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 higher-order predicate logic to formally specify object-to-object interactions which can then be used to describe reference dynamics 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).

[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.

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[7] 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.

2 OCap Model

2.1 From Object to Capability

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[8]. When these references cannot be forged in the system, and when there there are no default globally accessible objects (for an object to be globally accessible, the reference to use that object has to be *explicitly* granted to every object globally), then it becomes necessary that one needs to obtain references to objects in order to call the associated methods of the object. Object references therefore embody authority, and we use the word capability to describe the encapsulation of both reference (designation) and

authority which are unforgeable.

2.2 From Capability to Object-Capability

Origins of capabilities date back to Dennis and Van Horn[2]'supervisor as a mechanism of protecting low-level resources like memory segments. Early attempts to implement the capability model include the MIT PDP-1 computer and the CAL-TSS system. The capability model was then extended by the early operating system designers of HYDRA[12] into a protected ability to invoke arbitrary services provided by other processes. Capabilities were later implemented on computer and operating systems such as CAP[11], KeyKOS[5], and EROS[9]. More recent work includes the Capsicum kernel framework that provides capabilities support for UNIX[10], and Google's open-source capabilities-based operating system Fuchsia OS[6]. The development of capabilities and a thorough examination of the object-capability model with the capability-safe programming language E, can be found in Miller's PhD thesis[7].

2.3 OCap Languages

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

2.3.1 Language Restrictions

2.4 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

We borrow liberally the definitions of runtime state, module and arising configurations from the appendix of [4].

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.

[Runtime state] $\sigma \in \text{state} = \text{frame} \times \text{heap}$
$\varphi \in \text{frame} = \text{StackId} \to \text{val}$
$\chi \in \text{heap} = \text{addr} \to \text{object}$
$\text{v} \in \text{val} = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \mathit{addr} \cup \mathbb{N}$
$\text{StackId} = \{\mathit{this}\} \cup \text{VarId} \cup \text{ParId}$
$\text{object} = \text{ClassId} \times (\text{FieldId} \to \text{val})$

Module:

$$\text{M} \in \text{Module} = \text{ClassId} \cup \text{SpecId} \to (\text{ClassDescr} \cup \text{Specification})$$

Execution:

Arising State Configurations

3.1.1 MayAccess Definitions

We define in total four flavours of *MayAccess* predicates that are inspired by the work in Drossopoulou et al.[4], but our four MayAccess predicates extend and modify the definition in [4] to more precisely capture the different configurations of access in terms of mode and time:

- Mode: directly (*Dir*) or indirectly (*Ind*)
- Time: now (*Now*) or eventually (*Eve*)

and are broad enough to describe both non-OCap and OCap models:

*Four MayAccess Definitions

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

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

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

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

where, we say $\text{MayAccess}^{Dir,Now}(x,e)$ holds *iff* in some current state σ, we have a field in x that points to e, or that we have x as a receiver in σ and there exists some y in the same σ that points to e. We also say $\text{MayAccess}^{Ind,Now}(x,e)$ holds *iff* in some current state σ, we have a *series of fields* in x that leads to e, or that we have x as a receiver in σ and there exists some y with a series of fields in the same σ that leads to e. Lastly, the 'eventual' definitions of the two predicates are then defined as there existing an arising configuration σ' from σ, where the 'now' definitions hold at σ'.

Note that $\text{MayAccess}^{*,Now}$ holds imply $\text{MayAccess}^{*,Eve}$ holds, since an eventual state σ' that arises from σ can refer to σ, and therefore, the 'now' definitions implies and are stronger than the 'eventual' definitions. Also since a series of fields $\bar{f}$ can mean a singular field f, $\text{MayAccess}^{Dir,*}$ holds imply $\text{MayAccess}^{Ind,*}$ holds, and therefore the 'direct' definitions are stronger than the 'indirect' definitions. We surmarise the relationships between the four flavours of MayAccess in *Table 1*.

Note that without imposing any further assumptions (such as those from the OCap model), we have defined *both* $\text{MayAccess}^{Dir,Now}(x,e)$ and $\text{MayAccess}^{Ind,Now}(x,e)$ to mean forms of very weak access—that a directed path exists from x to e, and that we do *not* imply that such a path is *traverseable* by x. By a path being traverseable from x to e, we mean that x has a method that upon invocation, can lead to a method invocaton in e, and therefore x can interact with e, and possibly, modify e. These MayAccess definitions by themselves represent mere paths, or mere *possibilities* of object interaction—they do *not* imply that interaction would succeed. We therefore need another predicate describes method invocation between objects.

Table 1: *MayAccess Relations*

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

3.1.2 MayAffect Definitions

With our access predicates in place, we define a *MayAffect* predicate that describe what it means for x to be able to affect e:

*MayAffect Definition

$M,\sigma \vDash \text{MayAffect}(x,e) \iff$
$\sigma(this) = \lfloor x \rfloor_\sigma \wedge \exists \sigma' \in \text{Arising}(M,\sigma)$
$\wedge\ \sigma'(this) = \lfloor e \rfloor_{\sigma'}$

where we say MayAffect(x,e) holds *iff* we have x as a receiver in some state σ, and in some arising state σ' from σ, we have e as a receiver in σ'. This means that there must exist some method in x that upon invocation, will result in some state σ' where e is the receiver in the state σ'. Other possible definitions for MayAffect and why we have settled for the definition above arguing about method invocation, can be found in the Appendix (*section 4*).

3.2 Object-Oriented Paradigm

As our paper focuses on security between trusted and untrusted objects, we shift our use of these predicates to an object-oriented paradigm. So, what do our above definitions mean in an object-oriented world?

*Object as Receiver Nec. Cond.
$M,\sigma \vDash \sigma(\text{this}) = \lfloor o_x \rfloor_\sigma \implies$
$\exists \sigma^*, m, \bar{a}.\ M,\sigma^*, o_x.m(\bar{a}) \rightsquigarrow \sigma$

We first make clear that for an object to be a receiver in a particular state σ, we require a method to exist in the object in a prior state σ* that upon invocation can lead to state σ. This follows from our definition of execution. Let us illustrate this with an example where there is a particular state σ of the system with a configuration of 3 objects o_1, o_2, and o_3. In this example, to illustrate object encapsulation, we assume all fields in objects are private. o_1.next is a private field that holds the object reference of o_2, and o_2.next is a private field that holds the object reference of o_3. In this example:

- $M,\sigma \vDash \text{MayAccess}^{Ind,Now}(o_1,o_3)$ holds true, regardless of whether we know o_1 can invoke any methods on o_3.
- If o_1 contains no method that can invoke o_2, or o_2 has no methods that can be invoked externally, then according to our definitions, we say $M,\sigma \vDash \neg\text{MayAffect}(o_1,o_2)$, meaning that o_1 cannot affect o_2.
- If o_1 contains a method that can invoke o_2, and o_2 contains some public method that can be invoked externally which is empty and does nothing, but o_2 contains no method that can invoke o_3, then according to our definitions, we mean $M,\sigma \vDash \neg\text{MayAffect}(o_2,o_3)$, since we know o_2 does not have any way of invoking o_3. We also mean $M,\sigma \vDash \text{MayAffect}(o_1,o_2)$ to hold, because given this configuration, we know o_1 has a method that can invoke o_2 and, that o_2 has a public method that can be invoked, even if it does nothing.

Note that if we assume all fields are private, for an object o to *eventually modify* an object o', it requires:

- there exist a method in o that can eventually invoke a method in o'
- there eventually exist a method in o' that can be invoked externally to modify its own internal state

Our MayAffect(o,o') predicate however, is not concerned whether or not the state o' is modified —the fact that a method in o' can be eventually invoked by o is good enough grounds for us to mean that o can affect o'. Our justification is that in the object-oriented paradigm, programmers almost always enforce security of a sensitive field in a protected object to be declared private, and define some public method in the object that can modify the concerned private field based on some condition:

*Private Field Definition
$M,\sigma \vDash \text{PrivField}(o,f) \iff$
$[\exists \sigma' \in \text{Arising}(M,\sigma) \wedge \lfloor f \rfloor_{\sigma'} \neq \lfloor f \rfloor_\sigma \implies$
$\exists m, \bar{a}.\ M,\sigma, o.m(\bar{a}) \rightsquigarrow \sigma']$

Hence, the question of whether an untrusted object can modify a private field in a protected object can often be expressed as a question of whether the untrusted object can invoke a method in the protected object. Furthermore, given our definitions of PrivField and MayAffect, we can derive a lemma that states that a necessary condition for modification of a private field in o_x is the existence of some arbitrary object o* that can affect o_x, where it is also possible that o* can refer to o_x. Proving the falsity of our MayAffect predicate is therefore sufficient to prove that all private fields in o_x cannot be modified:

*Private Field Modification Lemma
$M,\sigma \vDash \forall \sigma', f.\ [\sigma' \in \text{Arising}(M,\sigma) \wedge \lfloor f \rfloor_{\sigma'} \neq \lfloor f \rfloor_\sigma$
$\wedge\ \text{PrivField}(o,f) \implies$
$\exists o^*.\ M,\sigma \vDash \text{MayAffect}(o^*,o)$

Therefore, our MayAffect(o,o') requires:

- there exist a method in o_x ($\sigma(\text{this}) = \lfloor o \rfloor_\sigma$) that can eventually invoke a method in o' ($\exists \sigma' \in \text{Arising}(M,\sigma) \wedge \sigma'(\text{this}) = \lfloor o' \rfloor_{\sigma'}$)
- which by collorary means there eventually exist a method in o' that can be invoked externally $\sigma'(\text{this}) = \lfloor o' \rfloor_{\sigma'}$

Note, that in some languages like JavaScript, methods can be added to objects dynamically at run-time, such that a method of an object that does not exist in some current state can potentially exist in some eventual later state. However, if we are operating in a language that does not allow dynamically adding of methods to objects, or that the protected object o_y

is defined such that methods cannot be added dynamically to it, proving there does not exist a public method now in o_y is sufficient to prove that there will not eventually exist a public method in o_y:

*No dynamic adding of methods to objects.
$M,\sigma \vDash \forall o,\sigma',m,\bar{a}_{\sigma},\bar{a}_{\sigma'}.$
$[\ \sigma' \in Arising(M,\sigma) \land \sigma'(o.m(\bar{a}_{\sigma'}))$
$\implies \sigma(o.m(\bar{a}_{\sigma}))\ ]$

3.2.1 Domination

Here, we introduce the concept of domination that is inspired by the work on ownership types by Clarke et al.[1], where they define that the owner o of an object o' dominates o' in the object graph such that for any path p from some arbitrary object o'' to o', either o'' belongs to the set of owners or that p passes through o. Similarly, we say that a set S dominates x *iff* for every object X* in the system that has a path to x, X* must either have a path to some arbitrary object y that is a member of the set S, or X* belongs to the set S.

*Domination Definition (DOM)
$M,\sigma \vDash Dom(S,x) \iff$
$\forall X^*.\ [MayAccess^{Ind,Now}(X^*,x) \implies$
$\exists y \in S.\ MayAccess^{Ind,Now}(X^*,y) \lor X^* \in S]$

3.2.2 Corollaries

*Path-Execution Condition (PEC)
$M,\sigma \vDash MayAffect(o,o') \implies$
$MayAccess^{Ind,Eve}(o,o')$

*Global Path Chain (GPC)
$M,\sigma \vDash MayAccess^{Ind,Now}(o,o')$
$\iff$
$\exists X^*.\ [MayAccess^{Ind,Now}(o,X^*) \land$
$MayAccess^{Dir,Now}(X^*,o')]$

*Global Path Chain II (GPC II)
$M,\sigma \vDash MayAccess^{Ind,Now}(o,o') \land Dom(S,o')$
$\implies$
$\exists X^* \in S.\ [MayAccess^{Ind,Now}(o,X^*) \land$
$MayAccess^{Dir,Now}(X^*,o')]$

*Global Execution Chain (GEC)
$M,\sigma \vDash MayAffect(o,o')$
$\iff$
$\exists X^*.\ [MayAffect(o,X^*) \land$
$MayAffect(X^*,o')]$

*Global Execution Chain II (GEC II)
$M,\sigma \vDash MayAffect(o,o') \land Dom(S,o')$
$\implies$
$MayAccess^{Ind,Eve}(o,o') \land$
$\exists x \in S.\ [MayAffect(o,x) \land$
$MayAffect(x,o')]$

3.2.3 Lemmas

*Object Existence Lemma (OEL)
if z exists in σ,
$\vDash M,\sigma \vDash MayAccess^{Ind,Now}(z,z)$ is true $\land$
$\vDash M,\sigma \vDash MayAccess^{Dir,Now}(z,z)$ is true $\land$
$\vDash M,\sigma \vDash MayAffect(z,z)$ is true

3.3 OCap Specifications

3.3.1 What is protection?

In an object-oriented world, security concerns between objects are often a question of whether what one object can do to another object in *any* eventual state of a system. Because an object encapsulates both state and behaviour, strictly speaking, security of an object should govern over both the integrity of the object's fields (object state) and whether the objects' methods can be called (object behaviour). Our *MayAffect* predicate enables a discussion of protection of behaviour[3] and because we assume that pro-

[3] Our MayAffect is too strong to reason specifically which behaviours can be called. This is however not a big issue in *pure* OCap systems, where often giving away the capability of an object typically means allowing *all* public behaviours of the object to be called without distinction.

tected objects should have fields that are privately encapsulated, we only need to focus our reasoning on object to object method invokations as a security concern. With these simplifications, protection for us then becomes solely a matter of whether we can allow or deny an object to invoke a method of another object. To help us reason about protection, we first formalize our assumptions and the rules of OCap using our predicates in *subsection 3.4* which will help guide us in constructing our necessary conditions for MayAffect later in **??**.

3.4 Formal specifications in the OCap Model

3.4.1 Passing object references

Objects cannot forge capabilities, and only connectivity begets connectivity.

---

*OCap Introduction Lemma (OCI)

$$M,\sigma \vDash \neg \text{MayAccess}^{Dir,Now}(o,o') \wedge \text{MayAccess}^{Dir,Eve}(o,o') \implies \exists X^*.\ \text{MayAccess}^{Dir,Now}(X^*,o') \wedge \text{MayAffect}(X^*,o)$$

The OCap introduction lemma states that in given a state σ, if o does not have the capability o′ but o may eventually obtain the capability of o′, then there must exist an object X* that has the capability of o′ at σ, and that X* can affect o, so that X* can introduce the capability of o′ to o.

Note that this lemma does not hold in non-OCap systems, if object capability or references can be forged. That is, even though object o might not have the capability of o′ in σ, the object o may contain a capability forging method that can forge and return the capability of o′, such that after execution of the method to some state σ′, o would have the capability of o′ in σ′, without involving any other object.

---

*OCap Eventual Path Connect Lemma (EPC)

$$M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(o,o') \implies \exists X^*.\ [(\text{MayAccess}^{Dir,Now}(o,X^*) \vee \text{MayAccess}^{Dir,Now}(X^*,o)) \wedge \text{MayAccess}^{Ind,Eve}(X^*,o')]$$

EPC has the meaning that if an object o has an eventual path to o′ in some arising state σ′, then there must exist an object X* that has an eventual path to o′, and there must exist a way for o to have a path to X* in σ′. Object o either must already have a path to X* in σ, or that it must be able to receive the capability of X* through introduction by X* in σ. EPC1 is actually a formal representation of connectivity begets connectivity across time:

- **Initial Conditions or Endowment:** o has the capability of o′ in σ, therefore X* refers to o′
- **Parenthood:** if o can create o′ in some arising σ′, then o can also create o′ in σ, therefore X* refers to o′
- **Introduction:** o will only obtain the path to o′ in some arising σ′ through another object introducing o a path to o′, therefore X* refers to an object that is not the same object as o ($X^* \neq o$).

Note that for o to have an eventual path to o′, we only require o to have a a direct capability to some arbitrary object X* or that X* has a direct capability of o so that X* can introduce itself to o. This is because we have stated that X* will have an eventual path to o′. If o already has the capability of X* in σ then we know o can reach X* in σ′ by definition. If not, the capability of X* must be introduced to o. For X* to introduce itself, X* must have the capability of o in σ.

Well, what if the capability of X* is introduced by some *other* object $\tilde{X}^*$? Note that in such a case, $\tilde{X}^*$ must have the capability of object X* (for $\tilde{X}^*$ to even introduce X* to o in the first place), and therefore $\tilde{X}^*$ will also be able to eventually have a path to o′. Also $\tilde{X}^*$ must also be able to introduce itself to o. There is hence no logical difference between $\tilde{X}^*$ and X* in our

formal description and $\tilde{X}$* might simply be referred to as X*.

There is one final critical result from EPC. Notice how, there is a 'recursive' $\text{MayAccess}^{Ind,Eve}$(X*,o') in our condition for $\text{MayAccess}^{Ind,Eve}$(o,o'). This allows us to recursively expand the condition to incorporate *all* X* intermediate objects in the path leading to o'. Repeated recursive expansions will eventually give us conditions that are only defined in $\text{MayAccess}^{Dir,Now}$ predicates, where the terminating $\text{MayAccess}^{Ind,Eve}$(o',o') can be determined to be true or false based on whether o' exists, according to OEL. This result allows us to define an eventual path between two objects based purely on a present configuration of objects on the reference graph in σ.

In the example, we make concrete EPC using only 3 objects o, x, o'. Let us assume o, x, and o' always exists, such that the terminating recursive predicates would return true. This example then illustrates an eventual path from o to o' can only exist if one of these present 6 configurations in σ holds:

- If X*=x, then o has the capability of x, and x has an eventual path to o'—
  by x having the direct capability of o'
- If X*=x, then o has the capability of x, and x has an eventual path to o'—
  by o' having the direct capability of x
- If X*=x, then x has the capability of o, and x has an eventual path to o'—
  by x having the direct capability of o'
- If X*=x, then x has the capability of o, and x has an eventual path to o'—
  by o' having the direct capability of x
- If X*=o', then o has the capability of o'
- If X*=o', then o' has the capability of o

Note how, these 6 configurations are direct capability configurations at σ, but allows us to reason about whether a potential path from o to o' can exist in some arising state σ' from σ.

Lastly, from the OCI lemma, we know that for an object o to introduce a capability to another object o', we require that o can affect o'. Therefore, EPC can be modified:

---

*OCAP EPC LEMMA II (EPC II)

$$\begin{aligned} &M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(o,o') \\ &\quad\Longrightarrow \\ &\quad\exists X^*.\ [(\text{MayAccess}^{Dir,Now}(o,X^*) \\ &\qquad\vee \\ &\qquad\text{MayAffect}(X^*,o)) \\ &\qquad\wedge \\ &\qquad\text{MayAccess}^{Ind,Eve}(X^*,o')] \end{aligned}$$

3.5 Protection and Cooperation

It is good to know we can enforce protection through some present path configurations that can guarantee eventual path-isolation, but in practice this is still not entirely useful. The quintessential question in OCap systems is whether we can enforce protection such as controlling whether o may affect o', on *some condition*. In our framework, this just means that cooperation between objects first require paths to exist, but *how* objects cooperate are capture in the object method definitions and availability.

So, let us now assume we have code knowledge of some objects in our system so we know how they behave. If we have a simple object o' that has *only* private methods, then we know:

$$M,\sigma \vDash \forall X^*.\ \text{MayAffect}(X^*,o') \implies \lfloor X^* \rfloor_\sigma = \lfloor o' \rfloor_\sigma$$

Even though in theory any object may possess the capability of o', as long as we know there is no way for anybody but o' to *exercise* the capability of o', then we know that for all objects X* that are not o', X* has no way of affecting of o' Recall, in our definition of MayAffect, the requirement that for an object o to affect o', we require a method to exist in o that can invoke o', and a method in o' that can be invoked externally. Consequently, with the knowledge that o' has only private methods, we also know o' has no way of invoking any other objects but itself:

$$M,\sigma \vDash \forall X^*.\ \text{MayAffect}(o',X^*) \implies \lfloor X^* \rfloor_\sigma = \lfloor o' \rfloor_\sigma$$

The two outcomes from knowing an object with

only private behaviours are:

- $\vDash [M,\sigma \vDash \forall X^* \neq o'.\neg \text{MayAffect}(X^*,o')]$
- $\vDash [M,\sigma \vDash \forall X^* \neq o'.\neg \text{MayAffect}(o',X^*)]$

This means that methods in objects play a dual important role of what objects can do to others, and what others can do to the object.
Let us see how we further develop this in our framework with another example.

We have some starting configuration in a system where there is some object o which we have code knowledge of, and that o has the capability of o'. Note that we do not need code knowledge of o' to enforce a policy of how o might want to interact with o'. Since o has the capabilitiy of o', we say o has the power to enforce the conditions on when and how it will *exercise* o'. To put it in another way, we can now allow an outgoing path to exist from o to o', based on some condition *written in o* on how the path to o' can be *used*.

If o has only *one* public method that will use the capability of o' based on the condition that the boolean field `access` within o is evaluated to true at the time of the method call, then we have:

$$M,\sigma \vDash \forall X^*.\ \text{MayAffect}(o,X^*) \implies \lfloor X^* \rfloor_\sigma = \lfloor o' \rfloor_\sigma \wedge \exists \sigma'.\ [\sigma' \in \text{Arising}(M,\sigma) \wedge \lfloor o.\texttt{allow} \rfloor_{\sigma'} = \lfloor \texttt{true} \rfloor_{\sigma'}]$$

$$M,\sigma \vDash \forall Y^*.\ \text{MayAffect}(Y^*,o) \implies \exists \sigma'.\ [\sigma' \in \text{Arising}(M,\sigma) \wedge \lfloor o.\texttt{allow} \rfloor_{\sigma'} = \lfloor \texttt{true} \rfloor_{\sigma'}]$$

The logics we have developed show that security of an object can be enforced in an OCap model by:

- being careful about our present path configurations so that we can ensure eventual path-isolation between the object and untrusted object
- specifying security policies within objects that dictate how incoming messages and outgoing messages are handled

Therein lies also the difference between OCap and non-OCap models, because non-OCap models that allow forging of capabilities or a global ambient authoriy *cannot* enforce security through pure path isolation, which is a distinctive feature of OCap models.

'Internal' object protection can be done through using defensive programming techniques like encapsulation and checking of conditions for methods to succeed. In a way these forms of protection can be considered a form of 'stack-based' access control from the called object's perspective, because they only happen *after* the calling object has already called the object. The Ethereum blockchain platform uses prominently such a form of 'stack-based' access control.

In the OCap pattern examples we will go through in the next section, protection of an object is typically accomplished through some attenuating object that the protected object knows will never leak its capability directly such that the attenuating objects dominate the trusted object (the trusted object can ensure this if it creates those attenuating objects), and that protection is enforced through some security policies *within* those attenuating objects.

3.6 Pattern 1: The JavaScript DOM Tree

We use a JavaScript Document Object Model (DOM) Tree pattern largely inspired by the example in Devriese et al.[3] where they use a Kripke worlds framework to reason about the pattern but here instead, we will use our prediate logic framework that we developed in the previous section. With the DOM, JavaScript can access and change all the elements of an HTML document. Typically, a website will have some kind of external third-party advertisement object, where it is given a particular node (we call it an Ad Node) within the DOM Tree, to render in the website. Additionally, we have the Document root node which all other nodes descend from, and hence these nodes form a kind of access hierachy, where higher in the tree are more security-sensitive and powerful. We therefore need a way to protect the other nodes in the DOM Tree from the external third party advertisement object. Can we protect the other nodes convincingly in an OCap language? We illutrate this by first explaining why giving away a direct capability to a node is dangerous. We next construct an attenuating restricted node object called ReNode with some configuration that is meant to address the vulnerabilities of the node it is meant to protect. This ReNode can prevent access to the immediate parent of the node it is protecting, or allow access to a specified number of levels in the tree. Once configured, it can then be safely given away to an external third-party, if we can prove convincingly that the other nodes in the DOM tree are protected. We then use our predicate logic framework to prove why, in an OCap system, we can be assured that giving away a ReNode is safe.

We begin by stating the variables we will be using, where Object is the most general object in our system which supersets Node and ReNode. We only have code knowledge of Node and ReNode, and the question is if giving away a ReNode of a specific configuration to any arbitrary unknown object that we do not have code knowledge of, is safe.

- o,o′ ∈UntrustedObject
- n,n′ ∈Node
- rn,rn′ ∈ReNode

---

*Node Vulnerability

$$M,\sigma \vDash \forall o,n,n'.\ \text{MayAccess}^{Dir,Now}(o,n) \implies \text{MayAffect}^{Now}(o,n) \wedge \text{MayAccess}^{Dir,Now}(o,n')$$

The vulnerability of a node lies in the fact that it contains one public method `setProperty(key,value)` that can be called by anyone holding its capability which will modify the properties of the node (by creating or modifying a key-value pair in a map within the node). A node also has a public field `parent` that will return the capability of its parent node in the DOM Tree. Consequently, this allows an object which has the direct capability of any one node in the tree to navigate up to the root node (Document), and consequently navigate to all other nodes in the tree and modify them.

The question is therefore how do we prove that by giving away an attenuating object 'ReNode' that restricts access to a Node, that the recipient of the ReNode is adhering to the security policies of the system. For example, if an unknown object has the capability of a ReNode that allows the unknown object to traverse 2 depth levels upwards in the hierachy of the DOM Tree to modify a node at that level, how can we be sure that the unknown object has no way to traverse more than just 2 levels in the Tree and modify nodes above that level? We illustrate this OCap pattern using an example, following which we show the specifications and prove protection of nodes using our predicate framework.

Obj->Node Weakest Condition

$M,\sigma \vDash \forall o,rn,n.\ MayAffect(o,n) \implies$
$\quad MayAccess^{Ind,Eve}(o,n) \wedge$
$\quad \exists o'.\ [MayAffect(o,o') \wedge$
$\quad\quad MayAffect(o',n)]$

Obj->Node with Dom Condition

$M,\sigma \vDash \forall o,rn,n.\ MayAffect(o,n) \wedge Dom(RN,n) \implies$
$\quad MayAccess^{Ind,Eve}(o,n) \wedge$
$\quad \exists rn \in RN.\ [MayAffect(o,rn) \wedge$
$\quad\quad MayAffect(rn,n)]$

ReNode->Node Condition

$M,\sigma \vDash \forall rn,n.\ MayAffect(rn,n) \implies$
$\quad \exists \sigma' \in Arising(M,\sigma).\ \lfloor rn.\texttt{node} \rfloor_{\sigma'} = \lfloor n \rfloor_{\sigma'}$

ReNode->ReNode Condition

$M,\sigma \vDash \forall rn,rn',n,n'.\ MayAffect(rn,rn')$
$\implies \exists \sigma' \in Arising(M,\sigma).$
// if d>0 then we can create new parent renode
$[\lfloor rn.\texttt{depth} \rfloor_{\sigma'} = \lfloor d \rfloor_{\sigma'} \wedge (\lfloor d \rfloor_{\sigma'} > 0) \wedge$
$\lfloor rn.\texttt{parent} \rfloor_{\sigma'} = \lfloor rn' \rfloor_{\sigma'} \wedge$
$\lfloor rn'.\texttt{depth} \rfloor_{\sigma'} = \lfloor d' \rfloor_{\sigma'} \wedge [(\lfloor d' \rfloor_{\sigma'} = (\lfloor d \rfloor_{\sigma'} - 1))] \wedge$
$\lfloor rn'.\texttt{node} \rfloor_{\sigma'} = \lfloor rn.\texttt{node.parent} \rfloor_{\sigma'}]$
$\vee$
// create new child node
$[\exists i \in \mathbb{R}_{\geq 0}\ \lfloor rn.\texttt{child(i)} \rfloor_{\sigma'} = \lfloor rn' \rfloor_{\sigma'} \wedge$
$\lfloor rn'.\texttt{parent} \rfloor_{\sigma'} = \lfloor rn \rfloor_{\sigma'} \wedge$
$\lfloor rn'.\texttt{depth} \rfloor_{\sigma'} = \lfloor d' \rfloor_{\sigma'} \wedge [(\lfloor d' \rfloor_{\sigma'} = (\lfloor d \rfloor_{\sigma'} + 1))] \wedge$
$\lfloor rn'.\texttt{node} \rfloor_{\sigma'} = \lfloor n.\texttt{child(i)} \rfloor_{\sigma'}]$

ReNode->Untrusted Condition

$M,\sigma \vDash \forall rn,o.\ MayAffect(rn,o) \implies$
$\quad o \in ReNode \vee o \in Node$

ReNode Definition

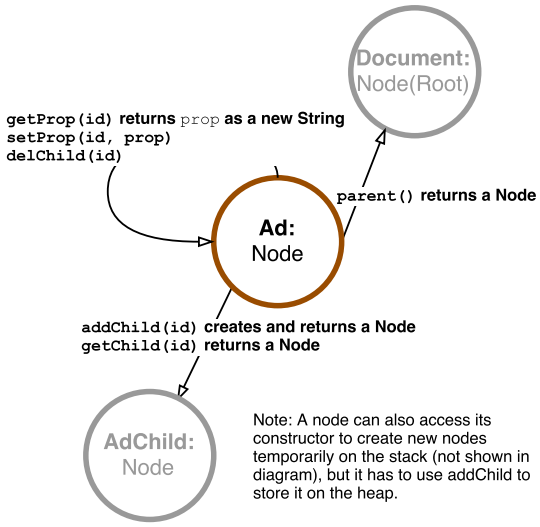
$M,\sigma \vDash \forall rn,rn',rn'',n,\ \forall \sigma' \in Arising(M,\sigma).$
$\quad \lfloor rn.\texttt{depth} \rfloor_{\sigma'} = \lfloor d \rfloor_{\sigma'} \wedge$
$\quad \lfloor rn.\texttt{node} \rfloor_{\sigma'} = \lfloor n \rfloor_{\sigma'} \wedge$
$\quad [\lfloor rn.\texttt{parent} \rfloor_{\sigma'} = \lfloor p \rfloor_{\sigma'} \wedge$
$\quad\ (\lfloor p \rfloor_{\sigma'} = \lfloor rn' \rfloor_{\sigma}) \vee (\lfloor p \rfloor_{\sigma'} = \texttt{null})]$
$\quad \wedge$
$\quad [\forall i \in \mathbb{R}_{\geq 0}.\ \lfloor rn.\texttt{child(i)} \rfloor_{\sigma'} = \lfloor c \rfloor_{\sigma'} \wedge$
$\quad\ (\lfloor c \rfloor_{\sigma'} = \lfloor rn'' \rfloor_{\sigma'}) \vee (\lfloor c \rfloor_{\sigma'} = \texttt{null})]$

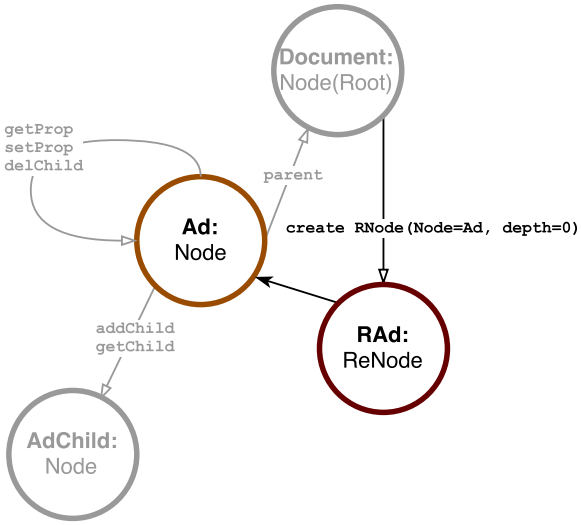
Node Definition

$M,\sigma \vDash \forall n,n',n'',\forall \sigma' \in Arising(M,\sigma).$
$\quad [\lfloor n.\texttt{parent} \rfloor_{\sigma'} = \lfloor p \rfloor_{\sigma'} \wedge$
$\quad\ (\lfloor p \rfloor_{\sigma'} = \lfloor n' \rfloor_{\sigma'}) \vee (\lfloor p \rfloor_{\sigma'} = \texttt{null})\ ]$
$\quad \wedge$
$\quad [\forall i \in \mathbb{R}_{\geq 0}.\ \lfloor n.\texttt{child(i)} \rfloor_{\sigma'} = \lfloor c \rfloor_{\sigma'} \wedge$
$\quad\ (\lfloor c \rfloor_{\sigma'} = \lfloor n'' \rfloor_{\sigma'}) \vee (\lfloor c \rfloor_{\sigma'} = \texttt{null})\ ]$

1) A simplified representation of a Javascript HTML DOM tree. A node can perform 6 functions and the result of each function call is pointed to by empty arrowheads. Notice below that giving away the capability of the **Ad** node to a third-party is unsafe, because using the `parent()` function call on the **Ad** node returns **Document,** the root node, from which all nodes and their capabilities in the entire DOM tree can be accessed.

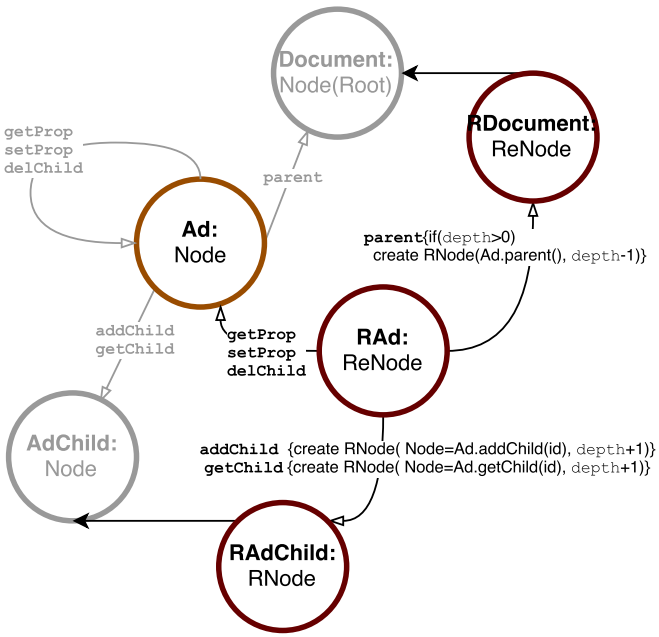
2) A `Node` can now construct an attenuating `ReNode` over a child `Node` it has created, and also specify an integer variable `depth` to restrict how far up in a tree the newly created `ReNode` can travel. A `ReNode` with `depth=0` means that it cannot access its immediate parent. Also, `depth` can only be declared once in the `ReNode` constructor and cannot be subsequently changed or re-declared (`depth` is of a Javascript `let` type). The `ReNode` possesses the capability of the `Node` that it wraps over (filled arrowhead in diagram below) but this is stored in a private field. Therefore the capability of `Node` is not accessible externally and can only be used internally by `ReNode`'s functions.





3) A `ReNode` has all the functions of a `Node`, and it forwards all capability-insensitive messages (that return a non-capability type - `getProp`, `setProp`, `delChild`) to the `Node` that it wraps over , and returns `Node`'s results. For capability-sensitive functions that return a capability (`addChild`, `getChild`, `parent`), `ReNode` always checks some condition and if successful, always creates and return a new `ReNode` imbued with an adjusted `depth` so as to protect the access integrity of the tree. The function `parent` succeeds only if the `ReNode` has sufficient depth access to call its immediate parent (`depth>0`).

4) In the final diagram below, notice how it is safe now to give away the capability of the `RNode` **RAd** to a third-party, when **RAd** is constructed by **Document** with `depth=0`. The wrapper guarantees that the user of **RAd** cannot modify the properties of **Document** through the chained function call `parent().setProp(id, prop)` because `parent()` will first fail. Consequently, the wrapper also prevents **RAd**'s user from accessing any other node descended from **Document.**



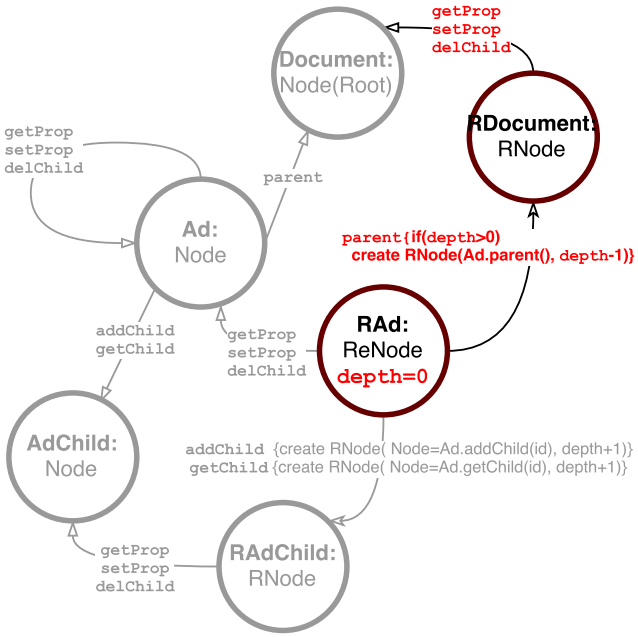


Figure 1: *JavaScript DOM Tree OCap Pattern*

4 Appendix

References

[1] David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[2] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[3] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 147–162. IEEE, 2016.

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

[5] Norman Hardy. Keykos architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[6] Daniel Matte. Open-Source Clues to Google's Mysterious Fuchsia OS. http://spectrum.ieee.org/tech-talk/computing/software/a-modern-os-from-google, April 2017.

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

[8] 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.

[9] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[10] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix.

[11] Maurice Vincent Wilkes and Roger Michael Needham. The cambridge cap computer and its operating system. 1979.

[12] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.