Object Capability Patterns: Policies and Specifications

Shu-Peng Loh
Imperial College London
shu.loh16@imperial.ac.uk

Sophia Drossopoulou
Imperial College London
s.drossopoulou@imperial.ac.uk

Abstract

We propose a set of first-order logic predicates to formally specify object-to-object interactions which can then be used to describe a set of policies in an object-oriented computational model. Using these predicates, we attempt to formally specify the policies of well-established Object-Capability (OCap) patterns within the OCap literature which we have implemented in the capability-safe language Pony. We also offer some preliminary insights on how such specifications can be used in the context of a non-Ocap model by describing the security properties of a pattern built on the Ethereum smart contract programming language Solidity, which we argue implements a form of stack-based access control.

1 Introduction

Recent widespread adoption of distributed ledger technology (blockchain) has created multiple decentralized, distributed computational platforms where millions of dollars are transacted over codified constructs called smart contracts[1]. The power of distributed modern computing hence lies in facilitating cooperation between multiple agents, but it comes with risk as an agent is vulnerable to *unexpected* outcomes[2] from participating in these smart contracts. This might generally arise from two issues:

- oversight or misconception of the outcomes of executing a piece of *known* code
- failure to defend against malicious execution of *unknown* code components

These two issue are often closely intertwined in any system of execution that has both trusted and untrusted code components (the second issue is often a result of the first).
In recent years the Object-Capability (OCap) model has received increasing attention as a compelling approach to building robust, distributed systems that promote what Miller[2] calls *cooperation without vulnerability*. The OCap model attemps to address these two issues by alleviating security as a separate concern from the mind of the programmer, by leveraging the object-oriented programming paradigm and imposing certain prohibitions.

[1]For example, as of 10 Aug 2017, Ethereum is a US$28 billion blockchain platform with an in-built Turing-complete programming language that can be used to create and deploy such contracts.

[2]Representing in general any outcome arising from a piece of code execution that has deviated from an agent's original intention or objective independent from code.

2 OCap Model

The OCap model uses the reference graph of the objects as the access graph, and strictly requires objects to interact with each other only by sending messages on object references[3].

2.1 From Capability to Object-Capability

2.2 OCap Languages

- Joe-E (inspired by Java)
- Emily (inspired by OCaml)
- Caja (inspired by JavaScript)
- E
- Pony

2.2.1 Language Restrictions

2.3 OCap Patterns

An OCap pattern is a concrete representation of the OCap model and comprises a set of objects

connected to each other by capabilities. Objects interact with each other by sending messages on capabilities. An OCap pattern may be visualised as a directed graph—nodes represent objects, and each edge from an object o to another o' represents o holding a capability that allows it to directly access o'.

3 Formal specifications

3.1 Definitions

Here, we borrow liberally the definitions of runtime state, module and arising configurations from the appendix of [1].

Runtime state: σ consists of a stack frame φ, and a heap χ. A stack frame is a mapping from receiver (this) to its address, and from the local variables (VarId) and parameters (ParId) to their values. Values are integers, the booleans true or false, addresses, or null. The heap maps addresses to objects. Objects are tuples consisting of the class of the object, and a mapping from field identifiers onto values.

$\sigma \in$ state = frame $\times$ heap
$\varphi \in$ frame = StackId $\rightarrow$ val
$\chi \in$ heap = addr $\rightarrow$ object
v $\in$ val = {`null`, `true`, `false`} $\cup$ *addr* $\cup$ $\mathbb{N}$
StackId = {*this*} $\cup$ VarId $\cup$ ParId
object = ClassId $\times$(FieldId $\rightarrow$ val)

Module:

M $\in$ Module = ClassId $\cup$ SpecId
$\rightarrow$
(ClassDescr $\cup$ Specification)

Reach and Execution:

Arising Configurations

Domination:

3.1.1 MayAccess Definitions

$\forall$M,σ,x,e. M,$\sigma \vDash$ MayAccess^{Dir,Now}(x,e) $\iff$
$\exists$f. $\lfloor$x.f$\rfloor_\sigma = \lfloor$e$\rfloor_\sigma \vee$
(σ(this) = $\lfloor$x$\rfloor_\sigma \wedge \exists$y. σ(y) = $\lfloor$e$\rfloor_\sigma$)

$\forall$M,σ,x,e. M,$\sigma \vDash$ MayAccess^{Dir,Eve}(x,e) $\iff$
$\exists\sigma' \in$Arising(M,σ).
M,$\sigma' \vDash$ MayAccess^{Dir,Now}(x,e)

$\forall$M,σ,x,e. M,$\sigma \vDash$ MayAccess^{Ind,Now}(x,e) $\iff$
$\exists\bar{\text{f}}$. $\lfloor$x.$\bar{\text{f}}\rfloor_\sigma = \lfloor$e$\rfloor_\sigma \vee$
(σ(this) = $\lfloor$x$\rfloor_\sigma \wedge \exists$y. σ(y.$\bar{\text{f}}$) = $\lfloor$e$\rfloor_\sigma$)

$\forall$M,σ,x,e. M,$\sigma \vDash$ MayAccess^{Ind,Eve}(x,e) $\iff$
$\exists\sigma' \in$Arising(M,σ).
M,$\sigma' \vDash$ MayAccess^{Ind,Now}(x,e)

A note on f and $\bar{f}$: While f can considered as an object field, it can also represent an object method that returns a val. Similarly $\bar{\text{f}}$ can be considered a series of fields, or methods that return a val, or a combination of both.

Note that we define MayAccess^{Ind,*}(x,e) to mean a form of weak access—that a path exists from x to some well-formed e, but it does *not* imply that such a path is navigateable by x. Take the following example where at a particular state, x1.next points to x2, and x2.next points to x3. MayAccess^{Ind,*}(x1,x3) holds, regardless of whether x2.next is navigateable by x1. If x2.next is a private method that can only be called internally by x2, the predicate still holds—we say that a path from x1 to x3 exists, but is not navigateable by x1.

On the other hand, we define MayAccess^{Dir,*}(x,e) to mean a form of strong access—it implies that a patch exists from x0 to e and that such a path *is* navigateable. This is because the reference to e exists within x0's state and can be used by x0. Notice how this differs from the previous example, where x1 cannot guarantee that it can execute x2.next, because it is possible that x2.next is protected by x2 through encapsulation and data-hiding.

3.1.2 MayAffect Definitions

$\forall$M,σ,x,e. M,$\sigma \vDash$ MayAffect^{Now}(x,e) $\iff$
$\exists\bar{\text{m}},\bar{\text{a}},\sigma'$. x.$\bar{\text{m}}(\bar{\text{a}}) \rightsquigarrow \sigma' \wedge \lfloor$e$\rfloor_\sigma \neq \lfloor$e$\rfloor_{\sigma'}$

$\forall$M,σ,x,e. M,$\sigma \vDash \text{MayAffect}^{Eve}(x,e) \iff$
$\exists\sigma\in$Arising(M,σ).
$\exists\bar{m},\bar{a},\sigma'$. $x.\bar{m}(\bar{a}) \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{\sigma} \neq \lfloor e \rfloor_{\sigma'}$

If e is an object:
$\forall\sigma,\sigma'$, e$\in$Object. $\lfloor e \rfloor_{\sigma} \neq \lfloor e \rfloor_{\sigma'} \iff$
$\exists$f. $\lfloor e.f \rfloor_{\sigma} \neq \lfloor e.f \rfloor_{\sigma'}$

3.2 Implications

3.2.1 Imposing OCap rules

We assume $\forall${o,o'}$\in$Object.

Table 1: *MayAccess(o,o') Relations in OCap*

	Now		**Eventually**
Direct	$\text{MayAccess}^{Dir,Now}$	$\Longrightarrow$ $\not\Longleftarrow$	$\text{MayAccess}^{Dir,Eve}$
	$\Downarrow \Uparrow$		$\Downarrow \Uparrow$
Indirect	$\text{MayAccess}^{Ind,Now}$	$\Longrightarrow$ $\not\Longleftarrow$	$\text{MayAccess}^{Ind,Eve}$

Table 2: *MayAffect(o,o') Relations in OCap*

MayAffect^{Now}	$\Longrightarrow$ $\not\Longleftarrow$	MayAffect^{Eve}

Results from Contraposition:
$\neg\text{MayAccess}^{Dir,Eve}$(o,o') $\Longrightarrow$
$\neg\text{MayAccess}^{Dir,Now}$(o,o')
$\neg\text{MayAccess}^{Ind,Eve}$(o,o') $\Longrightarrow$
$\neg\text{MayAccess}^{Ind,Now}$(o,o')
$\neg\text{MayAccess}^{Ind,Now}$(o,o') $\Longrightarrow$
$\neg\text{MayAccess}^{Dir,Now}$(o,o')
$\neg\text{MayAccess}^{Ind,Eve}$(o,o') $\Longrightarrow$
$\neg\text{MayAccess}^{Dir,Eve}$(o,o')

Rule: Objects can only interact with each other through sending messages on capabilities. Therefore, if an object o can affect o', then there must be a path from o to o':

$$\text{M},\sigma \vDash \text{MayAffect}^{Eve}(\text{o,o}') \implies \text{MayAccess}^{Ind,Eve}(\text{o,o}') \quad (1)$$

By contraposition,

$$\text{M},\sigma \vDash \neg\text{MayAccess}^{Ind,Eve}(\text{o,o}') \implies \neg\text{MayAffect}^{Eve}(\text{o,o}') \quad (2)$$

Here, we immediately see a first defensive outcome of the OCap model. Isolation of object o' from o guarantees that the state of object o' cannot be modified by object o.

Rule: Objects cannot forge capabilities, and only connectivity begets connectivity. Therefore, if an object o' exists in a state σ and there are no possible paths from o to o' in that state, then it is not possible in any arising state σ' from σ where there is a path from o to o':

$$\text{M},\sigma \vDash \neg\text{MayAccess}^{Ind,Now}(\text{o,o}') \wedge \text{Exists}(\text{o}')_{\sigma} \implies \neg\text{MayAccess}^{Ind,Eve}(\text{o,o}') \quad (3)$$

We require the enforcement of o' to exist in σ because o' can refer to an object that does not exist in σ but will be created and exist in some arising σ' from σ. From *Equation 3* and *Equation 2*, to protect o' from o in all states arising from a state σ, we only need to ensure isolation of o' at σ :

$$\text{M},\sigma \vDash \neg\text{MayAccess}^{Ind,Now}(\text{o,o}') \wedge \text{Exists}(\text{o}')_{\sigma} \implies \neg\text{MayAffect}^{Eve}(o, o') \quad (4)$$

So far these results only serve as a base case, because the protection of an object's state requires the impractical configuration that objects are completely isolated from each other on the reference graph. Cooperation between objects imply that there needs to be some path established between the objects for interaction to take place and therefore the predicate $\neg\text{MayAccess}^{Ind,Now}$(o,o') rarely holds in a system of cooperation.

The power of OCap patterns hence lies in providing concrete examples of a system of cooperation that allows the existence of paths between objects for cooperation while still dictating the degree of control of one object can have over another. For example, we can have variations of following configurations in an Ocap pattern:

- $\text{M},\sigma \vDash \text{MayAccess}^{Ind,Now}(\text{o,o}') \wedge \text{Exists}(\text{o}')_{\sigma} \implies \neg\text{MayAffect}^{Eve}(o, o')$
- $\text{M},\sigma \vDash \text{MayAccess}^{Ind,Eve}(\text{o,o}') \implies \neg\text{MayAffect}^{Eve}(o, o')$

Notice how we have also also focused on the predicate MayAffect^{Eve} rather than MayAffect^{Now}. Security concerns are often a question of whether what one object can do to another object's in any eventual state of a system. If we are concerned with the protection of o' from o, it is not very useful to have a policy where $\neg\text{MayAffect}^{Now}$(o,o') holds while

$\neg\text{MayAffect}^{Eve}$(o,o') does not. Furthermore, by ensuring $\neg\text{MayAffect}^{Eve}$ holds, we can also ensure $\neg\text{MayAffect}^{Now}$ holds since, by contraposition, $\neg\text{MayAffect}^{Eve} \implies \neg\text{MayAffect}^{Now}$.

3.3 Pattern 1: The JavaScript DOM Tree

We define the following variables throughout our pattern:

- o,o' $\in$Object
- Node,ReNode $\subseteq$Object
- n,n' $\in$Node
- rn,rn' $\in$ReNode

Vulnerability of a Node: The vulnerability of a node lies in the fact that its fields and methods are defined as public. Therefore, any object that has a path and can navigate a path to a node may change the state of a node.

$$\forall \text{o, n. } \text{MayAccess}^{Dir,Eve}\text{(o,n)} \implies \text{MayAffect}^{Eve}\text{(o,n)}$$

Policy 1: Necessary Condition

$$\forall \text{n,o, RN} \subseteq \text{ReNode.}\ \text{Dom(RN,n)} \wedge \text{MayAffect}^{Eve}\text{(o,n)} \implies \exists \text{rn} \in \text{RN. MayAccess}^{Ind,Eve}\text{(o,rn)} \wedge \text{MayAccess}^{Dir,Eve}\text{(rn,n)}$$

3.4 Pattern 2: Redell's Caretaker

3.5 Pattern 3: Membrane

Policy 1:

References

[1] Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Reasoning about risk and trust in an open word. 2015.

[2] Mark S Miller. Robust composition: Towards a unified approach to access control and concurrency control. 2006.

[3] Mark S Miller and Jonathan S Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Annual Asian Computing Science Conference*, pages 224–242. Springer, 2003.