client-server model

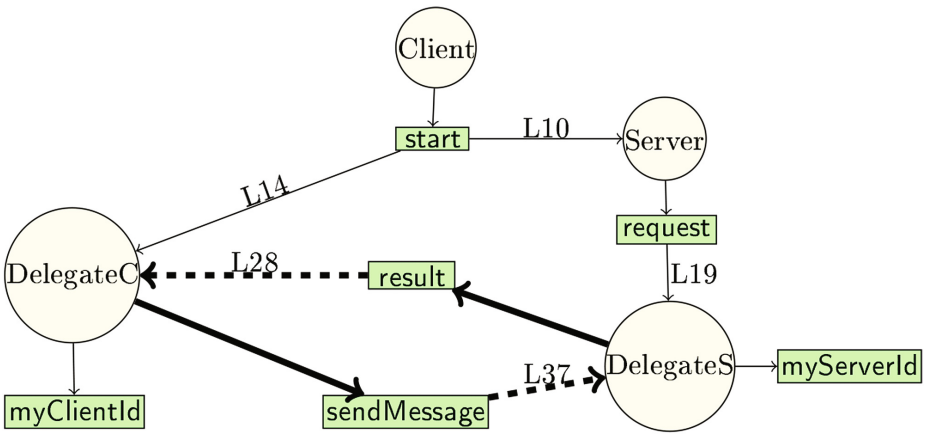



**Fig. 3.** Partial data-flow graph of example in Fig. 2.

result and sendMessage executing respectively in objects ds and dc. These two tasks wait for the termination of tasks myClientId and myServerId in each other object, thus creating a potential cycle. Our MHP analysis will accurately infer that these two tasks cannot happen simultaneously, and will allow the deadlock analyzer to break this unfeasible deadlock cycle. Figure 4 shows some of the MHP pairs that the analysis in [3] infers, and we mark in bold font those pairs

| | | | | | | |
|---|---|---|---|---|---|---|
| $L11\|\|L19$ | $L11\|\|L20$ | $L11\|\|L21$ | $L11\|\|L28$ | $L11\|\|L29$ | $L11\|\|L40$ | $L12\|\|L22$ |
| $L12\|\|L28$ | $L12\|\|L29$ | $L12\|\|L40$ | $L14\|\|L22$ | **L14\|\|L28** | $L14\|\|L29$ | **L14\|\|L40** |
| $L15\|\|L22$ | **L15\|\|L28** | $L15\|\|L29$ | **L15\|\|L40** | $L15\|\|L31$ | $L21\|\|L29$ | $L21\|\|L40$ |
| $L22\|\|L28$ | $L22\|\|L44$ | $L28\|\|L41$ | $L28\|\|L43$ | $L28\|\|L44$ | $L29\|\|L41$ | $L29\|\|L44$ |
| $L29\|\|L41$ | $L29\|\|L44$ | **L28\|\|L37** | **L28\|\|L38** | $L29\|\|L40$ | $L21\|\|L44$ | |

**Fig. 4.** Results of MHP analysis.

that our analysis spots as spurious (as will be explained along the paper). For instance, the original analysis infers L15||L28 and L37||L28. However, we detect that at program point L14 the task result is finished so it cannot run in parallel with task sendMessage and hence those pairs are eliminated, this allows us later to discard the potential deadlock described above.

## 4 MHP Analysis

The MHP analysis of [3] consists of two phases. The first one, the local phase, considers each method separately and infers information (at each program point of the method) about the status of the tasks that are created locally in that method. The second one, the global phase, uses the information inferred by the first phase to construct an MHP graph from which an over-approximation of the MHP pairs set can be extracted. As mentioned already, the limitation of this analysis is that it does not track inter-procedural synchronizations originating from (1) passing future variables as method parameters; or (2) returning future variables from one method to another. The work of [5] extends [3] to handle the first issue, and in this paper we extend it to handle the second one. Both extensions require different techniques, and are both complementary and compatible. To simplify the presentation, we have not started from the analysis with future variables as parameters [5], but rather from the original formulation [3]. In Sect. 6, we provide a detailed comparison of [5] and our current extension.

### 4.1 Local MHP

The local phase of the MHP analysis (LMHP) of [3] considers each method $n$ separately, and for each program point $\ell \in \texttt{ppoints}(n)$ it infers a LMHP state that describes the status of each task invoked in $n$ before reaching $\ell$. Formally, a LMHP state $E$ is a *multiset* of MHP atoms, where an MHP atom is:

1. $y{:}\texttt{T}(m, \texttt{act})$, which represents a task that is an instance of method $m$ and can be executing at any program point. We refer to it as *active* task; and
2. $y{:}\texttt{T}(m, \texttt{fin})$, which represents a task that is an instance of method $m$ and has finished its execution already (i.e., it is at its exit program point). We refer to it as *finished* task.

$$
\begin{array}{ll}
(1) & \tau(y = o.m(\bar{x}), E) = E[y{:}\mathtt{T}(m, X)/\star{:}\mathtt{T}(m, X)] \cup \{y{:}\mathtt{T}(m, \mathtt{act})\} \\
(2) & \tau(\textbf{await } y?, E) = E[y{:}\mathtt{T}(m, \mathtt{act})/y{:}\mathtt{T}(m, \mathtt{fin})] \\
(3) & \tau(z = y.\textbf{get}, E) = E' \cup E'' \cup E''' \quad \text{where:}
\end{array}
$$

$$
\boxed{\begin{array}{l}
E' \;\; = \mathtt{eliminate}(\{y\}, E[z{:}\mathtt{T}(m, X)/\star{:}\mathtt{T}(m, X)]) \\
E'' \;\; = \{z{:}\mathtt{T}(n, X) \mid y{:}\mathtt{T}(f, \mathtt{fin}) \in E,\ \mathtt{T}(n, X) \in \mathit{Ret}(f)\} \\
E''' = \{y{:}\mathtt{T}(f, \overline{\mathtt{fin}}) \mid y{:}\mathtt{T}(f, \mathtt{fin}) \in E\}
\end{array}}
$$

$$
(4) \qquad \tau(b, E) = E \qquad \textit{otherwise}
$$

**Fig. 5.** Local MHP transfer function $\tau$.

In both cases, the task is associated to future variable $y$, i.e., in the concrete state that $E$ describes $y$ is bound to the unique identifier of the corresponding task. Intuitively, the MHP atoms of $E$ represent the tasks that were created locally and are executing in parallel. In what follows, we use $y{:}\mathtt{T}(m, X)$ to refer to an MHP atom without specifying if it corresponds to an active or finished task. MHP atoms might also use the symbol $\star$ instead of a future variable to indicate that we do not know to which future variable, if any, the task is bound. Note that if we have two atoms with the same future variable in a LMHP state $E$, then they are mutually exclusive, i.e., only one of the corresponding tasks might be executing since at the concrete level $y$ can be bound only to one task identifier. This might occur when merging branches of a conditional statement. Note also that MHP states are multisets because we might have several tasks created by invoking the same method. Since LMHP states are multisets, we write $(q, i) \in E$ to indicate that atom $q$ appears $i > 0$ times in $E$.

The LMHP analysis of [3], that infers the LMHP states described above, is a data-flow analysis based on the transfer function $\tau$ in Fig. 5, except for Case (3) which is novel to our extension and whose auxiliary functions will be given and explained later. Recall that the role of the transfer function in a data-flow analysis is to abstractly execute the different instructions, i.e., transforming one LMHP state to another. Let use explain the relevant cases of $\tau$:

- Case (1) handles method calls, it adds a new active task (an instance of $m$) that is bound to future variable $y$, and renames all atoms that already use $y$ to use $\star$ since it is overwritten;
- Case (2) handles **await**, it changes the state of any task bound to future variable $y$ to finished; and
- Case (4) corresponds to other instructions that do not create or wait for tasks to finish. In this case the abstract state is not affected.

In addition, the LMHP analysis merges states of conditional branches using union of multisets, and loops are iterated, with a corresponding widening operator that transforms unstable MHP atoms $(q, i)$ to $(q, \infty)$, until a fix-point is reached.

*Example 1.* Consider a method $f$ with a body **while**(∗){ y = o.m();}. The first time we apply $\tau$ over $f$, we obtain $\{y{:}\mathtt{T}(m, \mathtt{act})\}$ at the exit program point of the while. At the next iteration, we add a new atom bound to $y$ so we lose the association existing in the current state and add the new atom, obtaining $\{\star{:}\mathtt{T}(m, \mathtt{act}), y{:}\mathtt{T}(m, \mathtt{act})\}$. After applying one more iteration, we lose the relation between $y$ and the task $m$ again obtaining $\{(\star{:}\mathtt{T}(m, \mathtt{act}), 2), y{:}\mathtt{T}(m, \mathtt{act})\}$. When comparing the last two LMHP states, we observe that $\star{:}\mathtt{T}(m, \mathtt{act})$ is unstable, thus we apply widening and obtain $\{(\star{:}\mathtt{T}(m, \mathtt{act}), \infty), y{:}\mathtt{T}(m, \mathtt{act})\}$.

In what follows we present how to extend the transfer function $\tau$ and the LMHP states to handle returned futures in Case (3). We first explain it using a simple example, and then describe it formally.

*Example 2.* Assume we have a method $f$ with an instruction "**return** $x$", and that at the exit program point of $f$ we have a LMHP state $E_0 = \{x{:}\mathtt{T}(h, \mathtt{act}), w{:}\mathtt{T}(g, \mathtt{act})\}$, which means that at the exit program point of $f$ we have two active instances of methods $h$ and $g$, bound to future variables $x$ and $w$ respectively. This means that $f$ returns a future variable that is bound to an active instance of $h$. Now assume that in some other method, at some program point, we have a state $E_1 = \{y{:}\mathtt{T}(f, \mathtt{fin}), r{:}\mathtt{T}(k, \mathtt{act}), u{:}\mathtt{T}(l, \mathtt{act})\}$, which means, among other things, that before reaching the corresponding program point, we have invoked $f$ and waited for it to finish (via future variable $y$). Let us now execute the instruction $u = y.\mathbf{get}$ in the context of $E_1$ and generate a new LMHP state $E_2$. Since $y$ is bound to a task that is an instance of $f$, $E_2$ should include an atom representing that $u$ is bound to an active task which is an instance of $h$ (which is returned by $f$ via a future variable). Having this information in $E_2$ allows us to mark $h$ as finished when executing **await** $u$? later. We do this as follows:

- any MHP atom from $E_1$ that does not involve $u$ or $y$ is copied to $E_2$.
- any MHP atom from $E_1$ that involves $u$ is copied to $E_2$ but with $u$ renamed to $\star$ because $u$ is overwritten.
- we transfer the atom $x{:}\mathtt{T}(h, \mathtt{act})$ from $E_0$ to $E_2$, by adding $u{:}\mathtt{T}(h, \mathtt{act})$ to $E_2$ since now the corresponding task is bound to $u$ as well.
- the atom $y{:}\mathtt{T}(f, \mathtt{fin})$ must be copied to $E_2$ as well, but we first rewrite it to $y{:}\mathtt{T}(f, \overline{\mathtt{fin}})$ (in $E_2$) to indicate that we have incorporated the information from the exit program point of $f$ already. This is important because after executing the **get**, we will have two instances of $h$ in $E_0$ and $E_2$ that refer to the same task, and we want to avoid considering them as two different ones in the global phase that we will describe in the next section.

This results in $E_2 = \{y{:}\mathtt{T}(f, \overline{\mathtt{fin}}), r{:}\mathtt{T}(k, \mathtt{act}), \star{:}\mathtt{T}(l, \mathtt{act}), u{:}\mathtt{T}(h, \mathtt{act})\}$.

To summarize the above example, the local phase of our analysis extends that of [3] in two ways: it introduces a new kind of LMHP atom; and it has to treat the **get** instruction in a special way. In the rest of this section we formalize this extension by providing the auxiliary functions and the data-flow inference. As notation, we let $E_\ell$ be the LMHP state that corresponds to program point $\ell$;

we let $E^m_{exit}$ be the LMHP state that corresponds to the exit program point of method $m$; and we define

$$Ret(m) = \{\mathtt{T}(n, X) \mid \textbf{return}\ y \in body(m),\ y{:}\mathtt{T}(n, X) \in E^m_{exit}\},$$

which is the set of tasks in $E^m_{exit}$ that are bound to a future variable that is returned by method $m$. This set is needed in order to incorporate these tasks when abstractly executing a **get** instruction as we have seen in the example above. We also let $\mathtt{eliminate}(Y, E)$ be the LMHP set obtained from $E$ by removing all atoms that involve a future variable $y \in Y$. We first modify the transfer function of [3] to treat the instruction $z = y.\textbf{get}$, similarly to what we have done in the example above. This is done by adding Case (3) to the transfer function of Fig. 5:

- The set $E'$ is obtained from $E$ by renaming future variable $z$ to $\star$, since variable $z$ is overwritten, and then eliminating all atoms associated to future variable $y$ (they will be incorporated in $E'''$ below).
- The set $E''$ consists of new MHP atoms that correspond to futures that are returned by methods to which $y$ is bound. Note that all are now bound to future variable $z$.
- In $E'''$ we add all atoms bound to $y$ from $E$ but rewritten to mark them as *already been incorporated*.

Due to the new case added to the transfer function, we need to modify the work-flow of the corresponding data-flow analysis in order to backpropagate the information learned from the returned future variables. This is because the LMHP analysis of one method depends on the LMHP states of other methods (via $Ret(m)$ in Case (3) of $\tau$). This means that a method cannot be analyzed independently from the others as in [3], but rather we have to iterate over their analysis results, in the reverse topological order induced by the corresponding call graph, until their corresponding results stabilize.

*Example 3.* The left column of the table below shows the LMHP states resulting from applying once the $\tau$ function to selected program points, the right column shows the result after one iteration of $\tau$ over the results in the left column:

| | |
|---|---|
| $E_{11}$: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{act})\}$ | $E_{11}$: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{act})\}$ |
| $E_{12}$: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{fin})\}$ | $E_{12}$: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{fin})\}$ |
| $E_{13}$: $\tau(z = x.\textbf{get}, E_{12})$ | $E_{13}$: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$ |
| $E_{14}$: $\tau(\textbf{await}\ z?, E_{13})$ | $E_{14}$: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{fin})\}$ |
| $E_{15}$: $E_{14} \cup \{\star{:}\mathtt{T}(\mathsf{sendMessage}, \mathtt{act})\}$ | $E_{15}$: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{fin}), \star{:}\mathtt{T}(\mathsf{sendMessage}, \mathtt{act})\}$ |
| $E_{20}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$ | $E_{20}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$ |
| $E_{21}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act})\}$ | $E_{21}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act})\}$ |
| $E_{22}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act}))\}$ | $E_{22}$: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act}))\}$ |

Let us explain some of the above LMHP states. In the left column, $E_{11}$ corresponds to the state when reaching program point L11, i.e., before executing the statement **await** $x$?. It includes $x$:T(request, act) for the active task invoked at L10. The state $E_{12}$ includes the finished task corresponding to the await instruction of the previous program point. $E_{13}$ cannot be solved, as we need the information from state $E_{22}$ (it is required when calculating $E''$), which has not been computed yet. Something similar happens with the state $E_{14}$, which cannot be calculated as the state $E_{13}$ has not been totally computed. Atoms $y$:T( result , act) and $p$:T(inform, act) appear in state $E_{22}$ for the active tasks invoked at L19 and L20. The state $E_{15}$ includes $\star$:T(sendMessage, act) for the task invoked at L14, which is not bound to any future variable.

In the right column, after one iteration, we observe that most states are not modified except for $E_{13}$, $E_{14}$ and $E_{15}$. As for $E_{13}$, in the previous step we could not obtain the set $E''$ when analyzing $E_{13}$ because the function $\tau$ had not been applied to request ($E_{22}$ had not been computed). Thus, it considered $E_{13}$: $E' = \{\}$ as there was no task bound to $z$; $E'' = \{z$:T( result , act)$\}$ and; $E''' = \{y$:T(request, $\overline{\texttt{fin}}$)$\}$. Having $E_{13}$ calculated, $E_{14}$ is computed modifying the state of result to finished and $E_{15}$ is updated with the new information.

### 4.2 Global MHP

In this section we describe how to use the LMHP information, inferred by the local phase of Sect. 4.1, in order to construct an MHP graph from which an over-approximation of the set of MHP pairs can be extracted. The construction of the MHP graph is different from the one of [3] in that we need to introduce new kind of nodes to reflect the information carried by the new kind of MHP atom $y$:T($m$, $\overline{\texttt{fin}}$). However, the procedure for computing the MHP pairs from the MHP graph is the same. The *MHP graph* of a given program $P$ is a (weighted) directed graph, denoted by $\mathcal{G}_P$, whose nodes are:

- *method nodes*: each method $m \in \texttt{methods}(P)$ contributes 3 nodes $\texttt{act}(m)$, $\texttt{fin}(m)$ and $\overline{\texttt{fin}}(m)$. We use $X(m)$ to refer to a method node without specifying if it corresponds to $\texttt{act}(m)$, $\texttt{fin}(m)$, or $\overline{\texttt{fin}}(m)$.
- *program point nodes*: each program point $\ell \in \texttt{ppoints}(P)$ contributes a node $\ell$.
- *return nodes*: each program point $\ell \in \texttt{ppoints}(P)$ that is an exit program point, of some method $m$, contributes a node $\bar{\ell}$.
- *future variable nodes*: each future variable $y \in \texttt{futures}(P)$ and program point $\ell \in \texttt{ppoints}(P)$ contribute a node $\ell_y$ (which can be ignored if $y$ does not appear in the corresponding LMHP state of $\ell$).

Note that nodes $\overline{\texttt{fin}}(m)$ and $\bar{\ell}$ are particular to our extension, they do not appear in [3] and will be used, as we will see later, to avoid duplicating tasks that are returned to some calling context.

The edges of $\mathcal{G}_P$ are constructed in two steps. First we construct those that do not depend on the LMHP states, and afterwards those that are induced by

LMHP states. The first kind of edges are constructed as follows, for each method $m \in \texttt{methods}(P)$:

- there are edges from $\texttt{act}(m)$ to all *program point nodes* $\ell \in \texttt{ppoints}(m)$. This kind of edges indicate that an active task can be executing at any program point, including its exit program point;
- there is an edge from $\texttt{fin}(m)$ to the exit *program point node* $\ell$ of $m$. This kind of edges indicate that a finished task can be only at the exit program point;
- there is an edge from $\overline{\texttt{fin}}(m)$ to the corresponding *return node* $\bar{\ell}$, i.e., $\ell$ here is the exit program point of $m$. This kind of edges are similar to the previous ones, but they will be used to avoid duplicating tasks that were returned to some calling context.

All the above edges have weight 0. Next we construct the edges induced by the LMHP states. For each program point $\ell \in \texttt{ppoints}(P)$, we consider $E_\ell$ and construct the following edges:

- if $(\star{:}\texttt{T}(m, X), i) \in E_\ell$, we add an edge from node $\ell$ to node $X(m)$ with weight $i$. If $\ell$ is an exit program point we also add an edge from node $\bar{\ell}$ to node $X(m)$ with weight $i$;
- if $(y{:}\texttt{T}(m, X), i) \in E_\ell$, we add an edge from node $\ell$ to node $\ell_y$ with weight 0 and an edge from node $\ell_y$ to node $X(m)$ with weight $i$. In addition, if $\ell$ is an exit program point and $y$ is not a returned future we add an edge from node $\bar{\ell}$ to node $\ell_y$ with weight 0.

Note that when $\ell$ is an exit program point, the difference between node $\ell$ and $\bar{\ell}$ is that the later ignores tasks that were returned via future variables.

*Example 4.* Figure 6 shows the MHP graph for some program points of interest for our running example. Note that the out-going edges of program point nodes in $\mathcal{G}$ coincide with the LMHP states at these program points depicted in Example 3. At program point $L15$, the LMHP state $E_{15}$ contains the atoms $x{:}\texttt{T}(\mathsf{request}, \overline{\texttt{fin}})$, $z{:}\texttt{T}(\mathsf{result}, \texttt{fin})$ and $\star{:}\texttt{T}(\mathsf{sendMessage}, \texttt{act})$. Each of these atoms corresponds to one of the edges from program point node 15. The first one is represented by the edge that goes from program point node 15 to future variable node $15_x$ and from $15_x$ to method node $\overline{\texttt{fin}}(\mathsf{request})$. The second one corresponds to the edge that goes from 15 to $15_z$ and from there to method node $\texttt{fin}(\mathsf{result})$. The edge which goes from 15 to method node $\texttt{act}(\mathsf{sendMessage})$ originates from the MHP atom $\star{:}\texttt{T}(\mathsf{sendMessage}, \texttt{act})$. This last edge does not go to a future variable node as the task is not bound to any future variable ($\star$). Note that we have two nodes 22 and $\overline{22}$ to represent the exit program point L22, connected to $\texttt{fin}(\mathsf{request})$ and $\overline{\texttt{fin}}(\mathsf{request})$. The edges that go out from 22 correspond to the atoms in $E_{22}$. As L22 is the exit program point of method request, we have to build an edge. This edge goes from $\overline{22}$ to $22_p$ and from there to $\texttt{act}(\mathsf{inform})$ and corresponds to the atom in $E_{22}$ whose future variable is not returned by request.

Given $\mathcal{G}_P$, using the same procedure as in [3], we say that two program points $\ell_1, \ell_2$ may run in parallel if one of the following conditions hold:

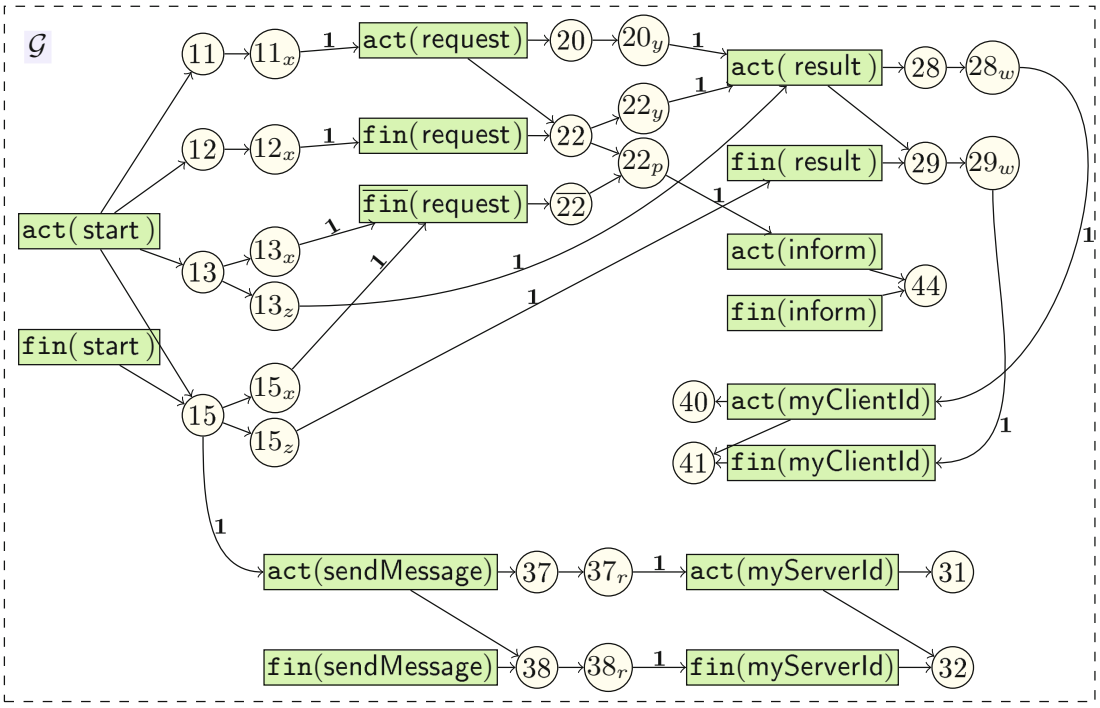



**Fig. 6.** MHP graph obtained from the analysis of program in Fig. 2.

1. there is a non-empty path from $\ell_1$ to $\ell_2$ or vice-versa; or
2. there is a program point $\ell_3$ and non-empty paths from $\ell_3$ to $\ell_1$ and from $\ell_3$ to $\ell_2$ such that the first edge is different, or they share the first edge but it has weight $i > 1$.

The first case is called *direct* MHP pairs and the second one *indirect* MHP pairs.

*Example 5.* Let us explain some of the MHP pairs shown in Fig. 4 and induced by $\mathcal{G}$. (22,28) and (22,44) are direct MHP pairs as we can find the paths $22 \rightsquigarrow 44$ and $22 \rightsquigarrow 44$ in $\mathcal{G}$. In addition, as the first edge is different, we can conclude that (22,44) is an indirect pair. In contrast to the graph that one would obtain for the original analysis, (15,28) is not an MHP pair (marked in bold in Fig. 4).

Instead, we have the path $15 \rightsquigarrow 29$ which indicates that the task result is finished. Similarly, the analysis does not infer the pair (28,37), allowing us to discard the deadlock cycle described in Sect. 3. We find the path $15 \rightsquigarrow 37$ in $\mathcal{G}$, but the path $15 \rightsquigarrow 28$, needed to infer this spurious pair, is not in $\mathcal{G}$.

Let $\tilde{\mathcal{E}}_P$ be the set of MHP pairs obtained by applying the procedure above.

**Theorem 1 (soundness).** $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

## 5 Implementation and Experimental Evaluation

The analysis presented in Sect. 4 has been implemented in SACO [2], a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination

| Examples | Lines | N | $PPs^2$ | OMHPs | MHPs | Lmhp | Gmhp | Mhp | OT | T |
|---|---|---|---|---|---|---|---|---|---|---|
| ServerClient | 69 | 19 | 361 | 176 | 140 | <5 | <5 | 22 | 64 | 60 |
| Chat | 331 | 73 | 5329 | 1351 | 1028 | <5 | <5 | 606 | 1686 | 1014 |
| MailServer | 140 | 28 | 784 | 315 | 284 | <5 | <5 | 47 | 172 | 166 |
| DistHT | 168 | 27 | 729 | 392 | 367 | <5 | <5 | 51 | 186 | 183 |
| PeerToPeer | 215 | 42 | 1764 | 325 | 297 | 8 | <5 | 112 | 452 | 238 |
| ETICS | 1717 | 297 | 88209 | 30554 | 30523 | 175 | 20 | 40887 | 53023 | 53450 |
| $TradingSys^1$ | 1508 | 216 | 46656 | 40562 | 33038 | 53 | 15 | 13260 | 32065 | 31589 |
| $TradingSys^2$ | 1508 | 216 | 46656 | 41345 | 41345 | 62 | 26 | 11765 | 30308 | 31057 |

**Fig. 7.** Examples and statistics

and resource boundedness [14]. Our analysis has been built on top of the original MHP analysis in SACO and can be tried online at: http://costa.ls.fi.upm.es/saco/web/ by selecting `MHP` from the menu as type of analysis, then enabling the option `Global Futures Synchronization` in the `Settings` section, and clicking on `Apply`. The benchmarks are also available in the folder `ATVA17`. Given a program with a main procedure, the analysis returns a list of MHP pairs and some statistics about the runtime of the local and global phases.

Figure 7 summarizes our experiments. The first benchmark ServerClient corresponds to the complete implementation of our running example. The next four are some traditional programs for distributed and concurrent programming: Chat models a chat application, MailServer models a distributed mail server with several users, DistHT implements and uses a distributed hash table and PeerToPeer which represents a peer-to-peer network. The last two examples, ETICS and TradingSys are industrial case studies, respectively, developed by Engineering® and Fredhopper® that model a system for remotely hosting and managing IT resources and a system to manage sales and other facilities on a large product database. These case studies are very conservative on the use of futures (namely only 3 tasks return a future), however, we have included them to assess the efficiency of our analysis on large programs. For the TradingSys, we have two versions, `TradingSys`$^1$ which creates a constant number of tasks (namely 3), and `TradingSys`$^2$ which creates an unknown number of tasks within a loop. Experiments have been performed on an Intel Core i7-6500U at 2.5 GHz x 4 and 7.5GB of Memory, running Ubuntu 16.04. For each program $P$, $\mathcal{G}_P$ is built and the relation $\tilde{\mathcal{E}}_P$ is computed for those points that affect the concurrency of the program (i.e., entry points of methods, awaits, gets and exit points of methods).

Let us first discuss the accuracy of our approach. Columns **Examples** and **Lines** show the name and number of lines of the benchmark. **N** is the number of program point nodes in $\mathcal{G}_P$. **$PPs^2$** is the square of the number of program points, i.e., the total number of pairs that could potentially run in parallel. **OMHPs** and **MHPs** show the number of MHP pairs inferred by the original analysis [3] and by ours. **$PPs^2$-MHPs** is thus the number of MHP pairs that are detected not to happen in parallel by the original analysis. Naturally the original analysis already eliminates many pairs that arise from local future variables (not

returned). **OMHPs-MHPs** gives us the number of further spurious MHP pairs that our analysis eliminates. We can observe that for all examples (except for `TradingSys`$^2$) we reduce the number of inferred MHP pairs (ranging from a small reduction of 0.2% pairs for `ETICS` to a big reduction of 23.9% for `Chat`). In `TradingSys`$^2$ we do not eliminate any pair because the tasks created within the loop use the same future variable to return their results, and the analysis needs to over-approximate and assume that all of them may run in parallel.

As regards the efficiency of the analysis, the next three columns contain the time (in milliseconds) taken by the local MHP (**Lmhp**), the graph construction (**Gmhp**) and the time needed to infer the MHP pairs (**Mhp**). The data presented are the average time obtained across several executions. We can observe that both LMHP and the graph construction are very efficient and they only take 0.175 s in the largest case. The inference of the MHP pairs is more complex and takes more time. This time depends on the number of program point nodes that the graphs contain. For medium programs, the inference technique is also efficient (taking 0.6 s in the largest case), but the time increases notably in bigger examples, reaching 40.8 s in our experiments. However, in most applications we are only interested in a subset of pairs. Besides, the pairs can be computed on demand, spending less time to infer them. The last two columns contain the total time (in milliseconds) taken by the analysis of [3] (**OT**) and our approach (**T**). It can be observed that our analysis is more efficient than the original one for all examples except for the `TradingSys`$^2$ and `ETICS`, being the overhead negligible in these cases (less than 2.5%). The reason for the efficiency gain is that when returned futures are tracked, our graph contains less paths that are inspected to infer the MHP pairs. Thus, the process of computing all the feasible paths is faster in these cases, and the global time of the analysis is smaller than [3].

## 6 Conclusions and Related Work

An MHP analysis learns from the future variables used in synchronization instructions when tasks are terminated, so that the analysis can accurately eliminate unfeasible MHP pairs that would be otherwise inferred. Some existing MHP analyses [1,3,15,16] for asynchronous programs lose all the information when future variables are awaited in a different scope to the one that spawns the tasks bound to the futures. We have presented a static MHP analysis which captures inter-procedural MHP relations in which future variables are propagated *backwards* from one task to another(s). This implies that a task can be awaited in an outer scope from the one in which it was created. Previous work [5] has considered the propagation of future variables *forward*, i.e., when future variables are passed as arguments of the tasks. This implies that a task can be awaited in an inner scope from the one in which it was created. Also, other MHP analyses allow synchronizing the termination of the tasks in an inner scope, passing them as arguments of methods, namely: [11] considers a fork-join semantics and uses a Happens-Before analysis to infer the MHP information; in [6,8], programs are abstracted to a thread model which is then analyzed to infer the MHP pairs; [9]

builds a time based model to infer race conditions in high performance systems; this work is extended in [7], using a model checker to solve the MHP decision problem. The last six analyses are imprecise though when future variables or the tasks identifiers are returned by methods and awaited in an outer scope.

The solutions for the backwards and forward inference (namely as formalized in [5]) are technically different, but fully compatible. Essentially, they only have in common that both the local and global analysis phases need to be changed. For the forward inference, the analysis includes a separated *must-have-finished* (MHF) pre-analysis that allows inferring, for each program point $\ell$, which tasks (both the tasks spawned locally and the passed as arguments) have finished their execution when reaching $\ell$. In contrast, for the backwards inference, the local phase itself has to be extended to propagate backwards the new relations created when a future variable is returned, which requires changing the analysis flow. In both analyses, the creation of the graph needs to be modified to reflect the new information inferred by the respective local phases, but in each case is different. For the forward inference, the way in which the MHP pairs are inferred besides has to be modified. All in all, both extensions are fully compatible, and together provide a full treatment of future variables in the MHP analysis.

## References


1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of x10 programs. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) Proceedings of PPOPP 2007, pp. 183–193. ACM (2007)
2. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54862-8_46
3. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of may-happen-in-parallel in concurrent objects. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE - 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30793-5_3
4. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: Termination and cost analysis of loops with concurrent interleavings. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 349–364. Springer, Cham (2013). doi:10.1007/978-3-319-02444-8_25
5. Albert, E., Genaim, S., Gordillo, P.: May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 72–89. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48288-9_5
6. Barik, R.: Efficient computation of may-happen-in-parallel information for concurrent Java programs. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 152–169. Springer, Heidelberg (2006). doi:10.1007/978-3-540-69330-7_11
7. Chang, C.W., Dömer, R.: May-happen-in-parallel analysis of ESL models using UPPAAL model checking. In: DATE 2015, pp. 1567–1570. IEEE, March 2015

8. Chen, C., Huo, W., Li, L., Feng, X., Xing, K.: Can we make it faster? Efficient may-happen-in-parallel analysis revisited. In PDCAT 2012, pp. 59–64, December 2012
9. Chen, W., Han, X., Dömer, R.: May-happen-in-parallel analysis based on segment graphs for safe ESL models. In: DATE 2014, pp. 1–6. IEEE, March 2014
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71316-6_22
11. Di, P., Sui, Y., Ye, D., Xue, J.: Region-based may-happen-in-parallel analysis for C programs. In: ICPP 2015, pp. 889–898. IEEE, September 2015
12. Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: POPL 1995, 22nd ACM SIGPLAN-SIGACT (1995)
13. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FMOODS/-FORTE -2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38592-6_19
14. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25271-6_8
15. Lee, J.K., Palsberg, J., Majumdar, R., Hong, H.: Efficient may happen in parallel analysis for async-finish parallelism. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 5–23. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33125-1_4
16. Sankar, A., Chakraborty, S., Nandivada, V.K.: Improved MHP analysis. In: CC 2016, pp. 207–217. ACM, September 2016

## Proofs

Our proof for Theorem 1 is an extension of the proof of the original analysis as it appears in its journal version [21]. We use some of the auxiliary notions of [21] as included below.

### Auxiliary results

Let $\mathcal{L}_P$ be the result of the local phase presented in Section 4.1. In order to prove the soundness of the analysis we have to extend the semantics. The new semantics is shown in Figure 7.1. We add to each task additional information $\mathcal{L}r$, that records the calls that the current task has performed and their status. Then, these tasks are terms of the form $tsk(t, m, l, o, lk, s, \mathcal{L}r)$. $\mathcal{L}r$ can be seen as a concrete version of $\mathcal{L}_P$ and contains information about the future variable related to the task called, if there is one, and the state of the task ($\mathtt{T}(tid, \mathtt{act})$,$\mathtt{T}(tid, \mathtt{fin})$,$\mathtt{T}(tid, \overline{\mathtt{fin}})$).

We use the notion of *runtime* MHP $\mathcal{E}_P^r$, that appears in Definition A.1 in [21] to represent the MHP information in the runtime in a given program $P$ or $\mathcal{E}_S^r$ to represent it in a specific state $S$: given a program $P$, we let $\mathcal{E}_P^r = \cup\{\mathcal{E}_S^r \mid S_0 \rightsquigarrow^* S\}$ where $\mathcal{E}_S^r$ is defined as

$$\mathcal{E}_S^r = \left\{ ((tk_1, pp(s_1)), (tk_2, pp(s_2))) \;\middle|\; \begin{array}{l} tsk(tk_1, \_, \_, \_, \_, s_1, \_) \in S, \\ tsk(tk_2, \_, \_, \_, \_, s_2, \_) \in S, \\ tk_1 \neq tk_2 \end{array} \right\}.$$

**Definition 1 (Concrete MHP graph).** *Given a state $S$, we define a concrete graph $\mathcal{G}r_S$ using $\mathcal{L}r$ as follows*

$$\begin{array}{rl} \mathcal{G}r_S = & \langle V_S, E_S \rangle \\ V_S = & \{\mathtt{act}(tk), \mathtt{fin}(tk), \overline{\mathtt{fin}}(tk) \| \; tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S\} \cup cP_S \\ cP_S = & \{(tk, pp(s)) | \; tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S\} \\ E_S = & ei_S \cup el_S \\ ei_S = & \{\mathtt{act}(tk) \rightarrow (tk, pp(s)) | \; tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S\} \\ & \cup\{\mathtt{fin}(tk) \rightarrow (tk, exit(m)) | \; tsk(tk, m, l, o, lk, \epsilon(v), \mathcal{L}r) \in S\} \\ & \cup\{\overline{\mathtt{fin}}(tk) \rightarrow (tk, exit(m)) | \; tsk(tk, m, l, o, lk, \epsilon(v), \mathcal{L}r) \in S\} \\ el_S = & \{(tk, pp(s)) \rightarrow x | \; tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S \wedge \_ : x \in \mathcal{L}r\} \end{array}$$

Once the graph has been constructed, we use the notion of *MHP-Graph relation* $\mathcal{EG}_S^r$ to obtain the concrete relations over this graph:

$$\begin{array}{rl} \mathcal{EG}_S^r = & dMHP_S \cup iMHP_S \\ dMHP_S = & \{(x, y) | x, y \in cP_S \wedge x \rightsquigarrow y\} \\ iMHP_S = & \{(x, y) | x, y \in cP_S \wedge (\exists z \in cP_S : z \rightsquigarrow x \wedge z \rightsquigarrow y)\} \end{array}$$

The abstraction function $\varphi$ allows us to obtain the set $\mathcal{E}_P$ from $\mathcal{E}_P^r$.

**Definition 2 (MHP relation abstraction).** *Let $\varphi$ be the abstraction function such that $\varphi(tk_1, p_1) = p_1$. The abstraction of the MHP-Graph relation $\mathcal{EG}^r_S$ is the abstraction of each of its pairs.*

**Definition 3.** *$\psi$ abstracts $\mathcal{L}r$ sets into multisets; $\psi'$ abstracts a single mhp atom; and $\psi''$ abstracts tasks into methods.*

$\psi''(\mathtt{T}(tk, \mathtt{act})) = \mathtt{T}(m, \mathtt{act})$
$\psi''(\mathtt{T}(tk, \mathtt{fin})) = \mathtt{T}(m, \mathtt{fin})$
$\psi''(\mathtt{T}(tk, \overline{\mathtt{fin}})) = \mathtt{T}(m, \overline{\mathtt{fin}})$ *where* $m = method(tk)$
$\psi'(y{:}\mathtt{T}(tk, X)) = y{:}\psi''(\mathtt{T}(tk, X))$
$\psi(\mathcal{L}r) = \{(\psi'(a), i) | a \in \mathcal{L}r \wedge (\#i : b \in \mathcal{L}r : \psi'(a) = \psi'(b))\}$

**Lemma 1 (Soundness of $\mathcal{L}_P$).**

$$\forall S : S_i \rightsquigarrow^* S : tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S \Rightarrow \psi(\mathcal{L}r) \sqsubseteq \mathcal{L}_P(\varphi(tk, pp(s)))$$

*The result of the local phase presented in Section 4.1 is a safe approximation of the concrete property defined in the semantics.*

*Proof.* The proof is a direct correspondence between the transformations on $\mathcal{L}r$ applied in the rules showed in the extended semantics in Figure 7.1 and the transition function $\tau$ defined in Section 4.1. □

Lemma A.8 in [21], which proves that any path in the concrete MHP graph has a corresponding abstract path in the MHP graph, holds in our case considering also the new state added in Section 4.1.

## Proof of Theorem 1

In order to prove Theorem 1 we will need two more results. The first one expresses that $\mathcal{EG}^r_S$ captures the concurrency information of a state $S$.

**Theorem 2.**

$$\forall S : (S_0 \rightsquigarrow^* S) \Rightarrow (\mathcal{E}^r_S \subseteq \mathcal{EG}^r_S)$$

*Proof.* The proof of the theorem is similar to the one presented in [21], including the new task state defined in Section 4.1 and the new get instruction.

Theorem 2 is equivalent to:

$$\begin{array}{l}\forall S : S_0 \rightsquigarrow^* S : \\ \quad \forall\ tsk(tk_1, m_1, l_1, o_1, lk_1, s_1, \mathcal{L}r_1), tsk(tk_2, m_2, l_2, o_2, lk_2, s_2, \mathcal{L}r_2) \in S : \\ \quad tk_1 \neq tk_2 : ((tk_1, pp(s_1)), (tk_2, pp(s_2))) \in \mathcal{EG}^r_S\end{array}$$

However, it is sufficient to prove that every task is reachable from the main node $(0, pp(s_0))$ that corresponds to the main task $(tsk(0, \mathsf{main}, l_0, 0, \top, s_0, \mathcal{L}r)$. This can be expressed:

$$\begin{array}{l}\forall S : S_i \rightsquigarrow^* S : \\ \exists\ \ tsk(0, \mathsf{main}, l_0, 0, \top, s_0, \mathcal{L}r) in S : \\ \forall\ tsk(tk_1, m_1, l_1, o_1, lk_1, s_1, \mathcal{L}r_1) \in S : tk_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk_1, pp(s_1))\end{array}$$

In such case, for every two tasks either one of them is the main one and the other is reachable from it or both are different from the main one and they belong to *iMHP*. We can prove it by induction on the states of the program:

**Base case:** Only the main task is present.

$\forall tsk(tk_1, m_1, l_1, o_1, lk_1, s_1, \mathcal{L}r_1) \in T : tk_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk_1, pp(s_1))$ trivially holds.

**Inductive case:** For any possible transition $S \rightsquigarrow S'$. The induction hypothesis is:

$\exists\ tsk(0, \mathsf{main}, l_0, 0, \top, s_0, \mathcal{L}r) \in S :$

$\forall\ tsk(tk_1, m_1, l_1, o_1, lk_1, s_1, \mathcal{L}r_1) \in S : tk_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk_1, pp(s_1))$

Although most semantic rules have several effects on the program state, they can be split into steps. Each step is proved to maintain the property. Finally, each semantic rule is expressed as a combination of simple steps.

1. Sequential step: The new state $S'$ can be obtained through a substitution $S' = S\tau$ of the form:

   $\tau = \{tsk(tk, m, l, o, lk, s, \mathcal{L}r)/tsk(tk, m, l', o, lk, s', \mathcal{L}r')\}$.

   $tsk(tk, m, l', o, lk, s', \mathcal{L}r') \in S$ and the object $o$ where it is executing has the lock of the task. $\mathcal{G}r_{S'} = \langle V_{S'}, E_{S'}\rangle$ and $\mathcal{G}r_S = \langle V_S, E_S\rangle$ are isomorphic graphs and we can define a graph bijection as a substitution:

   $$V'_S = V_S[(tk, pp(s))/(tk, pp(s'))]$$

   It is easy to see that the given substitution is indeed a bijection. Let $a \to b$ and edge of $\mathcal{G}r_S$ we have one of the following:

   (a) Both $a$ and $b$ are not $(tk, pp(s))$. In this case, $a \to b$ is in $\mathcal{G}r_{S'}$ as they are not affected by the substitution.

   (b) $a = (tk, pp(s))$. This implies that $tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in T$ and $\_{:}b \in \mathcal{L}r$ where $\_$ can be a future variable or $\star$.
   We have that $tsk(tk, m, l', o, lk, s', \mathcal{L}r) \in S'$ with the same $\mathcal{L}r$ thus $(tk, pp(s')) \to b$ is in $\mathcal{G}r_{S'}$.

   (c) $a \to b = \mathtt{act}(tk) \to (tk, pp(s))$. This implies that $tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S$. We have that $tsk(tk, m, l', o, lk, s', \mathcal{L}r) \in S'$. $\mathtt{act}(tk) \to (tk, pp(s'))$ is in $\mathcal{G}r_{S'}$ by definition.

   (d) There cannot be edges of the form $\mathtt{fin}(tk) \to (tk, pp(s))$ or $\overline{\mathtt{fin}}(tk) \to (tk, pp(s))$ because they require that $tsk(tk, m, l, o, lk, s, \mathcal{L}r)$ does not have the lock and that contradicts our condition that object $o$ has the lock of the task.

   Once concluded that the graphs are isomorphic the induction hypothesis can be applied to conclude:

   $\exists\ tsk(0, \mathsf{main}, l_0, 0, \top, s_0, \mathcal{L}r) \in S' : \forall\ tsk(tk_1, m_1, l_1, o_1, lk_1, s_1, \mathcal{L}r_1) \in S' :$
   $tk_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_1, pp(s_1))$.

2. Loss of a future variable association:
   $S' = S[tsk(tk, m, l, o, lk, s, \mathcal{L}r)/tsk(tk, m, l, o, lk, s, \mathcal{L}r')]$ where
   $\mathcal{L}r' = \mathcal{L}r[y{:}\mathtt{T}(tk_n, X)/\star{:}\mathtt{T}(tk_n, X)]$. Such substitution does not change the graph as atoms $y{:}\mathtt{T}(tk_n, X)$ and $\star{:}\mathtt{T}(tk_n, X)$ generate the same edges and the nodes remain unchanged.

3. New task added:
   $S' = S[tsk(tk, m, l, o, lk, s, \mathcal{L}r)/tsk(tk, m, l, o, lk, s, \mathcal{L}r')] \cup$
   $\cup\{tsk(tk_1, m_1, l_1, o_1, \bot, body(m_1), \emptyset)\}$ where $\mathcal{L}r' = \mathcal{L}r \cup \{y{:}\mathtt{T}(tk_1, \mathtt{act})\}$.

   $\mathcal{G}r_{S'} = \langle V', E' \rangle$ where:

   - $V' = V \cup \{\mathtt{act}(tk_1), \mathtt{fin}(tk_1), \overline{\mathtt{fin}}(tk_1), (tk_1, entry(m_1))\}$ and;
   - $E' = E \cup \{(tk, s) \rightarrow \mathtt{act}(tk_1), \mathtt{act}(tk_1) \rightarrow (tk_1, entry(m_1))$ where $entry(m)$ refers to the entry program point of method $m$).

   In this case, $\mathcal{G}r_{S'} \supseteq \mathcal{G}r_S$ so any path in $\mathcal{G}r_S$ is still valid in $\mathcal{G}r_{S'}$. Applying the induction hypothesis we conclude that for any task

   $tsk(tk_2, m_2, l_2, o_2, lk_2, s_2, \mathcal{L}r_2) \in S, (0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_2, s_2)$.

   The only task that is added to $S'$ is $tsk(tk_1, m_1, l_1, o_1, \bot, body(m_1), \emptyset)$. But the program point in this task is reachable from $tsk(tk, m, l, o, lk, s, \mathcal{L}r')$ as we can create a path $p$ from $(tk, pp(s))$ to $(tk_1, entry(m_1))$:

   $p = (tk, pp(s)) \rightarrow \mathtt{act}(tk_1), \mathtt{act}(tk_1) \rightarrow (tk_1, entry(m_1))$.

   We have already proved that $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk, pp(s))$ and
   $(tk, pp(s)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_1, entry(m_1))$. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_1, entry(m_1))$.

4. Task ending:

   $S' = S[tsk(tk, m, l, o, lk, s, \mathcal{L}r)/tsk(tk, m, l, o, lk, s, \mathcal{L}r')]$ where
   $tsk(tk_1, m_1, l_1, o_1, \bot, \epsilon(v), \mathcal{L}r) \in S$ and
   $\mathcal{L}r' = \mathcal{L}r[y{:}\mathtt{T}(tk_1, \mathtt{act})/y{:}\mathtt{T}(tk_1, \mathtt{fin})\}$.

   For a given future variable $y$ there is at most one atom in $\mathcal{L}r$. If there is none, $S' = S$ and the property holds. Otherwise, one atom $y{:}\mathtt{T}(tk_1, \mathtt{act})$ gets substituted by $y{:}\mathtt{T}(tk_1, \mathtt{fin})$.

   This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph. By the graph definition we see that changes in a $\mathcal{L}r$ set affect the edges in $el_S$:

   $el_{S'} = el_S \setminus \{(tk, pp(s)) \rightarrow \mathtt{act}(tk_1)\} \cup \{(tk, pp(s)) \rightarrow \mathtt{fin}(tk_1)\}$.

   Given a task $tsk(tk_2, m_2, l_2, o_2, lk_2, s_2, \mathcal{L}r_2) \in S$, by the induction hypothesis we know that $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk_2, pp(s_2))$. That is, there is a path $p$ from $(0, s)$ to $(tk_2, pp(s_2))$.

   If $p_y \rightarrow \mathtt{act}(tk_1)$ does not appear in $p$, then $p$ is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$ and $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_2, pp(s_2))$.

If $(tk, pp(s)) \to \texttt{act}(tk_1)$ appears in $p$, then $p = x_1 \to x_2, x_2 \to x_3 \cdots$ $\cdots (tk, pp(s)) \to \texttt{act}(tk_1), \texttt{act}(tk_1) \to (tk_1, exit(m_1)) \cdots x_{n-1} \to x_n$. We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tk, pp(s)) \to \texttt{fin}(tk_1)$, $\texttt{fin}(tk_1) \to (tk_1, exit(m_1)) \cdots x_{n-1} \to x_n$.

This new path $p'$ is valid in $\mathcal{G}r_{S'}$ as $(tk, pp(s)) \to \texttt{fin}(tk_1)$ is the edge added in $el_{S'}$ and $\texttt{fin}(tk_1) \to (tk_1, exit(m_1))$ belongs to $\mathcal{G}r_{S'}$ and $\mathcal{G}r_S$ by definition. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_2, pp(s_2))$.

5. Take lock:

   $S' = S[tsk(tk, m, l, o, \bot, s, \mathcal{L}r), obj(o, f, \top)$
   $/tsk(tk, m, l, o, \top, s, \mathcal{L}r), obj(o, f, \bot)]$

   This transformation can not affect any path between program points and does not change the graph. It only allows continuing the execution.

6. Change ending state:

   $S' = S[tsk(tk, m, l, o, lk, s, \mathcal{L}r)/tsk(tk, m, l, o, lk, s, \mathcal{L}r')]$ where
   $\mathcal{L}r' = \mathcal{L}r[y{:}\texttt{T}(tk_1, \texttt{fin})/y{:}\texttt{T}(tk_1, \overline{\texttt{fin}})]$.

   For a given future variable $y$ there is at most one atom in $\mathcal{L}r$. If there is none, $S' = S$ and the property holds. Otherwise, one atom $y{:}\texttt{T}(tk_1, \texttt{fin})$ is substituted by $y{:}\texttt{T}(tk_1, \overline{\texttt{fin}})$. This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph.

   $el_{S'} = el_S \setminus \{(tk, pp(s)) \to \texttt{fin}(tk_1)\} \cup \{(tk, pp(s)) \to \overline{\texttt{fin}}(tk_1)\}$.

   Given a task $tsk(tk_2, m_2, l_2, o_2, lk_2, s_2, \mathcal{L}r_2) \in S$, by the induction hypothesis it is guaranteed that $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk_2, pp(s_2))$. That is, there is a path $p$ from $(0, s)$ to $(tk_2, pp(s_2))$.

   If $p_y \to \texttt{fin}(tk_1)$ does not appear in $p$, then $p$ is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$ and $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_2, pp(s_2))$.

   If $(tk, pp(s)) \to \texttt{fin}(tk_1)$ appears in $p$, then $p = x_1 \to x_2, x_2 \to x_3 \cdots$ $\cdots (tk, pp(s)) \to \texttt{fin}(tk_1), \texttt{fin}(tk_1) \to (tk_1, exit(m_1)) \cdots x_{n-1} \to x_n$. We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tk, pp(s)) \to \overline{\texttt{fin}}(tk_1)$, $\overline{\texttt{fin}}(tk_1) \to (tk_1, exit(m_1)) \cdots x_{n-1} \to x_n$.

   This new path $p'$ is valid in $\mathcal{G}r_{S'}$ as $(tk, pp(s)) \to \overline{\texttt{fin}}(tk_1)$ is the edge added in $el_{S'}$ and $\overline{\texttt{fin}}(tk_1) \to (tk_1, exit(m_1))$ belongs to $\mathcal{G}r_{S'}$ and $\mathcal{G}r_S$ by definition. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk_2, pp(s_2))$.

7. Get instruction:

   $S' = S[tsk(tk_1, m_1, l_1, o_1, \top, s_1, \mathcal{L}r_1), tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v), \mathcal{L}r_2)$
   $/tsk(tk_1, m_1, l_1, o_1, \top, s_1, \mathcal{L}r'_1), tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v), \mathcal{L}r'_2)$ where
   $\mathcal{L}r'_1 = \mathcal{L}r_1 \cup \{z{:}\texttt{T}(tk_3, X)\}$, $\mathcal{L}r'_2 = \mathcal{L}r_2 \backslash \{x{:}\texttt{T}(tk_3, X)\}$ and $y{:}\texttt{T}(tk_2, X) \in \mathcal{L}r_1$.

   These changes have no effect on the graph nodes, $V'_S = V_S$. However, the edges of the graph are modified.

$el_{S'} = el_S \setminus \{(tk_2, exit(m_2)) \to \texttt{act}(tk_3)\} \cup \{(tk, pp(s_1)) \to \texttt{act}(tk_3)\}$, or $el_{S'} = el_S \setminus \{(tk_2, exit(m_2)) \to \texttt{fin}(tk_3)\} \cup \{(tk, pp(s_1)) \to \texttt{fin}(tk_3)\}$.

Given a task $tsk(tk, m, l, o, lk, s, \mathcal{L}r) \in S$, by the induction hypothesis there is a path $p$, $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tk, pp(s))$.

If $p_y \to \texttt{fin}(tk_2)$ does not appear in $p$ we are not able to reach the node $\texttt{act}(tk_3)$ (or $\texttt{fin}(tk_3)$). In this case $p$ is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$ and $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk, pp(s))$.

If $(tk_1, pp(s_1)) \to \texttt{fin}(tk_2)$ appears in $p$, then $p = x_1 \to x_2, x_2 \to x_3 \cdots (tk_1, pp(s_1)) \to \texttt{fin}(tk_2), \texttt{fin}(tk_2) \to (tk_2, exit(m_2)), (tk_2, exit(m_2)) \to \texttt{act}(tk_3) \cdots x_{n-1} \to x_n$.

We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tk_1, pp(s_1)) \to \texttt{act}(tk_3) \cdots x_{n-1} \to x_n$.

This new path $p'$ is valid in $\mathcal{G}r_{S'}$ as $(tk_1, pp(s_1)) \to \texttt{act}(tk_3)$ is the edge added in $el_{S'}$, and the path $p'$ does not contain the edge $(tk_2, exit(m_2)) \to \texttt{act}(tk_3)$ which is not in $el_{S'}$. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tk, pp(s))$.

The case in which the task $tk_3$ has finished its execution is analogous.

Finally, we can express the semantic rules as combination of basic steps:

- (NewObject) is an instance of sequential step (1) with the addition of a new object $a(o', f', \bot)$ that does not affect the graph.
- (Select) is an instance of Take lock step (6).
- (Async) is an instance of sequential step (1) followed by loss of future variable association (3) and New task added (4).
- (Await) is a sequential step (1) followed by task ending (5).
- (Get) is a sequential step (1) followed by loss of future variable association (3), change the ending state (6) and get the instruction returned (7).
- (Return) is a sequential step (1) followed by a release (2).
- (Sequential) is a sequential step (1).

□

The second one corresponds to the Theorem A.10 presented in [21] that does not need to be modified in our case. This theorem states that for any pair obtained in the concrete graph of a given state ($\mathcal{EG}_S$) is also obtained by the analysis $\tilde{\mathcal{E}}_P$:

$$\forall S : S_0 \rightsquigarrow^* S : \mathcal{EG}_S \subseteq \tilde{\mathcal{E}}_P$$

These two lemmas thus prove the desired soundness of $\mathcal{E}_P$

*Proof of Theorem 1: soundness of $\mathcal{E}_P$.*

$$\mathcal{E}_P = \varphi(\mathcal{E}_P^r) = \varphi(\cup_S \mathcal{E}_S^r) \overset{Theorem2}{\subseteq} \varphi(\cup_S \mathcal{E}\mathcal{G}_S^r) = \cup_S \varphi(\mathcal{E}\mathcal{G}_S^r) = \cup_S \mathcal{E}\mathcal{G}_S \overset{TheoremA.10}{\subseteq} \tilde{\mathcal{E}}_P$$

□

$$(1)\quad \frac{l' = l[o \to o_1], O' = O \cup \{obj(o_1, a, \bot)\}, a = init\_atts(C, \overline{x}), o_1 \text{ is a fresh id}}{\begin{array}{c}\langle O, \{tsk(tk, m, l, o, \top, o = \textbf{new } C(\bar{x}); s, \mathcal{L}r) \parallel T\}\rangle \rightsquigarrow \\ \langle O', \{tsk(tk, m, l', o, \top, s, \mathcal{L}r) \parallel T\}\rangle\end{array}}$$

$$(2)\quad \frac{\begin{array}{c}l(o) = o_1 \neq \textbf{null}, l' = l[y \to tk_1], l_1 = buildLocals(\bar{x}, m)), tk_1 \text{is a fresh id}, \\ \mathcal{L}r' = \mathcal{L}r[y{:}\mathtt{T}(tk_n, X)/\star{:}\mathtt{T}(tk_n, X)] \cup \{y{:}\mathtt{T}(tk_1, \mathtt{act})\}\end{array}}{\begin{array}{c}\langle O, \{tsk(tk, m, l, o, \top, y{=}o.m_1(\bar{x}); s, \mathcal{L}r) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tk, m, l', o, \top, s, \mathcal{L}r'), tsk(tk_1, m_1, l_1, o_1, \bot, body(m_1), \emptyset) \parallel T\}\rangle\end{array}}$$

$$(3)\quad \frac{l_1(y) = tk_2, \mathcal{L}r' = \mathcal{L}r[y{:}\mathtt{T}(tk_2, \mathtt{act})/y{:}\mathtt{T}(tk_2, \mathtt{fin})]}{\begin{array}{c}\langle O, \{tsk(tk_1, m_1, l_1, o_1, \top, \textbf{await } y?; s_1, \mathcal{L}r), \\ tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v)) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tk_1, m_1, l_1, o_1, \top, s_1, \mathcal{L}r'), tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v)) \parallel T\}\rangle\end{array}}$$

$$(4)\quad \frac{\begin{array}{c}l_1(y) = tk_2, l_1' = l_1[z \to v], return(tk_1) = tk_3, \\ \mathcal{L}r' = \mathcal{L}r[z{:}\mathtt{T}(tk_n, X)/\star{:}tk_n] \cup \{z{:}\mathtt{T}(tk_3, X)\}, \\ \mathcal{L}r_2' = \mathcal{L}r_2 \backslash \{x{:}\mathtt{T}(tk_3, X)\}\end{array}}{\begin{array}{c}\langle O, \{tsk(tk_1, m_1, l_1, o_1, \top, z{=}y.\textbf{get}; s_1, \mathcal{L}r), \\ tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v), \mathcal{L}r_2) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tk_1, m_1, l_1', o_1, \top, s_1, \mathcal{L}r'), tsk(tk_2, m_2, l_2, o_2, \bot, \epsilon(v), \mathcal{L}r_2') \parallel T\}\rangle\end{array}}$$

$$(5)\quad \frac{obj(o, a, \top) \in O, O' = O[obj(o, a, \top)/obj(o, a, \bot)], v = l(y)}{\begin{array}{c}\langle O, \{tsk(tk, m, l, o, \top, \textbf{return } y, \mathcal{L}r) \parallel T\}\rangle \rightsquigarrow \\ \langle O', \{tsk(tk, m, l, o, \bot, \epsilon(v), \mathcal{L}r) \parallel T\}\rangle\end{array}}$$

$$(6)\quad \frac{\begin{array}{c}(l', s') = eval(instr, O, l) \\ instr \in \{\textbf{skip}, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2, \textbf{while } b \textbf{ do } s_3\}\end{array}}{\begin{array}{c}\langle O, \{tsk(tk, m, l, o, \top, instr; s, \mathcal{L}r) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tk, m, l', o, \top, s', \mathcal{L}r) \parallel T\}\rangle\end{array}}$$

$$(7)\quad \frac{obj(o, a, \bot) \in O, O' = O[obj(o, a, \bot)/obj(o, a, \top)], s \neq \epsilon(v)}{\langle O, \{tsk(tk, m, l, o, \bot, s, \mathcal{L}r) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tk, m, l, o, \top, s, \mathcal{L}r) \parallel T\}\rangle}$$

Figure 7.1: Extended Semantics with local information

# EthIR: A Framework for High-Level Analysis of Ethereum Bytecode

Elvira Albert[1], Pablo Gordillo[1(✉)], Benjamin Livshits[2], Albert Rubio[3], and Ilya Sergey[4]

[1] Universidad Complutense de Madrid, Madrid, Spain
pabgordi@ucm.es
[2] Imperial College London, London, UK
[3] Universitat Politècnica de Catalunya, Barcelona, Spain
[4] University College London, London, UK

**Abstract.** Analyzing Ethereum bytecode, rather than the source code from which it was generated, is a necessity when: (1) the source code is not available (e.g., the blockchain only stores the bytecode), (2) the information to be gathered in the analysis is only visible at the level of bytecode (e.g., gas consumption is specified at the level of EVM instructions), (3) the analysis results may be affected by optimizations performed by the compiler (thus the analysis should be done ideally *after* compilation). This paper presents EthIR, a framework for analyzing Ethereum bytecode, which relies on (an extension of) Oyente, a tool that generates CFGs; EthIR produces from the CFGs, a *rule-based representation* (RBR) of the bytecode that enables the application of (existing) high-level analyses to infer properties of EVM code.

## 1 Introduction

Means of creating distributed consensus have given rise to a family of distributed protocols for building a replicated transaction log (a *blockchain*). These technological advances enabled the creation of decentralised cryptocurrencies, such as Bitcoin [9]. Ethereum [12], one of Bitcoin's most prominent successors, adds Turing-complete stateful computation associated with funds-exchanging transactions—so-called *smart contracts*—to replicated distributed storage.

Smart contracts are small programs stored in a blockchain that can be invoked by transactions initiated by parties involved in the protocol, executing some business logic as automatic and trustworthy mediators. Typical applications of smart contracts involve implementations of multi-party accounting, voting and arbitration mechanisms, auctions, as well as puzzle-solving games

This work was funded partially by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish MINECO projects TIN2015-69175-C4-2-R and TIN2015-69175-C4-3-R, the CM project S2013/ICE-3006 and by the UCM CT27/16-CT28/16 grant. Sergey's research was supported by a generous gift from Google.


S. K. Lahiri and C. Wang (Eds.): ATVA 2018, LNCS 11138, pp. 513–520, 2018.
https://doi.org/10.1007/978-3-030-01090-4_30

with reward distribution. To preserve the global consistency of the blockchain, every transaction involving an interaction with a smart contract is replicated across the system. In Ethereum, replicated execution is implemented by means of a uniform execution back-end—Ethereum Virtual Machine (EVM) [12]—a stack-based operational formalism, enriched with a number of primitives, allowing contracts to call each other, refer to the global blockchain state, initiate sub-transactions, and even create new contract instances dynamically. That is, EVM provides a convenient *compilation target* for multiple high-level programming languages for implementing Ethereum-based smart contracts. In contrast with prior low-level languages for smart contract scripting, EVM features mutable persistent state that can be modified, during a contract's lifetime, by parties interacting with it. Finally, in order to tackle the issue of possible denial-of-service attacks, EVM comes with a notion of *gas*—a cost semantics of virtual machine instructions.

All these features make EVM a very powerful execution formalism, simultaneously making it quite difficult to formally analyse its bytecode for possible inefficiencies and vulnerabilities—a challenge exacerbated by the mission-critical nature of smart contracts, which, after having been deployed, cannot be amended or taken off the blockchain.

**Contributions** In this work, we take a step further towards *sound* and *automated* reasoning about high-level properties of Ethereum smart contracts.

- We do so by providing EthIR, an open-source tool for precise decompilation of EVM bytecode into a high-level representation in a rule-based form; EthIR is available via GitHub: https://github.com/costa-group/ethIR.
- Our representation reconstructs high-level control and data-flow for EVM bytecode from the low-level encoding provided in the CFGs generated by Oyente. It enables application of state-of-the-art analysis tools developed for high-level languages to infer properties of bytecode.
- We showcase this application by conducting an automated resource analysis of existing contracts from the blockchain inferring their loop bounds.

## 2 From EVM to a Rule-based Representation

The purpose of decompilation –as for other bytecode languages (see, *e.g.*, the Soot analysis and optimization framework[11])– is to make explicit in a higher-level representation the *control flow* of the program (by means of rules which indicate the continuation of the execution) and the *data flow* (by means of explicit variables, which represent the data stored in the stack, in contract fields, in local variables, and in the blockchain), so that an analysis or transformation tool can have this control flow information directly available.

### 2.1 Extension of Oyente to Generate the CFG

Given some EVM code, the Oyente tool generates a set of blocks that store the information needed to represent the CFG of such EVM code. However, when the

jump address of a block is not unique (depends on the flow of the program), the blocks generated by Oyente sometimes only store the last value of the jump address. We have modified the structure of Oyente blocks in order to include all possible jump addresses, so that the whole CFG is reconstructed. As an example, Fig. 1 shows the Solidity source code for a fragment of a contract (left), and the CFG generated from it (right). Observe that in the CFGs generated by our extension of Oyente, the instructions SSTORE or SLOAD are annotated with an identifier of the contract field they operate on (for instance, a SSTORE operation that stores a value on the contract field 0 is replaced by SSTORE 0). Similarly, the EVM instructions MSTORE and MLOAD instructions are annotated with the memory address they operate on (such addresses will be transformed into variables in the RBR whenever possible). These annotations cannot be generated when the memory address is not statically known, though, (for instance, when we have an array access inside a loop with a variable index). In such cases, we annotate the corresponding instructions with "?".

```
contract BlockKing {
  ...
  uint public warriorBlock;
  uint public kingBlock;
  ...
  function kingBlock(){
    uint var = kingBlock;
    ...
  }

  function process_payment() {
    uint singleDigit = warriorBlock;
    ...
    while (singleDigit > 10) {
      singleDigit -= 10;
    }
    ...
  }
}
```

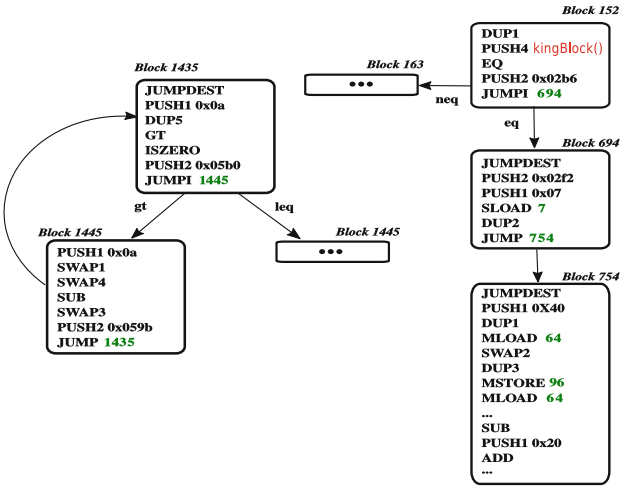



**Fig. 1.** Solidity code (left), and EVM code for `process_payment` within CFG (right).

Finally, when we have Solidity code available, we are able to retrieve the name of the functions invoked from the hash codes (see e.g. Block 152 in which we have annotated in the second bytecode kingBlock, the name of the function to be invoked). This allows us to statically know the continuation block.

## 2.2 From the CFG to Guarded Rules

The translation from EVM into our *rule-based representation* is done by applying the translation in Definition 1 to each block in a CFG. The identifiers given to the rules *–block_x* or *jump_x–* use $x$, the PC of the first bytecode in the block

being translated. We distinguish among three cases: (1) if the last bytecode in the block is an unconditional jump (JUMP), we generate a single rule, with an invocation to the continuation block, (2) if it is a conditional jump (JUMPI) we produce two additional *guarded* rules which represent the continuation when the condition holds, and when it does not, (3) otherwise, we continue the execution in block $x{+}s$ (where $s$ is the size of the EVM bytecodes in the block being translated). As regards the variables, we distinguish the following types:

1. *Stack variables*: a key ingredient of the translation is that the stack is flattened into variables, *i.e.*, the part of the stack that the block is using is represented, when it is translated into a rule, by the explicit variables $s_0, s_1, \ldots$, where $s_1$ is above $s_0$, and so on. The initial stack variables are obtained as parameters $s_0, s_1, \ldots, s_n$ and denoted as $\bar{s}_n$.
2. *Local variables*: the content of the local memory in numeric addresses appearing in the code, which are accessed through MSTORE and MLOAD with the given address, are modelled with variables $l_0, l_1, \ldots, l_r$, denoted as $\bar{l}_r$, and are passed as parameters. For the translation, we assume we are given a map `lmap` which associates a different local variable to every numeric address memory used in the code. When the address is not numeric, we represent it using a fresh variable local to the rule to indicate that we do not have information on this memory location.
3. *Contract fields*: we model fields with variables $g_0, \ldots, g_k$, denoted as $\bar{g}_k$, which are passed as parameters. Since these fields are accessed using SSTORE and SLOAD using the number of the field, we associate $g_i$ to the $i$th field. As for the local memory, if the number of the field is not numeric because it is unknown (annotated as "?"), we use a fresh local variable to represent it.
4. *Blockchain data*: we model this data with variables $\overline{bc}$, which are either indexed with $md_0, \ldots, md_q$ when they represent the message data, or with corresponding names, if they are precise information of the call, like the gas, which is accessed with the opcode GAS, or about the blockchain, like the current block number, which is accessed with the opcode NUMBER. All this data is accessed through dedicated opcodes, which may consume some offsets of the stack and normally place the result on top of the stack (although some of them, like CALLDATACOPY, can store information in the local memory).

The translation uses an auxiliary function $\tau$ to translate each bytecode into corresponding high-level instructions (and updates the size of the stack $m$) and $\tau_G$ to translate the guard of a conditional jump. The grammar of the resulting RBR language into which the EVM is translated is given in Fig. 2. We optionally can keep in the RBR the original bytecode instructions from which the higher-level ones are obtained by simply wrapping them within a nop functor (*e.g.*, nop(DUPN)). This is relevant for a gas analyzer to assign the precise gas consumption to the higher-level instruction in which the bytecode was transformed.

**Definition 1.** *Given a block $B$ with instructions $b_1, \ldots, b_i$ in a CFG starting at PC $x$, and local variables map* `lmap`*, the generated rules are:*

| |
|---|
| *if* $b_i \equiv$ *JUMP* $p$ |
| $block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q) \Rightarrow \tau(b_1, \ldots, b_{i-1}), call(block\_p(\bar{s}_{m-1}, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$ |
| *if* $b_i \equiv$ *JUMPI* $p$ |
| $block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q) \Rightarrow \tau(b_1, \ldots, b_{c-1}), call(jump\_x(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$ |
| $jump\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q) \Rightarrow \tau_G(b_c, \ldots, b_{i-2}) \mid call(block\_p(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$ |
| $jump\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q) \Rightarrow \neg\tau_G(b_c, \ldots, b_{i-2}) \mid call(block\_(x+s)(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$ |
| *if* $b_i \not\equiv$ *JUMP and* $b_i \not\equiv$ *JUMPI* |
| $block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q) \Rightarrow \tau(b_1, \ldots, b_i), call(block\_(x+i)(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$ |

*where functions $\tau$ and $\tau_G$ for some representative bytecodes are:*

| | | |
|---|---|---|
| $\tau$(*JUMPDEST*) | $= \{\};\ m := m$ | |
| $\tau$(*PUSHN* $v$) | $= \{s_{m+1} = v\};\ m := m+1$ | |
| $\tau$(*DUPN*) | $= \{s_{m+1} = s_{m+1-N}\};\ m := m+1$ | |
| $\tau$(*SWAPN*) | $= \{s_{m+1} = s_m, s_m = s_{m-N}, s_{m-N} = s_{m+1}\};\ m := m$ | |
| $\tau$(*ADD*\|*SUB*\|*MUL*\|*DIV*) | $= \{s_{m-1} = s_m + \mid - \mid * \mid / s_{m-1}\};\ m := m-1$ | |
| $\tau$(*SLOAD*\|*MLOAD* $v$) | $= \{s_m = g_v \mid l_{lmap(v)}\};\ m := m$ | *if $v$ is numeric* |
| | $= \{gl \mid ll = s_m, s_m = \mathsf{fresh}()\};\ m := m$ | *otherwise* |
| $\tau$(*SSTORE*\|*MSTORE* $v$) | $= \{g_v \mid l_{lmap(v)} = s_{m-1}\};\ m := m-2$ | *if $v$ is numeric* |
| | $= \{gs_1 \mid ls_1 = s_{m-1}, gs_2 \mid ls_2 = s_m\};\ m := m-2$ | *otherwise* |
| ... | | |
| $\tau_G$(*GT,ISZERO*)\|$\tau_G$(*GT*) | $= leq(s_m, s_{m-1}) \mid gt(s_m, s_{m-1});\ m := m-2$ | |
| $\tau_G$(*EQ,ISZERO*)\|$\tau_G$(*EQ*) | $= neq(s_m, s_{m-1}) \mid eq(s_m, s_{m-1});\ m := m-2$ | |
| ... | | |

| | |
|---|---|
| $RBR$ | $\rightarrow (B \mid J)\ \ RBR \mid \epsilon$ |
| $B$ | $\rightarrow block\_id\ (\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}) \Rightarrow Instr\ \ (Call \mid \epsilon)$ |
| $J$ | $\rightarrow jump\_id\ (\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}) \Rightarrow InstrJ$ |
| $Instr$ | $\rightarrow S\ \ Instr \mid \epsilon$ |
| $S$ | $\rightarrow s = Exp$ |
| $Exp$ | $\rightarrow num \mid x \mid x+y \mid x-y \mid x*y \mid x/y \mid x\%y \mid x^y$ |
| | $\mid and(x,y) \mid or(x,y) \mid xor(x,y) \mid not(x)$ |
| $Call$ | $\rightarrow call(block\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc})) \mid call(jump\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}))$ |
| $InstrJ$ | $\rightarrow Guard$ "\|" $call(block\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}))$ |
| $Guard$ | $\rightarrow eq(x,y) \mid neq(x,y) \mid lt(x,y) \mid leq(x,y) \mid gt(x,y) \mid geq(x,y)$ |

**Fig. 2.** Grammar of the RBR into which the EVM is translated

– *$c$ is the index of the instruction, where the guard of the conditional jump starts. Note that the condition ends at the index $i-2$ and there is always a* PUSH *at $i-1$. Since the pushed address (that we already have in $p$) and the result of the condition are consumed by the* JUMPI*, we do not store them in stack variables.*

- *$m$ represents the size of the stack for the block. Initially we have $m := n$.*
- *variables $gs_1$, $gs_2$ and $gl$, and $ls_1$, $ls_2$ and $ll$, are local to each rule and are used to represent the use of SLOAD and SSTORE, and MLOAD and MSTORE, when the given address is not a concrete number. For SLOAD and MLOAD we also use fresh(), to denote a generator of fresh variables to safely represent the unknown value of the loaded address.*

*Example 1.* As an example, an excerpt of the RBR obtained by translating the three blocks on the right-hand side of Fig. 1 is as follows (selected instructions keep using nop annotations the bytecode from which they have been obtained):

| | | |
|---|---|---|
| $block152(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$<br>$s_1 = s_0$ nop(DUP1)<br>$s_2 = 6584849474$ nop(PUSH4)<br>$call(jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc})$<br>nop(EQ) nop(PUSH2) nop(JUMPI)<br>$jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$<br>$eq(s_2, s_1)$<br>$call(block694(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc})$<br>$jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$<br>$neq(s_2, s_1)$<br>$call(block163(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc})$ | $block694(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$<br>$s_1 = 754$ nop(PUSH2)<br>$s_2 = 7$ nop(PUSH1)<br>$s_2 = g_7$ nop(SLOAD)<br>$s_3 = s_1$ nop(DUP2)<br>$call(block754(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc})$<br>nop(JUMP)<br>$block754(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$<br>$s_3 = 64$ nop(PUSH1)<br>$s_4 = s_3$ nop(DUP1)<br>$s_4 = l_0$ nop(MLOAD) | $s_5 = s_4$<br>$s_4 = s_2$<br>$s_2 = s_5$ nop(SWAP2)<br>$s_5 = s_2$ nop(DUP3)<br>$l_1 = s_4$ nop(MSTORE)<br>$s_3 = l_0$ nop(MLOAD)<br>$\cdots$<br>$s_3 = s_4 - s_3$ nop(SUB)<br>$s_4 = 32$ nop(PUSH1)<br>$s_3 = s_4 + s_3$ nop(ADD)<br>$\cdots$ |

## 3 Case Study: Bounding Loops in EVM using SACO

To illustrate the applicability of our framework, we have analyzed quantitative properties of EVM code by translating it into our intermediate representation and analyzing it with the high-level static analyzer SACO [3]. SACO is able to infer, among other properties, *upper bounds* on the number of iterations of loops. Note that this is the first crucial step to infer the gas consumption of smart contracts, a property of much interest [4]. The internal representation of SACO (described in [2]) matches the grammar in Fig. 2 after minor syntactic translations (that we have solved implementing a simple translator that is available in github, named `saco.py`). As SACO does not have bit-operations (namely `and`, `or`, `xor`, and `not`), our translator replaces such operations by fresh variables so that the analyzer forgets the information on bit variables. After this, for our running example, we prove termination of the 6 loops that it contains and produce a linear bound for those loops. We have included in our github other smart contracts together with the loop bounds inferred by SACO for them. Other high-level analyzers that work on intermediate forms like Integer transition systems or Horn clauses (*e.g.*, AProVE, T2, VeryMax, CoFloCo) could be easily adapted as well to work on our RBR translated programs.

## 4 Related Approaches and Tools

In the past two years, several approaches tackled the challenge of fully formal reasoning about Ethereum contracts implemented directly in EVM bytecode by modeling its rigorous semantics in state-of-the-art proof assistants [5,6]. While those mechanisations enabled formal machine-assisted proofs of various safety and security properties of EVM contracts [5], none of them provided means for fully *automated* sound analysis of EVM bytecode.

Concurrently, several other approaches for ensuring correctness and security of Ethereum contracts took a more aggressive approach, implementing automated toolchains for detecting bugs by symbolically executing EVM bytecode [8,10]. However, low-level EVM representation poses difficulties in applying those tools immediately for analysis of more high-level properties. For instance, representation of EVM in Oyente, a popular tool for analysis of Ethereum smart contracts [1] is too low-level to implement analyses of high-level properties, *e.g.*, loop complexity or commutativity conditions. Zeus, a tool for analysing Ethereum smart contracts via symbolic execution *wrt.* client-provided *policies*, operates directly on Solidity sources [7]. Soundness of Zeus as an analysis approach, thus, depends on the semantics of Solidity, which is not formally defined.

## References

1. Oyente: An Analysis Tool for Smart Contracts (2018). https://github.com/melonproject/oyente
2. Albert, E., et al.: Object-sensitive cost analysis for concurrent objects. STVR **25**(3), 218–271 (2015)
3. Albert, E., et al.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_46
4. Chow, J.: Ethereum, Gas, Fuel & Fees, 2016. Published online on June 23, 2016. https://media.consensys.net/ethereum-gas-fuel-and-fees-3333e17fe1dc
5. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
6. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
7. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: NDSS (2018, to appear)
8. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS, pp. 254–269. ACM (2016)
9. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
10. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. CoRR, abs/ arXiv:1802.06038 (2018)

11. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCON (1999)
12. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)

# SAFEVM: A Safety Verifier for Ethereum Smart Contracts

Elvira Albert, Jesús Correas, Pablo Gordillo
Complutense University of Madrid
Spain

Guillermo Román-Díez
Universidad Politécnica de Madrid
Spain

Albert Rubio
Complutense University of Madrid
Spain

## ABSTRACT

Ethereum smart contracts are public, immutable and distributed and, as such, they are prone to vulnerabilities sourcing from programming mistakes of developers. This paper presents SAFEVM, a verification tool for Ethereum smart contracts that makes use of state-of-the-art verification engines for C programs. SAFEVM takes as input an Ethereum smart contract (provided either in Solidity source code, or in compiled EVM bytecode), optionally with assert and require verification annotations, and produces in the output a report with the verification results. Besides general safety annotations, SAFEVM handles the verification of array accesses: it automatically generates SV-COMP verification assertions such that C verification engines can prove safety of array accesses. Our experimental evaluation has been undertaken on all contracts pulled from etherscan.io (more than 24,000) by using as back-end verifiers CPAchecker, SeaHorn and VeryMax.

## CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Software verification and validation**.

## KEYWORDS

Smart contracts, Ethereum blockchain, Safety verification.

**ACM Reference Format:**
Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338999

## 1 OVERVIEW OF SAFEVM

Each blockchain provides its own programming language to implement smart contracts. Solidity, a Turing complete language, is the most popular language to write smart contracts for the Ethereum platform that are then compiled to EVM (Ethereum Virtual Machine [22]) bytecode. Each instruction executed by the EVM has an associated gas consumption specified by Ethereum. Being security a main concern of Ethereum, the Solidity language contains the verification-oriented functions, `assert` and `require`, to check for safety conditions or requirements and terminate the execution if they are not met. As usual, the `assert` function can be used for verification purposes (e.g., to check invariants), while the `require` function is used to specify preconditions (e.g., to ensure valid conditions on the inputs or contract state variables, or to validate return values from calls to external contracts). When the Solidity code is compiled into EVM bytecode, the `require` condition is transformed into a test that checks the condition and invokes a REVERT bytecode if it does not hold. REVERT aborts the whole execution of the smart contract, reverts the state and all remaining gas is refunded to the caller. The `assert` checks the condition and invokes an INVALID bytecode if it does not hold. When executing INVALID, the state is reverted but no gas is refunded, and hence it has more serious consequences than REVERT: besides the economic consequences of losing the gas, the only information given to the transaction is an out-of-gas error message. The treatment of array accesses is done similarly as for the `assert`, when an array position is accessed, the generated EVM bytecode checks if the position accessed is within the array bounds and otherwise the INVALID bytecode is executed. Division and related bytecodes like MOD, SMOD, ADDMOD, MULMOD, also lead to executing INVALID when the denominator is zero.

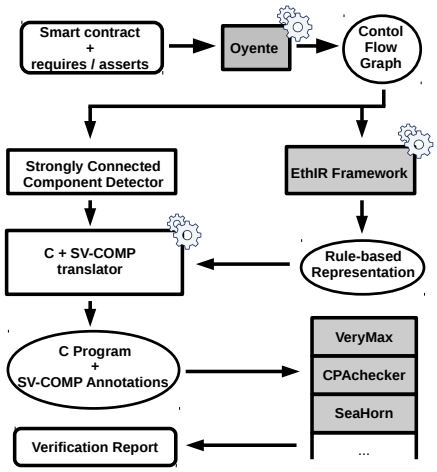



**Figure 1: SAFEVM's architecture**

Therefore, the INVALID bytecodes are key for the verification of the Ethereum smart contracts, as they capture both assertion violations and several sources of fatal operations (e.g., out-of-bounds access, division by zero). In essence, our approach to the verification of smart contracts consists in decompiling the EVM bytecode for the smart contract into a C program with ERROR annotations (following the SV-COMP format, https://sv-comp.sosy-lab.org/2019/rules.php) to enable their verification using existing tools for the verification of C programs. Developing the verifier from the low-level EVM has


*ISSTA '19, July 15–19, 2019, Beijing, China*

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338999

important advantages: (i) sometimes the source code is not available (e.g., the blockchain only stores the bytecode), (ii) the INVALID bytecodes are visible at the level of bytecode and we can give a uniform treatment to the various safety concerns described above, (iii) our analysis works for any other language that compiles to EVM (e.g., Vyper), and it is not affected by changes in the source language, or by compiler optimizations. Luckily, there are a number of open-source tools that help us in the decompilation process and that we have integrated within our tool-chain.

Fig. 1 depicts the main components of SAFEVM that are as follows (shaded boxes are off-the-shelf used systems not developed by us): (1) *Input.* SAFEVM takes a smart contract, optionally with assert and require verification annotations. The smart contract can be given in Solidity source code or in EVM compiled code. In the latter case, the annotations have been compiled into bytecode as described above. (2) *CFG.* In either form, the code is given to Oyente [2], a symbolic execution engine that has been extended to compute the complete CFG from the given smart contract. As Oyente does not handle recursive functions, they are already discarded at this step. The CFG generation phase is not described in the paper, we refer to [2, 3]. (3) *EthIR.* The decompilation of the EVM bytecode into a higher-level *rule-based representation* (RBR) is carried out from the generated CFG by EthIR [3]. Technical details of this phase are not described in the paper, we refer to [3]. (4) *C+SV-COMP translator.* We have implemented a translator for the recursive RBR representation into an *abstract* Integer C program (i.e., all data is of type Integer) with verification annotations using the SV-COMP format. Features of the EVM that we cannot handle yet (e.g., bit-wise operations) are abstracted away in the translation (see Sec. 2). INVALID instructions are transformed into ERROR annotations in the C program following the SV-COMP format. (5) *Verification.* Any verification tool for Integer C programs that uses SV-COMP annotations can be used to verify the safety of our C-translated contracts. We have evaluated our approach using three state-of-the-art C verifiers, CPAchecker [6], VeryMax [9], and SeaHorn [14], and the verification report they produce is processed by us to report the results in terms of functions of the smart contract.

Our tool SAFEVM has a very large (potential) user base, as Ethereum is currently the most advanced platform for coding and processing smart contracts. As we will describe in Sec. 3, using SAFEVM we have automatically verified safety of around 20% of *all* functions (depending on the verifier) that might execute INVALID bytecodes from the whole set of contracts pulled from etherscan.io (more than 24,000 contracts), and we have found potential vulnerabilities in functions that could not be verified.

## 2 TRANSLATION TO C WITH SV-COMP ANNOTATIONS

As motivating example, we use a Solidity contract that implements a lottery system called *SmartBillions* (available at https://smartbillions.com/). We illustrate the safety verification of its internal function commitDividend (an excerpt of its code appears to the left of Fig. 2) that commits remaining dividends to the user wh. We have shortened the variable names by removing the vowels from the names. Lines marked with ⓘ might lead to executing different sources of INVALID: Line 16 (L16 for short) to a division by zero when ttlSpply is 0; at L19 when lst ≥ dvdnds.length and thus

```
1  contract SmartBillions {
2   struct Wallet {
3     ...,    uint16 lstDvdndPrd;}
4   uint public dvdndPrd;
5   uint[] public dvdnds;
6   mapping(address => uint) blncs;
7   uint public ttlSpply ;
8   mapping (address => Wallet) wllts;
9
10  function commitDividend(address wh) {
11  ⊕ //require( ttlSpply > 0);
12  ⊕ //require(dvdndPrd < dvdnds.length);
13    uint lst = wllts[wh].lstDvdndPrd;
14  ⊕ //require(dvdndPrd >= lst );
15    ...
16  ⓘ uint shr=blncs[wh]*0xffffffff/ttlSpply;
17    uint blnc = 0;
18    for (; lst <dvdndPrd;lst++) {
19  ⓘ       blnc += shr * dvdnds[lst];
20    }
21  ⓘ ⊕ assert(lst == dvdndPrd);
22    blnc = (blnc/0xffffffff);
23      ...
24  }
25 }
```

```
block734(s5 ,..., s0,g4,g1,g0,l3,l2) ←
   ...                  // block734 instructions
   call (jump734(s7 ,..., s0,g4,g1,g0,l3,l2))
jump734(s7 ,..., s0,g4,g1,g0,l3,l2) ←
   geq(s7,s6),          // lst ≥ dvdndPrd
   call (block789(s5 ,..., s0,g4,g0,l3,l2))
jump734(s7 ,..., s0,g4,g1,g0,l3,l2) ←
   lt (s7,s6),          // lst < dvdndPrd
   call (block745(s5 ,..., s0,g4,g1,g0,l3,l2))
block745(s5 ,..., s0,g4,g1,g0,l3,l2) ←
   ...                  // block745 instructions
   call (jump745(s9 ,..., s0,g4,g1,g0,l3,l2))
jump745(s9 ,..., s0,g4,g1,g0,l3,l2) ←
   lt (s9,s8),          // lst < dvdnds.length
   call (block759(s7 ,..., s0,g4,g1,g0,l3,l2))
jump745(s9 ,..., s0,g4,g1,g0,l3,l2) ←
   geq(s9,s8),          // lst ≥ dvdnds.length
   call (block758(s7 ,..., s0))
block758(s7 ,..., s0) ←
   INVALID
block759(s7 ,..., s0,g4,g1,g0,l3,l2) ←
   // block759 instructions
   ...
   s6 = s7+s6,          // ADD
   s6 = fresh0,         // SLOAD
   s7 = s4,             // DUP3
   ...
   call (block734(s5 ,..., s0,g4,g1,g0,l3,l2))
```

**Figure 2: Solidity code (left) and excerpt of RBR rules of for loop (lines 18-20)**

it is accessing a position out of the bounds of the array; and at L21 when the condition within the assert does not hold. In order to be able to verify its safety (i.e., absence of INVALID executions), we add the lines marked with ⊕ that introduce error-handling functions require and assert in the verification process.

The starting point of our translator is the RBR produced by EthIR [3]. The RBR is composed of a set of rules containing decompiled versions of bytecode instructions (e.g., LOAD and STORE are decompiled into assignments) and whose structure of rule invocations is obtained from the CFG produced by Oyente. The RBR might contain two kinds of rules: sequences of instructions referred to as *blockX*, and conditional jump rules, named *jumpX*, whose first instruction is the Boolean condition used to select between the rules of the function definition. Rule parameters include: the operand stack flattened in variables named $s_i$, the state of the contract (this is the global data), named $g_i$, and the local memory (represented by local variables), named $l_i$. To the right of Fig. 2 we show the fragment of the RBR produced by EthIR for the loop of L18-L20. At rule *block*759 we show the transformation of some EVM bytecodes (the original bytecodes appear in comments //) into higher-level RBR instructions. The RBR is already *abstract* in the sense that when variables refer to state or memory locations that are not known they become fresh variables (see variable $\texttt{fresh}_0$ in *block*759) so that a posterior analysis will not assume any value for them (details are in [3]). Observe that the fragment of the RBR contains an INVALID instruction within *block*758 and such block can be executed when $geq(s_9, s_8)$ (see rule *jump*745). By tracking variable assignments, we can infer that $s_9$ contains the value of lst and $s_8$ the size of dvdnds, hence the comparison is checking out-of-bounds array access. The remaining of the section explains the main four phases of the translation from the RBR to an abstract Integer C program.

*(1) C functions:* Our translation produces, for each non-recursive rule definition in the RBR, a C function without parameters that returns void. Recursive rules produced by loops are translated into iterative code. For this part of the translation, we compute the SCC

```
26 int g0 = __VERIFIER_nondet_int();
27 ...
28 int g4 = __VERIFIER_nondet_int();
29 int l0 = __VERIFIER_nondet_int();
30 ...
31 int l3 = __VERIFIER_nondet_int();
32 int who = __VERIFIER_nondet_int();
33 int s0;
34 ...
35 int s9;
36
37 void block758() {
38   ERROR: __VERIFIER_error();
39 }
40 void block734() {
41  init_loop_0 :
42    // block734  instructions
43    if(s7 >= s6){ // jump734
44      block789();
45      goto end_loop_0; }
46    // block745  instructions
47    if(s9 >= s8){ // jump745
48      block758();
49      goto end_loop_0; }
50   // block759  instructions
51   s6 = s7 + s6
52   s6 = __VERIFIER_nondet_int
         ();
53   s7 = s4;
54    ...
55   goto init_loop_0 ;
56  end_loop_0:    ;}
```

**Figure 3: C translated code with SV-COMP annotations**

from the CFG (see Fig. 1) and model the detected loops by means of `goto` instructions. Fig. 3 shows the obtained C functions from the RBR program of Fig. 2. Note that *jump* rules are translated into an *if-then-else* structure.

*(2) Types of variables:* Solidity basic, signed and unsigned data types are stored into untyped 256-bit words in the `EVM` bytecode, and the bytecode does not include information about the actual types of the variables. Moreover, most `EVM` operations do not distinguish among them except for few specific signed operations (`SLT`, `SGT`, `SIGNEXTEND`, `SDIV` and `SMOD`). As verifiers behave differently w.r.t. overflow (see details in [6, 9, 14]), our translation allows the user to choose (by means of a flag) if all variables are declared with type `int` in the C program, or of type `unsigned int` with casting to `int` for sign-specific operations. The code in Fig. 3 uses the default `int` transformation. Thus, although in `EVM` integers have overflow, the interpretation of them as unbounded integers or with overflow will be determined by the available options in the C verification tool (e.g., VeryMax only handles unbounded integers). Besides, instructions that contain `fresh` variables or that are not handled (like `SLOAD`) are translated into a call to function `__VERIFIER_nondet_int` in order to model the lack of information for them during verification. Observe that function `block734` includes some operations over the different integer variables. Arrays or maps are not visible in the `EVM` (nor in the RBR). The only information that is trackable about arrays corresponds to their sizes as it is stored in a stack variable that in the C program is stored in an integer variable.

*(3) Variable definitions:* In order to enable reasoning on them (within their scopes) during verification, SAFEVM translates them in the C program as follows: (i) as we flattened the execution stack, we declare the stack variables as global C variables to make them accessible to all C functions. These variables do not need to be initialized as they take values in the program code; (ii) local variables are defined as global C variables (L29-L31) because a function of the contract might be translated into several C-functions, and all of them need to access the local data. They are initialized at the beginning of the function corresponding to the block in which they are firstly used; (iii) state variables are also translated into global variables accessible by all functions and, as their values when functions are verified are unknown, they are initialized using `__VERIFIER_nondet_int` (L26-L28); and (iv) function input parameters are also defined as global variables (for the same reason as (ii)), whose initial values are not determined (L32).

*(4) SV-COMP annotations:* The verification of Ethereum smart contracts is done in SAFEVM by guaranteeing the unreachability of the `INVALID` operations in the C-translated code. Following the `SV-COMP` rules, we translate `INVALID` operations into calls to the `__VERIFIER_error` function so that its unreachability can be proven by any verification tool compatible with the `SV-COMP` annotations. An example of an `INVALID` operation can be seen in L38. Verification tools return that the program in Fig. 2 cannot be verified as the `INVALID` instruction could be executed. This is due to the fact that contract state values are unknown, that is: `ttlSpply` is not guaranteed to be different from 0 at L16 and the size of the array `dvdnds` is not guaranteed to be greater than the value of `lst` at L19. Lines L11 and L12 contain the Solidity instructions needed to guarantee that L16 and L19, respectively, will never execute an `INVALID` instruction. The `assert` at L21 can be verified by using the `require` at L14. The inclusion of the `require` annotation also improves the contract as, if it is violated, a `REVERT` rather than an `INVALID` bytecode will be executed, not causing a loss of gas of the transaction (while the gas needed to check it is negligible).

## 3 EXPERIMENTAL EVALUATION

All components of SAFEVM, except for the C verifiers, are implemented in Python and are open-source. SAFEVM accepts smart contracts written in versions of Solidity up to 0.4.25 and bytecode for the Ethereum Virtual Machine v1.8.18. This section reports the results of our experimental evaluation using SAFEVM with CPAchecker, SeaHorn and VeryMax as verification back-ends. An artifact to try our tool can be downloaded from http://costa.fdi.ucm.es/papers/costa/safevm.ova.

In order to experimentally evaluate SAFEVM, we pulled from etherscan.io all Ethereum contracts whose source code was available on January 2018. This ended up in 10,796 files. From those, we have searched for those files that contain `EVM` code with `INVALID` instructions, in total 7,323. The first phase of SAFEVM that performs the decompilation into the RBR fails for 1,000 files (this 13.65% is larger but quite aligned with the failing rates of other tools e.g. [1, 8]) and reaches a timeout of 60s for 22 files. Thus, our results are on the remaining 6,301 files, that contain 24,294 contracts with 44,046 public functions that can reach an `INVALID` instruction and 177,549 `INVALID`-free functions. We have tested both the translation to type `int` and `unsigned int` for defining C variables, as mentioned in Sec. 2 for those 44,046 functions. We get the following results by using 60s of timeout (Error denotes an error output by the verifier):

| Results | CPAchecker | | VeryMax | | SeaHorn | |
|---|---|---|---|---|---|---|
| | `int` | `uint` | `int` | `uint` | `int` | `uint` |
| Verified | 19.48% | 19.13% | 20.32% | 20.36% | 21.71% | 19.57% |
| Non-Verified | 77.04% | 79.82% | 73.32% | 73.44% | 77.72% | 80.15% |
| Timeout | 3.21% | 0.82% | 6.29% | 6.13% | 0.57% | 0.28% |
| Error | 0.27% | 0.23% | 0.07% | 0.07% | 0% | 0% |

The results for all verifiers are quite aligned, although VeryMax verifies a slightly lower number of functions, and SeaHorn verifies more functions and less reach a timeout. The interpretation made by the tools regarding the Integer semantics (bounded or unbounded) leads to the only relevant difference in the number of functions verified between both translations.

We have manually inspected, out of the 7,323 files, those files whose addresses start with `0x00` and `0x01` in order to understand

the cases that could not be verified. This is a sample of 29 files (243 public functions) that are available at https://github.com/costa-group/EthIR/tree/master/examples/safevm. The manual inspection on the subset gives 54 false alarms (22.2%), namely: 49 functions were verified by CPAchecker; 140 are correct alarms, most of them produced by asserts introduced by the programmers for safety to abort the execution (e.g. 83 come from `Safemath`); 54 are false alarms (many related to `enum` accesses and other imprecisions in the decompilation phase). More in detail, we have identified four types of situations: (1) false alarms due to *inaccuracy of our tool*: some `assert` statements contain non-integer types (e.g., strings, `enum`, etc.) which cannot be verified as we need a more accurate decompilation (see Sec. 4); (2) correct alarms that require *conditional verification*: some `assert` statements can only be verified for concrete contexts, e.g., we found `asserts` to prevent from under/overflow integer arithmetic operations in a widely used library `SafeMath` that can only be verified for given inputs. In the future we plan to integrate conditional verification [9] to infer the preconditions for the `asserts` to hold; (3) Correct alarms detecting *potential vulnerabilities*: we have detected several `INVALID` operations that could represent a vulnerability in the code (e.g., functions that access an array element without checking the boundary) and we have protected them adding `require` statements that enable subsequent verification; and (4) four functions whose verification results depend on the different semantics used for Integers.

As final observations, we notice that `assert` is overused (contradicting the best practices recommendations of Solidity) and that some contracts can be improved by using `require` to avoid the loss of gas when the `assert` statement does not hold. Finally, we argue that although there is much room for improving the accuracy, the results of our experimental evaluation are very encouraging: we have verified safety w.r.t. `INVALID` bytecodes for around 20% of the functions that might reach `INVALID` fully automatically by using state-of-the-art verifiers.

## 4 CONCLUSIONS

Verification of Ethereum smart contracts for potential safety and security vulnerabilities is becoming a popular research topic with numerous tools being developed, among them, we have tools based on symbolic execution [13, 15, 17, 18, 20, 21], tools based on SMT solving [16, 19], and tools based on certified programming [5, 7, 12]. There are some tools also that aim at detecting, analyzing and verifying non-functional properties of smart contracts, e.g., those focused on reasoning about the gas consumption [4, 10, 11, 19].

To the best of our knowledge, SAFEVM is the first tool that uses existing verification engines developed for C programs to verify low-level `EVM` code. This opens the door to the applicability of advanced techniques developed for the verification of C programs to the new languages used to code smart contracts. Although our tool is still in a prototypical stage, it provides a proof-of-concept of the transformational approach, and we argue that it constitutes a promising basis to build verification tools for `EVM` smart contracts. Some of the aspects that we aim at improving in future work is the handling of the data stored in the memory, as it is abstracted away by the EthIR component that SAFEVM is using as soon as there are storage operations on memory. Developing a memory analysis for `EVM` smart contracts can be crucial for the accuracy of verification. We also aim at handling bit-wise operations in the future that are extensively used in the `EVM` bytecode. Advanced reasoning for arrays and maps (the only data structures available in Ethereum smart contracts) can be also added to the framework to gain further accuracy. This requires also further work on the decompilation side. Along the same line, learning information on the types of variables during decompilation will have an impact in the accuracy of the verification process.

## ACKNOWLEDGMENTS

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R and MINECO/FEDER, UE project TIN2015-69175-C4-3-R, by Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM projects S2018/-TCS-4314 and S2018/TCS-4339, co-funded by EIE Funds of the European Union, and by the UCM CT27/16-CT28/16 grant.

## REFERENCES

[1] 2018. Mythril. Available at https://github.com/b-mueller/mythril.
[2] 2018. Oyente: An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente.
[3] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *ATVA (LNCS)*, Vol. 11138. Springer, 513–520.
[4] E. Albert, P. Gordillo, A. Rubio, and I. Sergey. 2018. GASTAP: A Gas Analyzer for Smart Contracts. *CoRR* abs/1811.10403 (2018). arXiv:1811.10403 http://arxiv.org/abs/1811.10403
[5] S. Amani, M. Bégel, M. Bortin, and M. Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.
[6] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 184–190.
[7] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.
[8] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981.
[9] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. 2015. Compositional Safety Verification with Max-SMT. In *FMCAD*. 33–40.
[10] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE Computer Society, 442–446.
[11] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2, OOPSLA (2018), 116:1–116:27.
[12] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269.
[13] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.
[14] T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli. 2015. The SeaHorn Verification Framework. In *CAV*.
[15] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society.
[16] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. *CoRR* abs/1810.11605 (2018). arXiv:1810.11605
[17] J. Krupp and C. Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security Symposium*. USENIX Association, 1317–1333.
[18] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.
[19] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 450–465.
[20] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *ACSAC*. To appear.
[21] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.
[22] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger.


# Running on Fumes

## Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts Using Static Resource Analysis

Elvira Albert[1], Pablo Gordillo[1(✉)], Albert Rubio[1], and Ilya Sergey[2]

[1] Complutense University of Madrid, Madrid, Spain
pabgordi@ucm.es
[2] Yale-NUS College and School of Computing, NUS, Singapore, Singapore

**Abstract.** Gas is a measurement unit of the computational effort that it will take to execute every single operation that takes part in the Ethereum blockchain platform. Each instruction executed by the Ethereum Virtual Machine (EVM) has an associated gas consumption specified by Ethereum. If a transaction exceeds the amount of gas allotted by the user (known as gas limit), an *out-of-gas* exception is raised. There is a wide family of contract vulnerabilities due to *out-of-gas* behaviors. We report on the design and implementation of GASTAP, a Gas-Aware Smart contracT Analysis Platform, which takes as input a smart contract (either in EVM, disassembled EVM, or in Solidity source code) and automatically infers gas upper bounds for all its public functions. Our bounds ensure that if the gas limit paid by the user is higher than our inferred gas bounds, the contract is free of out-of-gas vulnerabilities.

## 1 Introduction

In the Ethereum consensus protocol, every operation on a replicated blockchain state, which can be performed in a transactional manner by executing a *smart contract* code, costs a certain amount of *gas* [29], a monetary value in *Ether*, Ethereum's currency, paid by a transaction-proposing party. Computations (performed by invoking smart contracts) that require *more computational or storage resources*, cost more gas than those that require fewer resources. As regards storage, the EVM has three areas where it can store items: the *storage* is where all *contract state* variables reside, every contract has its own storage and it is persistent between external function calls (transactions) and quite expensive to use; the *memory* is used to hold temporary values, and it is erased between transactions and is cheaper to use; the *stack* is used to carry out operations and it is free to use, but can only hold a limited amount of values.

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R and MINECO/FEDER, UE project TIN2015-69175-C4-3-R, by Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM project S2018/TCS-4314 and by the UCM CT27/16-CT28/16 grant.


P. Ganty and M. Kaâniche (Eds.): VECoS 2019, LNCS 11847, pp. 63–78, 2019.
https://doi.org/10.1007/978-3-030-35092-5_5

The rationale behind the resource-aware smart contract semantics, instrumented with gas consumption, is three-fold. First, paying for gas at the moment of proposing the transaction does not allow the emitter to waste other parties' (aka *miners*) computational power by requiring them to perform a lot of worthless intensive work. Second, gas fees disincentivize users to consume too much of replicated *storage*, which is a valuable resource in a blockchain-based consensus system. Finally, such a semantics puts a cap on the number of computations that a transaction can execute, hence prevents attacks based on non-terminating executions (which could otherwise, *e.g.*, make all miners loop forever).

In general, the gas-aware operational semantics of EVM has introduced novel challenges *wrt.* sound static reasoning about resource consumption, correctness, and security of replicated computations: (1) While the EVM specification [29] provides the precise gas consumption of the low-level operations, most of the smart contracts are written in high-level languages, such as Solidity [13] or Vyper [14]. The translation of the high-level language constructs to the low-level ones makes static estimation of runtime gas bounds challenging (as we will see throughout this paper), and is implemented in an *ad-hoc* way by state-of-the art compilers, which are only able to give constant gas bounds, or return $\infty$ otherwise. (2) As noted in [17], it is discouraged in the Ethereum safety recommendations [16] that the gas consumption of smart contracts depends on the size of the data it stores (i.e., the *contract state*), as well as on the size of its functions inputs, or of the current state of the blockchain. However, according to our experiments, almost 10% of the functions we have analyzed do. The inability to estimate those dependencies, and the lack of analysis tools, leads to design mistakes, which make a contract unsafe to run or prone to exploits. For instance, a contract whose state size exceeds a certain limit, can be made forever *stuck*, not being able to perform any operation within a reasonable gas bound. Those vulnerabilities have been recognized before, but only discovered by means of unsound, pattern-based analysis [17].

In this paper, we address these challenges in a principled way by developing GASTAP, a *Gas-Aware Smart contracT Analysis Platform*, which is, to the best of our knowledge, the first automatic gas analyzer for smart contracts. GASTAP takes as input a smart contract provided in Solidity source code [13], or in low-level (possibly decompiled [26]) EVM code, and automatically infers an upper bound on the gas consumption for each of its public functions. The upper bounds that GASTAP infers are given in terms of the sizes of the input parameters of the functions, the contract state, and/or on the blockchain data that the gas consumption depends upon (e.g., on the *Ether* value).

The inference of gas requires complex transformation and analysis processes on the code that include: (1) construction of the control-flow graphs (CFGs), (2) decompilation from low-level code to a higher-level representation, (3) inference of size relations, (4) generation of gas equations, and (5) solving the equations into closed-form gas bounds. Therefore, building an automatic gas analyzer from EVM code requires a daunting implementation effort that has been possible thanks to the availability of a number of existing open-source tools that we have succeeded to extend and put together in the GASTAP system. In particular, an extension of the tool OYENTE [3] is used for (1), an improved representation of

EthIR [6] is used for (2), an adaptation of the size analyzer of Saco [4] is used to infer the size relations, and the Pubs [5] solver for (5).

The most challenging aspect in the design of Gastap has been the approximation of the EVM gas model (which is formally specified in [29]) that is required to produce the gas equations in step (4). This is because the EVM gas model is highly complex and unconventional. The gas consumption of each instruction has two parts: (i) the *memory gas cost*, if the instruction accesses a location in memory which is beyond the previously accessed locations (known as *active* memory [29]), it pays a gas proportional to the distance of the accessed location. (ii) The second part, the *opcode gas cost*, is related to the bytecode instruction itself. This component is also complex to infer because it is not always a constant amount, it might depend in some cases on the current global and local state.

Gastap has a wide range of applications for contract developers, attackers and owners, including the detection of vulnerabilities, debugging and verification/certification of gas usage. As contract developers and owners, having a precise resource analyzer allows answering the following query about a specific smart contract: "what is the amount of gas necessary to *safely* (i.e., without an out-of-gas exception) reach a certain execution point in the contract code, or to execute a function"? This can be used for debugging, verifying/certifying a safe amount of gas for running, as well as ensuring progress conditions. Besides, Gastap allows us to calculate the safe amount of gas that one should provide to an external data source (e.g., contracts using Oraclize [8]) in order to enable a successful callback. As an attacker, one might estimate, how much *Ether* (in gas), an adversary has to pour into a contract in order to execute the DoS attack. We note that such an attack may, however, be economically impractical.

Finally, we argue that our experimental evaluation shows that Gastap is an effective and efficient tool: we have analyzed more than 29,000 real smart contracts pulled from etherscan.io [2], that in total contain 258,541 public functions, and inferred gas bounds for 91.85% of them in 342.54 h. Gastap can be used from a web interface at https://costa.fdi.ucm.es/gastap.

## 2 Description of Gastap Components

Figure 1 depicts the architecture of Gastap. In order to describe all components of our tool, we use as running example a simplified version (without calls to the external service Oraclize and the authenticity proof verifier) of the `EthereumPot` contract [1] that implements a simple lottery. During a game, players call a method `joinPot` to buy lottery tickets; each player's address is appended to an array `addresses` of current players, and the number of tickets is appended to an array `slots`, both having variable length. After some time has elapsed, anyone can call `rewardWinner` which calls the `Oraclize` service to obtain a random number for the winning ticket. If all goes according to plan, the `Oraclize` service then responds by calling the `__callback` method with this random number and the authenticity proof as arguments. A new instance of the game is then started, and the winner is allowed to withdraw her balance using a `withdraw` method. In Fig. 2, an excerpt of the Solidity code (including the public function `findWinner`)

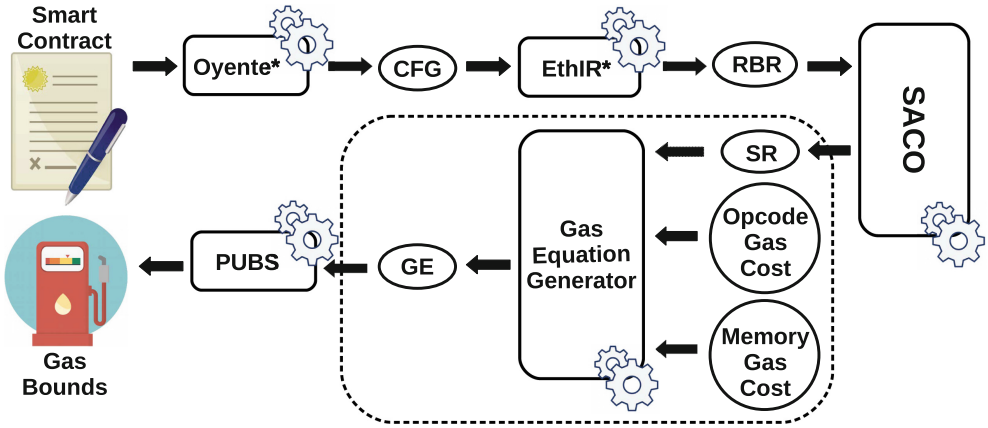



**Fig. 1.** Architecture of GASTAP (CFG: control flow graph; RBR: rule-based representation; SR: size-relations; GE: gas equations)

and a fragment of the EVM code produced by the compiler, are displayed. The Solidity source code is showed for readability, as GASTAP analyzes directly the EVM code (if it receives the source, it first compiles it to obtain the EVM code).

### 2.1 Oyente*: From EVM to a Complete CFG

The first component of our tool, OYENTE*, is an extension of the open-source tool OYENTE [3], a symbolic execution tool developed to analyze Ethereum smart contracts and find potential security bugs. As OYENTE's aim is on symbolic execution rather than on generating a complete CFG, some extensions are needed to this end. The ETHIR framework [6] had already extended OYENTE for two purposes: (1) to recover the list of addresses for unconditional blocks with more than one possible jump address (as OYENTE originally only kept the last processed one), and (2) to add more explicit information to the CFG: jump operations are decorated with the jumping address, discovered by OYENTE, and, other operations like store or load are also decorated with the address they operate: the number of state variable for operations on storage; and the memory location for operations on memory if OYENTE is able to discover it (or with "?" otherwise).

However ETHIR's extension still produced incomplete CFGs. OYENTE* further extends it to handle a more subtle source of incompleteness in the generated CFG that comes directly from the fact that OYENTE is a symbolic execution engine. For symbolic execution, a bound on the number of times a loop is iterated is given. Hence it may easily happen that some (feasible) paths are not reached in the exploration within this bound and they are lost. To solve this problem, we have modified OYENTE to remove the execution bound (as well as other checks that were only used for their particular applications), and have added information to the path under analysis. Namely, every time a new jump is found, we check if the jumping point is already present in the path. In such case, an edge to that point is added and the exploration of the trace is stopped. As a side effect, we not only produce a complete CFG, but also avoid much useless exploration for our purposes which results in important efficiency gain.

```
contract EthereumPot {
 address[] public addresses;
 address public winnerAddress;
 uint[] public slots;
 ...
 function __callback(bytes32 _queryId, string _result, bytes _proof)
   oraclize_randomDS_proofVerify(_queryId, _result, _proof) {
  if(msg.sender != oraclize_cbAddress()) throw;
  random_number = uint(sha3(_result))
  winnerAddress = findWinner(random_number);
  amountWon = this.balance * 98 / 100 ;
  winnerAnnounced(winnerAddress, amountWon);
  if(winnerAddress.send(amountWon)) {
   if(owner.send(this.balance)) {
    openPot();
   }}
  }
 function findWinner(uint random) constant returns(address winner){
  for(uint i = 0; i < slots.length; i++) {
   if(random <= slots[i]) {
    return addresses[i];
   }}
 }
 ...
}
```

```
...
DUP1
PUSH1 => 0x00
SWAP1
POP
PUSH1 => 0x03
DUP1
SLOAD
SWAP1
...
PUSH1 => 0x40
MLOAD
DUP1
SWAP2
SUB
SWAP1
SHA3
PUSH1 => 0x01
...
JUMPDEST
MOD
ADD
PUSH1 => 0x0a
DUP2
SWAP1
SSTORE
POP
PUSH2 => 0x0954
PUSH1 => 0x0a
SLOAD
PUSH2 => 0x064b
JUMP
...
```

**Fig. 2.** Excerpt of Solidity code for `EthereumPot` contract (left), and fragment of EVM code for function `__callback` (right)

When applying Oyente*, our extended/modified version of Oyente, we obtain a *complete* CFG, with the additional annotations already provided by [6].

### 2.2 EthIR*: From CFG to an Annotated Rule-Based Representation

EthIR*, an extension of EthIR [6], is the next component of our analyzer. EthIR provides a rule-based representation (RBR) for the CFG obtained from Oyente*. Intuitively, for each block in the CFG it generates a corresponding rule that contains a high-level representation of all bytecode instructions in the block (e.g., load and store operations are represented as assignments) and that has as parameters an explicit representation of the stack, local, state, and blockchain variables (details of the transformation are in [6]). Conditional branching in the CFG is represented by means of guards in the rules. EthIR* provides three extensions to the original version of EthIR [6]: (1) The first extension is related to the way function calls are handled in the EVM, where instead of an explicit `CALL` opcode, as we have seen before, a call to an internal function is transformed into a `PUSH` of the return address in the stack followed by a `JUMP` to the address where the code of the function starts. If the same function is called from different points of the program, the resulting CFG shares for all these calls the same subgraph (the one representing the code of the function) which ends with different jumping addresses at the end. As described in [17], there is a need to clone parts of the CFG to explicitly link the `PUSH` of the return address with the final `JUMP` to this address. This cloning in our implementation is done at the level of the RBR as follows: Since the jumping addresses are known thanks to the symbolic execution applied by Oyente, we can find the connection between the `PUSH` and the `JUMP` and clone the involved part of the

RBR (between the rule of the `PUSH` and of the `JUMP`) using different rule names for each cloning. (2) The second extension is a flow analysis intended to reduce the number of parameters of the rules of the RBR. This is crucial for efficiency as the number of involved parameters is a bottleneck for the successive analysis steps that we are applying. Basically, before starting the translation phase, we compute the inverse connected component for each block of the CFG, i.e, the set of its predecessor blocks. During the generation of each rule, we identify the local, state or blockchain variables that are used in the body of the rule. Then, these variables have to be passed as arguments only to those rules built from the blocks of its inverse connected component. (3) When we find a store on an unknown memory location "?", we have to "forget" all the memory from that point on, since the writing may affect any memory location, and it is not sound anymore to assume the previous information. In the RBR, we achieve this deletion by assigning fresh variables (thus unknown values) to the memory locations at this point.

Optionally, EthIR provides in the RBR the original bytecode instructions (from which the higher-level ones are obtained) by simply wrapping them within a nop functor (see Fig. 3). Although nop annotations will be ignored by the size analysis, they are needed later to assign a precise gas consumption to every rule.

$$
\begin{array}{l}
block1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) \Rightarrow \\
\quad nop(JUMPDEST), s_{11} = s_9, s_9 = s_{10}, s_{10} = s_{11}, nop(SWAP), s_{11} = 0, nop(PUSH), \\
\quad \underline{l_2 = s_{10}}, nop(MSTORE), s_{10} = 32, nop(PUSH), s_{11} = 0, nop(PUSH), \underline{s_{10} = sha3(s_{11}, s_{10})}, \\
\quad \underline{nop(SHA3)}, s_9 = s_{10} + s_9, nop(ADD), \underline{gl} = s_9, s_9 = fresh_0, nop(SLOAD), s_{10} = s_6, \\
\quad nop(DUP4), call(jump1647(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc})), nop(GT), nop(ISZERO), nop(ISZERO), \\
\quad nop(PUSH), nop(JUMPI)
\end{array}
$$

**Fig. 3.** Selected rule including nop functions needed for gas analysis

*Example 1.* Figure 3 shows the RBR for *block1647*. Bytecode instructions that load or store information are transformed into assignments on the involved variables. For arithmetic operations, operations on bits, sha, etc., the variables they operate on are made explicit. Since stack variables are always consecutive we denote by $\overline{s_n}$ the decreasing sequence of all $s_i$ form $n$ down to 0. $\overline{lv}$ includes $l_2$ and $l_0$, which is the subset of the local variables that are needed in this rule or in further calls (second extension of EthIR*). The unknown location "?" has become a fresh variable $fresh_0$ in *block1647*. For state variables, $\overline{sv}$ includes the needed ones $g_{11}, g_8, g_7, g_6, g_5, g_3, g_2, g_1, g_0$ ($g_i$ is the $i$-th state variable). Finally, $\overline{bc}$ includes the needed blockchain state variables `address`, `balance` and `timestamp`.

### 2.3 SACO: Size Relations for EVM Smart Contracts

In the next step, we generate *size relations* (SR) from the RBR using the Saco tool [4]. SR are equations and inequations that state how the sizes of data change in the rule [12]. This information is obtained by analyzing how each instruction of the rules modifies the sizes of the data it uses, and propagating this information as usual in dataflow analysis. SR are needed to build the gas equations and then generate gas bounds in the last step of the process. The size analysis of Saco has been slightly modified to ignore the *nop* instructions. Besides, before sending

the rules to SACO, we replace the instructions that cannot be handled (e.g., bitwise operations, hashes) by assignments with fresh variables (to represent an unknown value). Apart from this, we are able to adjust our representation to make use of the approach followed by SACO, which is based on abstracting data (structures) to their sizes. For integer variables, the size abstraction corresponds to their value and thus it works directly. However, a language specific aspect of this step is the handling of data structures like array, string or bytes (an array of byte). In the case of array variables, SACO's size analysis works directly as in EVM the slot assigned to the variable contains indeed its length (and the address where the array content starts is obtained with the hash of the slot address).

*Example 2.* Consider the following SR (those in brackets) generated for rule *jump1649* and *block1731*:
$jump1619(\overline{s_{10}}, \overline{sv}, \overline{lv}, \overline{bc}) = block1633(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc})\{s_{10} < s_9\}$
$block1731(\overline{s_8}, \overline{sv}, \overline{lv}, \overline{bc}) = 41 + block1619(s'_8, \overline{s_7}, \overline{sv}, \overline{lv}, \overline{bc})\{s'_8 = 1 + s_8\}$
The size relations for the *jump1619* function involve the `slots` array length ($g_3$ stored in $s_9$) and the local variable `i` (in $s_8$ and copied to $s_{10}$). It corresponds to the guard of the `for` loop in function `findWinner` that compares `i` and `slots.length` and either exits the loop or iterates (and hence consume different amount of gas). The size relation on $s_8$ for *block1731* corresponds to the size increase in the loop counter.

However, for bytes and string it is more challenging, as the way they are stored depends on their actual sizes. Roughly, if they are short (at most 31 bytes long) their data is stored in the same slot together with its length. Otherwise, the slot contains the length (and the address where the string or bytes content starts is obtained like for arrays). Our approach to handle this issue is as follows. In the presence of bytes or string, we can find in the rules of the RBR a particular sequence of instructions (which are always the same) that start pushing the contents of the string or bytes variable in the top of the stack, obtain its length, and leave it stored in the top of the stack (at the same position). Therefore, to avoid losing information, since SACO is abstracting the data structures to their sizes, every time we find this pattern of instructions applied to a string or bytes variable, we just remove them from the RBR (keeping the nops to account for their gas). Importantly, since the top of the stack has indeed the size, under SACO's abstraction it is equal to the string or bytes variable. Being precise, assuming that we have placed the contents of the string or bytes variable in the top of the stack, which is $s_i$, the transformation applied is the following:

$$\boxed{\begin{array}{l} s_{i+1} = 1, nop(PUSH1), s_{i+2} = s_i, nop(DUP2), s_{i+3} = 1, nop(PUSH1), \\ s_{i+2} = and(s_{i+3}, s_{i+2}), nop(AND), s_{i+2} = eq(s_{i+2}, 0), nop(ISZERO), \\ s_{i+3} = 256, nop(PUSH2), s_{i+2} = s_{i+3} * s_{i+2}, nop(MUL), s_{i+1} = s_{i+2} - s_{i+1}, \\ nop(SUB) s_i = and(s_{i+1}, s_i), nop(AND), s_{i+1} = 2, nop(PUSH1), \\ s_{i+2} = s_i, s_i = s_{i+1}, s_{i+1} = s_{i+2}, nop(SWAP1), s_i = s_{i+1}/s_i, nop(DIV) \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{array}{l} nop(PUSH1), nop(DUP2), nop(PUSH1), nop(AND), nop(ISZERO), nop(PUSH2), \\ nop(MUL), nop(SUB), nop(AND), nop(PUSH1), nop(SWAP1), nop(DIV) \end{array}}$$

Since the involved instructions include bit-wise operations among others and, as said, the value of the stack variable becomes unknown, without this transformation the relation between the stack variable and the length of the string or bytes would be lost and, as a result, the tool may fail to provide a bound on the gas consumption. This transformation is applied when possible and, e.g., is needed to infer bounds for the functions `getPlayers` and `getSlots` (see Table 2).

### 2.4 Generation of Equations

In order to generate gas equations (GE), we need to define the EVM gas model, which is obtained by encoding the specification of the gas consumption for each EVM instruction as provided in [29]. The EVM gas model is complex and unconventional, it has two components, one which is related to the memory consumption, and another one that depends on the bytecode executed. The first component is computed separately as will be explained below. In this section we focus on computing the gas attributed to the opcodes. For this purpose, we provide a function $C_{opcode} : s \mapsto g$ which, for an EVM opcode, takes a stack $s$ and returns a gas $g$ associated to it. We distinguish three types of instructions: (1) Most bytecode instructions have a *fixed* constant gas consumption that we encode precisely in the cost model $C_{opcode}$, i.e., $g$ is a constant. (2) Bytecode instructions that have different *constant* gas consumption $g_1$ or $g_2$ depending on some given condition. This is the case of `SSTORE` that costs $g_1 = 20000$ if the storage value is set from zero to non-zero (first assignment), and $g_2 = 5000$ otherwise. But it is also the case for `CALL` and `SELFDESTRUCT`. In these cases we use $g = max(g_1, g_2)$ in $C_{opcode}$. (3) Bytecode instructions with a non-constant (*parametric*) gas consumption that depends on the value of some stack location. For instance, the gas consumption of `EXP` is defined as $10 + 10 \cdot (1 + \lfloor log_{256}(\mu_s[1]) \rfloor)$ if $\mu_s[1] \neq 0$ where $\mu_s[0]$ is the top of the stack. Therefore, we have to define $g$ in $C_{opcode}$ as a parametric function that uses the involved location. Other bytecode instructions with parametric cost are `CALLDATACOPY`, `CODECOPY`, `RETURNDATACOPY`, `CALL`, `SHA3`, `LOG*`, and `EXTCODECOPY`.

Given the RBR annotated with the nop information, the size relations, and the cost model $C_{opcode}$, we can generate GE that define the gas consumption of the corresponding code applying the classical approach to cost analysis [28] which consists of the following basic steps: (i) Each rule is transformed into a corresponding cost equation that defines its cost. Example 2 also displays the GE obtained for the rules *jump1619* and *block1731*. (ii) The nop instructions determine the gas that the rule consumes according to the gas cost model $C_{opcode}$ explained above. (iii) Calls to other rules are replaced by calls to the corresponding cost equations. See for instance the call to *block1619* from rule *block1731* that is transformed into a call to the cost function *block1619* in Example 2. (iv) Size relations are attached to rules to define their applicability conditions and how the sizes of data change when the equation is applied. See for instance the size relations attached to *jump1619* that have been explained in Example 2.

As said before, the gas model includes a cost that comes from the memory consumption which is as follows. Let $C_{mem}(a)$ be the memory cost

function for a given memory slot $a$ and defined as $G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$ where $G_{memory} = 3$. Given an EVM instruction, $\mu_i'$ and $\mu_i$ denote resp. the *highest memory slot* accessed in the local memory, resp., after and before the execution of such instruction. The memory gas cost of every instruction is the difference $C_{mem}(\mu_i') - C_{mem}(\mu_i)$. Besides `MLOAD` or `MSTORE`, instructions like `SHA3` or `CALL`, among others, make use of the local memory, and hence can increase the memory gas cost.

In order to estimate this cost associated to all EVM instructions in the code of the function, we first make the following observations: (1) Computing the sum of all the memory gas cost amounts to computing the memory cost function for the highest memory slot accessed by the instructions of the function under analysis. This is because, as seen, $\mu_i$ and $\mu_i'$ refer to this position in each operation and hence we pay for all the memory up to this point. (2) This is not a standard memory consumption analysis in which one obtains the total amount of memory allocated by the function. Instead, in this case, we infer the actual value of the highest slot accessed by any operation executed in the function.

*Example 3.* Let us show how we obtain the memory gas cost for *block1647*. In this case, the two instructions in this block that cost memory are underlined in Fig. 3 and correspond to a `MSTORE` and `SHA3` bytecodes. In this block, both bytecodes operate on slot 0 of the memory, and they cost 3 units of gas because they only activate up to slot 1 of the memory.

### 2.5 PUBS Solver: From Equations to Closed-Form Bounds

The last step of the gas bounds inference is the generation of a *closed-form gas upper bound*, i.e., a solution for the GE as a non-recursive expression. As the GE we have generated have the standard form of cost relations systems, they can be solved using off-the-shelf solvers, such as PUBS [5] or COFLOCO [15], without requiring any modification. These systems are able to find polynomial, logarithmic and exponential solutions for cost relations in a fully automatic way. The gas bounds computed for all public functions of EthereumPot using PUBS can be found in Table 1, note that they are parametric on different state variables, input and blockchain data.

## 3 Experimental Evaluation

This section presents the results of our evaluation of GASTAP. In Sect. 3.1, we evaluate the accuracy of the gas bounds inferred by GASTAP on the EthereumPot by comparing them with the bounds computed by the Solidity compiler.

In Sect. 3.2, we evaluate the efficiency and effectiveness of our tool by analyzing more than 29,000 Ethereum smart contracts. To obtain these contracts, we pulled from etherscan.io [2] all Ethereum contracts whose source code was available on January 2018. GASTAP is available at https://costa.fdi.ucm.es/gastap.

### 3.1 Gas Bounds for EthereumPot Case Study

Table 1 shows in column **solc** the gas bound provided by the Solidity compiler **solc** [13], and in the next two columns the bounds produced by GASTAP for opcode gas and memory gas, respectively, for all public functions in the contract. If we add the gas and memory bounds, it can be observed that, for those functions with constant gas consumption, we are as accurate as **solc**. Hence, we do not lose precision due to the use of static analysis.

For those 6 functions that **solc** fails to infer constant gas consumption, it returns $\infty$. For opcode gas, we are able to infer precise *parametric* bounds for five of them, `rewardWinner` is linear on the size of the first and third state variables ($g1$ and $g3$ represent resp. the sizes of the arrays `addresses` and `slots` in Fig. 2), `getSlots` and `findWinner` on the third, `getPlayers` on the first, and `__callback` besides depends on the value of `result` (second function parameter) and `proof` (last parameter). It is important to note that, although the Solidity source code of some functions (*e.g.*, of `getSlots` and `getPlayers`) does not contain loops, they are generated by the compiler and are only visible at the EVM level. This also happens, for example, when a function takes a *string* or *bytes* variable as argument. This shows the need of developing the gas analyzer at the EVM level.

For `joinPot` we cannot ensure that the gas consumption is finite without embedding information about the blockchain in the analyzer. This is because `joinPot` has a loop: `for (uint i = msg.value; i >= minBetSize; i-= minBetSize) {tickets++;}`, where `minBetSize` is a state variable that is initialized in the definition line as **uint** `minBetSize = 0.01ether`, and `ether` is the value of the *Ether* at the time of executing the instruction. This code has indeed several problems. The first one is that the initialization of the state variable `minBetSize` to the value `0.01ether` does not appear in the EVM code available in the blockchain. This is because this instruction is executed only once when the contract is created. So our analyzer cannot find this instruction and the value of `minBetSize` is unknown (and hence no bound can be found). Besides, the loop indeed does not terminate if `minBetSize` in not strictly greater than zero (which could indeed happen if `ether` would take zero or a negative value). If we add the initialization instruction, and embed in the analyzer the invariant that `ether`$> 0$ (hence `minBetSize` becomes $> 0$), then we are able to infer a bound for `joinPot`.

For `__callback` we guarantee that the memory gas is *finite* but we cannot obtain an upper bound for it, GASTAP yields a *maximization error* which is a consequence of the information loss due to the soundness requirement described in extension 3 of Sect. 2.2. Intuitively, maximization errors may occur when the analyzer needs to compose the cost of the different fragments of the code. For the composition, it needs to maximize (*i.e.*, find the maximal value) the cost of inner components in their calling contexts (see [5] for details). If the maximization process involves memory locations that have been "forgotten" by ETHIR* (variables "?"), the upper bound cannot be inferred. Still, if there is no ranking function error, we know that all loops terminate, thus the memory gas consumption is finite.

**Table 1.** Gas bounds for `EthereumPot`. Function `nat` defined as `nat(l)=max(0,l)`.

| function | solc | opcode bound GASTAP | memory bound GASTAP |
|---|---|---|---|
| `totalBet` | 790 | 775 | 15 |
| `locked` | 706 | 691 | 15 |
| `getEndTime` | 534 | 519 | 15 |
| `slots` | 837 | 822 | 15 |
| `rewardWinner` | $\infty$ | 80391+5057·nat(g3)+5057·nat(g1) | 18 |
| `Kill` | 30883 | 30874 | 9 |
| `amountWon` | 438 | 423 | 15 |
| `getPlayers` | $\infty$ | 1373+292·nat(g1-1/32)<br>+75·nat(g1+31/32) | 6·nat(g1)+24+$\left\lfloor \frac{(6 \cdot nat(g1)+24)^2}{512} \right\rfloor$ |
| `getSlots` | $\infty$ | 1507+250·nat(g3-1/32)<br>+75·nat(g3+31/32) | 6·nat(g3)+24+$\left\lfloor \frac{(6 \cdot nat(g3)+24)^2}{512} \right\rfloor$ |
| `winnerAddress` | 750 | 735 | 15 |
| `__callback` | $\infty$ | 229380+3·(nat(proof)/32)<br>+103·nat(result/32)<br>+50·nat((32-nat(result)))<br>+5836·nat(g3)+5057·nat(g1) | max_error |
| `owner` | 662 | 647 | 15 |
| `endTime` | 460 | 445 | 15 |
| `potTime` | 746 | 731 | 15 |
| `potSize` | 570 | 555 | 15 |
| `joinPot` | $\infty$ | no_rf | 9 |
| `addresses` | 1116 | 1101 | 15 |
| `findWinner` | $\infty$ | 1555+779·nat(g3) | 15 |
| `random_number` | 548 | 533 | 15 |

Finally, this transaction is called always with a constant gas limit of 400,000. This contrasts with the non-constant gas bound obtained using GASTAP. Note that if the gas spent (without including the *refunds*) goes beyond the gas limit the transaction ends with an out-of-gas exception. Since the size of $g3$ and $g1$ is the same as the number of players, from our bound, we can conclude that from 16 players on the contract is in risk of running out-of-gas and get stuck as the 400,000 gas limit cannot be changed. So using GASTAP we can prevent an out-of-gas vulnerability: the contract should not allow more than 15 players, or the gas limit must be increased from that number on.

### 3.2 Statistics for Analyzed Contracts

Our experimental setup consists on 29,061 contracts taken from the blockchain as follows. We pulled all Ethereum contracts from the blockchain as of January 2018, and removed duplicates. This ended up in 10,796 files (each file often contains several contracts). We have excluded the files where the decompilation phase fails in any of the contracts it includes, since in that case we do not get any information on the whole file. This failure is due to OYENTE in 1,230 files, which represents a 11.39% of the total and to ETHIR in 829 files, which represents

a 7.67% of the total. The failures of EthIR are mainly due to the cloning mechanism in involved CFGs for which we fail to find the relation between the jump instruction and the return address.

After removing these files, our experimental evaluation has been carried out on the remaining 8,737 files, containing 29,061 contracts. In total we have analyzed 258,541 public functions (and all auxiliary functions that are used from them). Experiments have been performed on an Intel Core i7-7700T at 2.9 GHz x 8 and 7.7 GB of Memory, running Ubuntu 16.04. Gastap accepts smart contracts written in versions of Solidity up to 0.4.25 or bytecode for the Ethereum Virtual Machine v1.8.18. The statistics that we have obtained in number of functions are summarized in Table 2, and the time taken by the analyzer in Table 3. The results for the opcode and memory gas consumption are presented separately.

**Table 2.** Statistics of gas usage on the analyzed 29,061 smart contracts from Ethereum blockchain

| Type of result | #opc | %opc | #mem | %mem |
|---|---|---|---|---|
| Constant gas bound | 223,294 | 86.37% | 225,860 | 87.36% |
| Parametric gas bound | 14,167 | 5.48% | 13,312 | 5.15% |
| Time out | 13,140 | 5.08% | 13,539 | 5.24% |
| Finite gas bound (maximization error) | 7,095 | 2.74% | 5,830 | 2.25% |
| Termination unknown (ranking function error) | 716 | 0.28% | 0 | 0% |
| Complex control flow (cover point error) | 129 | 0.05% | 0 | 0% |
| Total number of functions | 258,541 | 100% | 258,541 | 100% |

Let us first discuss the results in Table 2 which aim at showing the effectiveness of Gastap. Columns **#opc** and **#mem** contain number of analyzed functions for opcode and memory gas, resp., and columns preceded by **%** the percentage they represent. For the analyzed contracts, we can see that a large number of functions, 86.37% (resp. 87.36%), have a constant opcode (resp. memory) gas consumption. This is as expected because of the nature of smart contracts, as well as because of the Ethereum safety recommendations mentioned in Sect. 1. Still, there is a relevant number of functions 5.48% (resp. 5.15%) for which we obtain an opcode (resp. memory) gas bound that is not constant (and hence are potentially vulnerable). Additionally, 5.08% of the analyzed functions for opcodes and 5.24% for memory reach the timeout (set to 1 min) due to the further complexity of solving the equations.

As the number of analyzed contracts is very large, a manual inspection of all of them is not possible. Having inspected many of them and, thanks to the information provided by the Pubs solver used by Gastap, we are able to classify the types of errors that have led to a "*don't-know*" answer and which in turn explain the sources of incompleteness by our analysis: (i) *Maximization error*: In many cases, a *maximization error* is a consequence of loss of information by the size analysis or by the decompilation when the values of memory locations are lost. As mentioned, even if we do not produce the gas formula, we know

that the gas consumption is *finite* (otherwise the system flags a ranking function error described below). (ii) *Ranking function error:* The solver needs to find ranking functions to bound the maximum number of iterations of all loops the analyzed code might perform. If GASTAP fails at this step, it outputs a *ranking function error*. Sect. 3 has described a scenario where we have stumbled across this kind of error. We note that number of these failures for **mem** is lower than for **opcode** because when the cost accumulated in a loop is 0, PUBS does not look for a ranking function. (iii) *Cover point error:* The equations are transformed into direct recursive form to be solved [5]. If the transformation is not feasible, a *cover point error* is thrown. This might happen when we have mutually recursive functions, but it also happens for nested loops as in non-structured languages. This is because they contain jump instructions from the inner loop to the outer, and vice versa, and become mutually recursive. A loop extraction transformation would solve this problem, and we leave its implementation for the future work.

**Table 3.** Timing breakdown for GASTAP on the analyzed 29,061 smart contracts

| Phase | $T_{opcode}$ (s) | $T_{mem}$ (s) | $T_{total}$ (s) | %opc | %mem | %total |
|---|---|---|---|---|---|---|
| CFG generation (OYENTE*) | — | — | 17,075.55 | — | — | 1.384% |
| RBR generation (ETHIR*) | — | — | 81.37 | — | — | 0.006% |
| Size analysis (SACO) | — | — | 105,732 | — | — | 8.57% |
| Generation of gas equations | 141,576 | 125,760 | 267,336 | 11.48% | 10.2% | 21.68% |
| Solving gas equation (PUBS) | 395,429 | 447,502 | 842,931 | 32.06% | 36.3% | 68.36% |
| Total time GASTAP | | | 1,233,155.92 | | | 100% |

As regards the efficiency of GASTAP, the total analysis time for all functions is 1,233,155.92 s (342.54 h). Columns **T** and **%** show, resp., the time in seconds for each phase and the percentage of the total for each type of gas bound. The first three rows are common for the inference of the opcode and memory bounds, while equation generation and solving is separated for opcode and memory. Most of the time is spent in solving the GE (68.36%), which includes some timeouts. The time taken by ETHIR is negligible, as it is a syntactic transformation process, while all other parts require semantic reasoning. All in all, we argue that the statistics from our experimental evaluation show the accuracy, effectiveness and efficiency of our tool. Also, the sources of incompleteness point out directions for further improvements of the tool.

## 4 Related Work and Conclusions

Analysis of Ethereum smart contracts for possible safety violations and security and vulnerabilities is a popular topic that has received a lot of attention

recently, with numerous tools developed, leveraging techniques based on symbolic execution [19,20,22,23,25,27], SMT solving [21,24], and certified programming [7,9,18], with only a small fraction of them focusing on analyzing gas consumption.

The GASPER tool identifies gas-costly programming patterns [11], which can be optimized to consume less. For doing so, it relies on matching specific control-flow patterns, SMT solvers and symbolic computation, which makes their analysis neither sound, nor complete. In a similar vein, the recent work by Grech *et al.* [17] identifies a number of classes of gas-focused vulnerabilities, and provides MadMax, a static analysis, also working on a decompiled EVM bytecode, data-combining techniques from flow analysis together with CFA context-sensitive analysis and modeling of memory layout. In its techniques, MadMax differs from Gastap, as it focuses on identifying control- and data-flow patterns inherent for the gas-related vulnerabilities, thus, working as a bug-finder, rather than complexity analyzer. Since deriving accurate worst-case complexity boundaries is not a goal of any of both GASPER and MadMax, they are unsuitable for tackling the challenge 1, which we have posed in the introduction.

In a concurrent work, Marescotti *et al.* identified three cases in which computing gas consumption can help in making Ethereum more efficient: (a) prevent errors causing contracts get stuck with *out-of-gas* exception, (b) place the right price on the gas unit, and (c) recognize semantically-equivalent smart contracts [24]. They propose a methodology, based on the notion of the so-called *gas consumption paths* (GCPs) to estimate the worst-case gas consumption using techniques from symbolic bounded model checking [10]. Their approach is based on symbolically enumerating all execution paths and unwinding loops to a limit. Instead, using resource analysis, Gastap infers the maximal number of iterations for loops and generates accurate gas bounds which are valid for any possible execution of the function and not only for the unwound paths. Besides, the approach by Marescotti *et al.* has not been implemented in the context of EVM and has not been evaluated on real-world smart contracts as ours.

*Conclusions.* Automated static reasoning about resource consumption is critical for developing safe and secure blockchain-based replicated computations, managing billions of dollars worth of virtual currency. In this work, we employed state-of-the art techniques in resource analysis, showing that such reasoning is feasible for Ethereum, where it can be used at scale not only for detecting vulnerabilities, but also for verification/certification of existing smart contracts.

## References

1. The EthereumPot contract (2017). https://etherscan.io/address/0x5a13caa82851342e14cd2ad0257707cddb8a31b7
2. Etherscan (2018). https://etherscan.io
3. Oyente: An Analysis Tool for Smart Contracts (2018). https://github.com/melonproject/oyente

4. Albert, E., et al.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_46
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
6. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: a framework for high-level analysis of ethereum bytecode. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 513–520. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_30
7. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: CPP 2018, pp. 66–77. ACM (2018)
8. Bernani, T.: Oraclize (2016). http://www.oraclize.it
9. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: PLAS 2016, pp. 91–96. ACM (2016)
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
11. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: SANER 2017, pp. 442–446. IEEE Computer Society (2017)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of aprogram. In: POPL 1978, pp. 84–96 (1978)
13. Ethereum. Solidity (2018). https://solidity.readthedocs.io
14. Ethereum. Vyper (2018). https://vyper.readthedocs.io
15. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 275–295. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_15
16. Ethereum Foundation. Safety - Ethereum Wiki (2018). https://github.com/ethereum/wiki/wiki/Safety. Accessed on 14 Nov 2018
17. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. In: PACMPL, 2(OOPSLA), pp. 116:1–116:27 (2018)
18. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
19. Grossman, S., et al.: Online detection of effectively callback free objects with applications to smart contracts. In: PACMPL, 2(POPL), pp. 48:1–48:28 (2018)
20. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: NDSS 2018. The Internet Society (2018)
21. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting The Laws of Order in Smart Contracts. CoRR, abs/1810.11605 (2018)
22. Krupp, J., Rossow, C.: Teether: Gnawing at ethereum to automatically exploit smart contracts. In: USENIX Security Symposium, pp. 1317–1333. USENIX Association (2018)
23. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In CCS 2016, pp. 254–269. ACM (2016)

24. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 450–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_33
25. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC 2018, pp. 653–663. ACM (2018)
26. Suiche, M.: Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode (2017)
27. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: CCS 2018, pp. 67–82. ACM (2018)
28. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
29. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014)