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).
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[3] 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

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[4].

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

[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.

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 [2].

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.

$$
\begin{array}{l}
\sigma \in \text{state} = \text{frame} \times \text{heap} \\
\varphi \in \text{frame} = \text{StackId} \rightarrow \text{val} \\
\chi \in \text{heap} = \text{addr} \rightarrow \text{object} \\
v \in \text{val} = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \mathit{addr} \cup \mathbb{N} \\
\text{StackId} = \{\mathit{this}\} \cup \text{VarId} \cup \text{ParId} \\
\text{object} = \text{ClassId} \times (\text{FieldId} \rightarrow \text{val})
\end{array}
$$

Module:

$$
\begin{array}{l}
M \in \text{Module} = \text{ClassId} \cup \text{SpecId} \\
\qquad\qquad\qquad\qquad \rightarrow \\
\qquad\qquad\qquad\qquad (\text{ClassDescr} \cup \text{Specification})
\end{array}
$$

Reach and Execution:

Arising Configurations

Domination:

3.1.1 MayAccess Definitions

We define in total four flavours of MayAccess predicates that describe the relation between two entities in a system (of arity 2 that represent the identifiers of these entities). These four flavours represent a combination of space (distance) and time (state):

- distance: directly (*Dir*) or indirectly (*Ind*)
- state: now (*Now*) or eventually (*Eve*)

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

$$
\begin{array}{l}
M,\sigma \models \text{MayAccess}^{Dir,Now}(x,e) \iff \\
\quad \exists f.\ \lfloor x.f \rfloor_\sigma = \lfloor e \rfloor_\sigma \lor \\
\quad (\sigma(\text{this}) = \lfloor x \rfloor_\sigma \land \exists y.\ \sigma(y) = \lfloor e \rfloor_\sigma)
\end{array}
$$

$$
\begin{array}{l}
M,\sigma \models \text{MayAccess}^{Dir,Eve}(x,e) \iff \\
\quad \exists \sigma' \in \text{Arising}(M,\sigma). \\
\quad M,\sigma' \models \text{MayAccess}^{Dir,Now}(x,e)
\end{array}
$$

$$
\begin{array}{l}
M,\sigma \models \text{MayAccess}^{Ind,Now}(x,e) \iff \\
\quad \exists \bar{f}.\ \lfloor x.\bar{f} \rfloor_\sigma = \lfloor e \rfloor_\sigma \lor \\
\quad (\sigma(\text{this}) = \lfloor x \rfloor_\sigma \land \exists y.\ \sigma(y.\bar{f}) = \lfloor e \rfloor_\sigma)
\end{array}
$$

$$
\begin{array}{l}
M,\sigma \models \text{MayAccess}^{Ind,Eve}(x,e) \iff \\
\quad \exists \sigma' \in \text{Arising}(M,\sigma). \\
\quad M,\sigma' \models \text{MayAccess}^{Ind,Now}(x,e)
\end{array}
$$

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

We surmarise the relationships between the four flavours of MayAccess in *Table 1*:

Table 1: *MayAccess*,*(x,e) Relations*

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

Let us assume that both x and e are well-defined. Note that without imposing any further assumptions (such as those from the OCap model), we have defined *both* 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, but we do *not* imply that such a path is traverseable by x (it might or might not be traverseable). Indeed, these definitions by themselves represent mere *possibilities* of interaction (or possible authorities), they do *not* represent that interaction (or authority) would succeed. The difference between $\text{MayAccess}^{Dir,Now}$(x,e) and $\text{MayAccess}^{Ind,Now}$(x,e) is only the computational distance between x and e on the reference graph, where the latter involves possibly intermediate entities (or objects in an object-oriented model).

What do the definitions mean then for non-OCap and OCap models in an object-oriented world? Again, let us assume both o and o′ are well-defined, valid object references. In non-OCap models, the possible presence of a global ambient authority means the predicates above say *nothing* about whether any interaction between an object o and o′ would succeed. This is true even if o possesses directly the reference of o′, where $\text{MayAccess}^{Dir,Now}$(o,o′) holds. For all we know, we could easily have in a non-OCap language a feature to completely restrict access to o′ using a global ambient authority, such that any object in the programming world which possesses references of o′ cannot use them, and all paths leading to o′ represent merely possibilities of interaction, but from which no authority can possibly arise.

Could we say more about OCap systems? In OCap systems, there can be no global ambient authority so that an object reference by itself represents both the designation and the authority to use the object. This therefore leads us to be able to make a crucial distinction between $\text{MayAccess}^{Ind,Now}$(o,o′) and $\text{MayAccess}^{Dir,Now}$(o,o′) in the OCap model:

- $\text{MayAccess}^{Ind,Now}$(o,o′) means—*only*
    1. a directed path from o to o′ exists (possible authority)
- $\text{MayAccess}^{Dir,Now}$(o,o′) means—*both*
    1. a directed path from o to o′ exists (possible authority) *and*
    2. o′s authority to use o′ *will* succeed

We elaborate the distinction with the following OCap example where there is a particular state σ of the system where, o_1.next points to o_2, and o_2.next points to o_3. o_2.next is a private method that can only be called internally by o_2. In this example, $M,\sigma \models \text{MayAccess}^{Ind,Now}(o_1,o_3)$ holds, regardless of whether o_2.next is traverseable by o_1. We say here that a path from o_1 to o_3 exists, but is not traverseable by o_1. On the other hand, $\text{MayAccess}^{Dir,Now}(o_1,o_3)$ does not hold true, because o_3 is not reachable from o_1 in a single step—o_2 sits between them on the reference graph as an intermediate object, and can possibly prevent or allow traversal from o_1 to o_3 (in this example, o_2 prevents such a traversal).

What does it mean then for $\text{MayAccess}^{Dir,Now}(*,*)$ to hold in an OCap model? From the same example, $\text{MayAccess}^{Dir,Now}(o_1,o_2)$ holds and implies a stronger form of access—it means that a path exists from o_1 to o_2, and that such a path *is* traverseable. This is because, by the definition of the predicate and configuration of the example, the object reference of o_2 exists within o_1's state. Therefore, o_2 is guaranteed therefore to be accessible, and its authority exercisable, by o_1, without the interference of any ambient authority. Notice how this o_1-o_2 relationship differs from the o_1-o_3 relationship, where o_1 cannot guarantee that it can exercise the authority of o_2.next which points to o_3, since o_2.next does not exist within o_1's state—it is possible that o_2.next is protected by x_2 through encapsulation and data-hiding.

Within an OCap model, we can now be convinced that the stronger

$\text{MayAccess}^{Dir,Now}(o_1,o_2) \simeq$
    o_1 has the capability of o_2

while the weaker $\text{MayAccess}^{Ind,Now}(o_1,o_3)$ does not say anything about whether o_1 has the capability of o_3, but that only a directed path exists. It does however, represent a necessary condition for capability.

$\text{MayAccess}^{Ind,Now}(o_1,o_3) \simeq$
    there is a directed path from o_1 to o_2

o_1 has the capability of $o_3 \implies$
    $\text{MayAccess}^{Ind,Now}(o_1,o_3)$

3.1.2 MayAffect Definitions

With our access predicates in place, we introduce a set of predicates that describe changes to the state of a system. Again, we highlight that these predicates are broad enough to describe both non-OCap and OCap models.

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

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

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

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

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

3.2 OCap Security Implications

3.2.1 What is protection?

In an object-oriented world[3] , 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 internal state and behaviour, strictly speaking, security of an object should govern over both the integrity of the object's fields (internal state) and whether the objects' methods can be called (behaviour). Our predicates are broad enough to enable a discussion of both protection of state (MayAffect^{Eve}) and behaviour ($\text{MayAccess}^{Dir,Eve}$)[4]. We emphasize however that protecting either state or behaviour of an object, does *not* necessarily imply the other. In fact, a common feature in OCap patterns is being able to protect a sensitive object's behaviours (they cannot be called directly by untrusted objects), but at the some allowing the same untrusted objects to modify the object's state in some controlled way.

There are however, some flexibility in working with objects, that allows us to simplify our discussion and work with only a broad definition of state protection in terms of our MayAffect^{Eve} predicate, *without* thinking too much about specific fields of the object we want to protect or the protection of behaviour. Moving away from our broad definition of whether an object may be affected (we defined it as being able to change at least one field of the object), to more precise spcifications of which particular field(s) of the object may be affected, should be trivial. We can in theory also stay and reason within our framework by separating the particular concerned field(s) of the object into separately encapsulated objects. We merely have to be careful as to *which* object's state we want to protect. Furthermore, in theory one can easily introduce a field within an object that behaves like a 'signal' which would be modified whenever a specified behaviour is called. Preventing a particular behaviour to be called by an untrusted object then becomes equivalent to denying the untrusted object the ability to modify the particular state of the signal field of the object.

With these simplifications, protection for us then becomes solely a matter of whether we can allow or deny an object to modify the state of another. To help us reason about protection, we first formalize our assumptions and the rules of OCap, using our predicates, to help guide us in constructing our necessary conditions in *subsection 3.3*. In the following *subsection 3.4* below we build the neccessary conditions for MayAffect^{Eve}, where MayAffect^{Eve} is placed in the antecedent, and we examine which of the family of MayAccess predicates is placed in the consequent.

3.3 Formal specifications of OCap Rules

Rule 1: Objects can only interact with each other through sending messages on capabilities. We begin our reasoning of protection in our OCap model, by first re-asserting the

[3] To simplify our discussion, we work with the variables {o,o'}∈Object for this entire section.

[4] Our $\text{MayAccess}^{Dir,Eve}$ is not weak enough 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.

necessary but not sufficient path condition of capability, and calling it PEC. By definition of our predicates, an object having a capability of o′ implies object o having a path to o′, but not vice-versa:

---

*Path-Execution Connectivity (PEC)
$M,\sigma \models \text{MayAccess}^{Dir,Now}(o,o') \implies \text{MayAccess}^{Ind,Now}(o,o')$

By contraposition,
$M,\sigma \models \neg\text{MayAccess}^{Ind,Now}(o,o') \implies \neg\text{MayAccess}^{Dir,Now}(o,o')$

Note also that PEC is a direct intepretation from Rule 1, since rule 1 says owning a capability is the only way of sending messages to another object, and therefore implies a path between the objects.

Rule 2: Objects cannot forge capabilities, and only connectivity begets connectivity.

---

*Global Path Existence (GPE)
$M,\sigma \models [\exists Y^*\in\text{Obj}.\ \text{MayAccess}^{Ind,Now}(Y^*,o')] \iff [\exists X^*\in\text{Obj}.\ \text{MayAccess}^{Dir,Now}(X^*,o')]$

By contraposition,
$M,\sigma \models [\forall X^*\in\text{Obj}.\ \neg\text{MayAccess}^{Dir,Now}(X^*,o')] \iff [\forall Y^*\in\text{Obj}.\ \neg\text{MayAccess}^{Ind,Now}(Y^*,o')]$

In addition to path being a necessary condition for capability, GPE gives us an additional new relation between the two concepts over the entire system. GPE says that in a given system of objects, iff there exists an object Y* which has a path to o′, then there exists an object X* that has the capability of o′. Looking from the left to right direction, since the path from Y* to o′ exists, then there must exist an object X* with a capability of o′ so that the path from Y* to o′ is well-established. Moving from the right to left direction, GPE says that if an object X* has a capability then there exist an object Y* with a path to o′, and this can be proven from PEC or by definition when (X*==Y*). The contraposition result of GPE says that we can prevent all paths to o′ through one of these conditions through denying all direct paths leading into o′.

However, GPE only postulates the existence of some object with some capability iff there exists some object with some path, and says nothing about how the objects are connected. From the connectivity begets connectivity rule, PEC, and GPE, we construct global path-execution relationships over the entire system:

---

*Global Path-Execution Connect. (GPEC)
$M,\sigma \models \text{MayAccess}^{Ind,Now}(o,o') \iff \exists X^*\in\text{Obj}.\ [\text{MayAccess}^{Ind,Now}(o, X^*) \wedge \text{MayAccess}^{Dir,Now}(X^*,o')]$

By contraposition,
$M,\sigma\ \forall X^*\in\text{Obj}.\ [\neg\text{MayAccess}^{Ind,Now}(o, X^*) \vee \neg\text{MayAccess}^{Dir,Now}(X^*,o')] \iff \neg\text{MayAccess}^{Ind,Now}(o,o')$

GPEC introduces an intermediate object X* between o and o′ that makes explicit the missing connection in GPE.

The contraposition result of GPEC says that for there to be no paths from o to o′ these configurations must hold:

- if there is some object x that o has a path to in σ, then x cannot have a path to o′ in σ.
- if there is some object x that has a path to o′ in σ, then o cannot have a path to x in σ.

The power in GPEC lies in the recursive predicate $\text{MayAccess}^{Ind,Now}(o, X^*)$ that can eventually be expanded into a set of $\text{MayAccess}^{Dir,Now}$ predicates that finishes with terminating predicate $\text{MayAccess}^{Ind,Now}(o, o)$, where the terminating predicate will always be true, unless the object o does not exist in σ. We now look for a relation between a path configuration at a state σ (*Now*) and eventual path configurations in states σ′ arising from σ (*Eve*).

*Eventual Path Connectivity 1 (EPC1)

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

By contraposition,

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

EPC1 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′. o either must already have a path to X* in σ, or that it will eventually receive the capability of X* in σ′. EPC1 is actually a formal representation of connectivity begets connectivity across time:

- **Initial Conditions or Endowment:** o has an existing path to o′ in σ, therefore X* refers to o′
- **Parenthood:** o will create o′ in some arising σ′, therefore X* refers to o
- **Introduction:** o will only obtain the path to o′ in some arising σ′, therefore X* refers to an object that is not the same object as o (X*≠o).

Note that for o to have an eventual path to o′, we require only o to have an eventual path to X* since we have stated that X* will have an eventual path to o′. If o already has a path to X* in σ then we know o can reach X* in σ′. 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′, and the only way we can guarantee this is to have X* possess 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, $\tilde{X}^{*}$ will have the capability of X*, and will therefore 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*.

The contraposition result of EPC1 says that for there to be no eventual paths from o to o′ these configurations must hold:

- if there is some object x that o has a path to in σ, then x cannot have an eventual path to o′
- if there is some object x that possesses the capability of o in σ, then x cannot have an eventual path to o′

Note that object x can refer to the same object as o. There is one final critical result from EPC1. 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′. This recursive expansion gives us both $\text{MayAccess}^{Dir,Now}$ and $\text{MayAccess}^{Ind,Now}$ predicates, where the terminating $\text{MayAccess}^{Ind,Eve}(o',o')$ always returns true if o′ exists. This result allows us to define $\text{MayAccess}^{Ind,Eve}(X^{*},o')$ in a configuration of paths completely based in σ.

What about capabilities? Using GPEC, we can always expand $\text{MayAccess}^{Ind,Now}(o,X^{*})$ into a set of $\text{MayAccess}^{Dir,Now}$ predicates that terminate with $\text{MayAccess}^{Ind,Now}(o,o)$ which is always true if o exists. The final result we will get is a configuration based purely on $\text{MayAccess}^{Dir,Now}$ predicates. We expand EPC1 into EPC2 to illustrate:

*Eventual Path Connectivity 2 (EPC2)

$$
\begin{aligned}
&M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(o,o') \\
&\quad\Longrightarrow \\
&\quad \exists X^{*}\in\text{Obj}.\{ \\
&\qquad [\exists X^{1}\in\text{Obj}.\ \text{MayAccess}^{Ind,Now}(o,X^{1}) \wedge \\
&\qquad\qquad \text{MayAccess}^{Dir,Now}(X^{1},X^{*}) \\
&\qquad\quad \vee \\
&\qquad \text{MayAccess}^{Dir,Now}(X^{*},o)] \\
&\qquad \wedge \\
&\qquad \exists Y^{*}\in\text{Obj}.\{[\text{MayAccess}^{Ind,Now}(X^{*},Y^{*}) \\
&\qquad\qquad \vee \\
&\qquad\qquad \text{MayAccess}^{Dir,Now}(Y^{*},X^{*})]
\end{aligned}
$$

$$\begin{array}{l}
\qquad\qquad \wedge \\
\qquad\qquad \text{MayAccess}^{Ind,Eve}(\text{Y*},\text{o}')\} \\
\quad \}
\end{array}$$

By contraposition,

$$\begin{array}{l}
\text{M},\sigma \vDash \\
\quad \forall \text{X*}{\in}\text{Obj}.\{ \\
\qquad [\forall \text{X}^1{\in}\text{Obj}.\ \neg\text{MayAccess}^{Ind,Now}(\text{o},\text{X}^1) \vee \\
\qquad\qquad \neg\text{MayAccess}^{Dir,Now}(\text{X}^1,\text{X*}) \\
\qquad\quad \wedge \\
\qquad \neg\text{MayAccess}^{Dir,Now}(\text{X*},\text{o})] \\
\qquad \vee \\
\quad \forall \text{Y*}{\in}\text{Obj}.\ \{[\neg\text{MayAccess}^{Ind,Now}(\text{X*},\text{Y*}) \\
\qquad\qquad \wedge \\
\qquad\qquad \neg\text{MayAccess}^{Dir,Now}(\text{Y*},\text{X*})] \\
\qquad\qquad \vee \\
\qquad\qquad \neg\text{MayAccess}^{Ind,Eve}(\text{Y*},\text{o}')\} \\
\quad \Longrightarrow \\
\quad \neg\text{MayAccess}^{Ind,Eve}(o,o')
\end{array}$$

To give a concrete example, let us assume a simple system with only three objects, o, x, o′.

*Eventual Path Example

$$\begin{array}{l}
\text{M},\sigma \vDash \text{MayAccess}^{Ind,Eve}(\text{o},\text{o}') \\
\quad \Longrightarrow \\
\qquad [\text{MayAccess}^{Ind,Now}(\text{o},\text{o}) \wedge \\
\qquad\quad \text{MayAccess}^{Dir,Now}(\text{o},\text{x}) \\
\qquad\quad \vee \\
\qquad \text{MayAccess}^{Dir,Now}(\text{x},\text{o})] \\
\qquad \wedge \\
\qquad \{[\ \text{MayAccess}^{Ind,Now}(\text{x},\text{x}) \wedge \\
\qquad\quad \text{MayAccess}^{Dir,Now}(\text{x},\text{o}') \\
\qquad\qquad \vee \\
\qquad\quad \text{MayAccess}^{Dir,Now}(\text{o}',\text{x})] \\
\qquad\quad \wedge \\
\qquad\quad \text{MayAccess}^{Ind,Eve}(\text{o}',\text{o}')\} \\
\qquad \}
\end{array}$$

In the example, we make concrete EPC2 using X* = x and Y* = o′, where the expansion terminates at the three predicates, $\text{MayAccess}^{Ind,Now}(\text{o},\text{o})$, $\text{MayAccess}^{Ind,Now}(\text{x},\text{x})$ and $\text{MayAccess}^{Ind,Eve}(\text{o}',\text{o}')$. Let us assume o, x, and o′ always exists, such that these predicates would return true. This example then illustrates an eventual path from o to o′ can only exist if one of these four conditions in σ hold:

- o has the capability of x, and x has the capability of o′
- o has the capability of x, and o′ has the capability of x
- x has the capability of o, and x has the capability of o′
- o has the capability of x, and x has the capability of o′

Note how, these four configurations are present state configurations at σ, but allows us to reason about whether a potential path from o to o′ can exist in all arising states σ′ from σ.

So far we have developed results for paths from o to o′, but what about a direct path from o to o′? Luckily, we only need construct EEC from EPC1, the only difference being we now require the path from o to X* in EPC1 to be traversable.

*Eventual Execution Connectivity (EEC)

$$\begin{array}{l}
\text{M},\sigma \vDash \text{MayAccess}^{Dir,Eve}(\text{o},\text{o}') \\
\quad \Longrightarrow \\
\quad \exists \text{X*}{\in}\text{Obj}.[(\text{MayAccess}^{Dir,Now}(\text{o},\text{X*}) \vee \\
\qquad\qquad \text{MayAccess}^{Dir,Now}(\text{X*},\text{o})) \\
\qquad\qquad \wedge \\
\qquad\qquad \text{MayAccess}^{Dir,Eve}(\text{X*},\text{o}')]
\end{array}$$

By contraposition,

$$\begin{array}{l}
\text{M},\sigma \vDash \forall \text{X*}{\in}\text{Obj}.\ \{[\neg\text{MayAccess}^{Dir,Now}(\text{o},\text{X*}) \wedge \\
\qquad\qquad \neg\text{MayAccess}^{Dir,Now}(\text{X*},\text{o})] \\
\qquad\qquad \vee \\
\qquad\qquad \neg[\text{MayAccess}^{Dir,Eve}(\text{X*},\text{o}')]\} \\
\quad \Longrightarrow \\
\quad \neg\text{MayAccess}^{Dir,Eve}(o,o')
\end{array}$$

The power of EEC is similar to EPC in that it allows an arbitrary recursive expansion of the $\text{MayAccess}^{Dir,Eve}(\text{X*},\text{o}')$ till the terminating $\text{MayAccess}^{Dir,Eve}(\text{o}',\text{o}')$ that is always true.

We can always use an automated theorem prover to prove whether a path can arise between two objects in a present configuration system. We now turn our attention formalizing Ocap state changes rules in the system.

3.4 Protection

With our formalizations of OCap rules in place, we can now reason about enforcing protection. In this subsection we focus on finding the necessary conditions for the weaker predicate MayAffect^{Eve} rather than MayAffect^{Now} in the antecedent. This is because negation on both sides of the implication, would yield a stronger $\neg\text{MayAffect}^{Eve}$ in the consequent. In practical terms, if we are concerned with the protection of o′ from o, it is also often not very useful to have a policy where $\neg\text{MayAffect}^{Now}$(o,o′) holds but $\neg\text{MayAffect}^{Eve}$(o,o′) does not. Furthermore, by ensuring $\neg\text{MayAffect}^{Eve}$(o,o′) holds, we can also ensure $\neg\text{MayAffect}^{Now}$(o,o′) holds since by contraposition:

$$[\text{MayAffect}^{Now}(o,o') \implies \text{MayAffect}^{Eve}(o,o')] \implies [\neg\text{MayAffect}^{Eve}(o,o') \implies \neg\text{MayAffect}^{Now}(o,o')]$$

Furthermore, in building the necessary conditions for the predicate MayAffect^{Eve}, we are also more concerned with finding some structure that contains the configuration of $\text{MayAccess}^{*,Now}$, rather than $\text{MayAccess}^{*,Eve}$. This is because it is much easier to prove a configuration of relations that holds in *one* specific state than think about whether a configuration holds in *all* possible eventual states, which makes the former much easier to implement.

Rule 3: Objects may have private encapsulation of state and behaviour
To begin, we make clean in our reasoning that all fields in our objects can only be declared private. Consequently, this implies the necessary condition that an object's state can only be modified if one of it's behaviour is called, either by itself or one other object in the system that holds its capability. This is also implied by the OCap rule that there is no ambient authority that can interact with any object's behaviour.

---

*PRIVATE FIELDS ASSUMPTION (PFA)
$$M,\sigma \models \exists X^*\in\text{Obj}.\ \text{MayAffect}^{Now}(X^*,o') \implies \exists Y^*\in\text{Obj}.\ \text{MayAccess}^{Dir,Now}(Y^*,o')$$

By contraposition,
$$M,\sigma \models \forall Y^*\in\text{Obj}.\ \neg\text{MayAccess}^{Dir,Now}(Y^*,o') \implies \forall X^*\in\text{Obj}.\ \neg\text{MayAffect}^{Now}(X^*,o')$$

PFA means that in order for an object o′ to change, it must be done through some object Y* calling its behaviour (Y* can refer to the same object as X*). Equivalently, denying all objects in the system the ability to call an object's behaviour implies that no object can modify the object's state.

---

*GLOBAL STATE CHANGE EXISTENCE (GSCE)
$$M,\sigma \models [\exists X^*\in\text{Obj}.\ \text{MayAffect}^{Now}(X^*,o')] \implies [\exists Y^*\in\text{Obj}.\ \text{MayAccess}^{Dir,Now}(Y^*,o')] \implies [\exists Z^*\in\text{Obj}.\ \text{MayAccess}^{Ind,Now}(Z^*,o')]$$

GSCE can be derived from our predicate definitions, and makes explicit the connection betwen state change, paths, and capabilities. The first implication states that if an object can change the state of o′, then there must exist an object in the system that holds the capability of o′. The second implication is that there must exist an object in the system that holds a path to o′. We view reasoning about state change as reasoning about protection, which we elaborate in the next subsection.

3.4.1 Protection is about local object-path denial and global capability denial

$$M,\sigma \models \text{MayAffect}^{Eve}(o,o') \not\Longrightarrow \text{MayAccess}^{Dir,Eve}(o,o')$$

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

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

$$M,\sigma \models \text{MayAccess}^{Dir,Eve}(o,o') \implies \text{MayAccess}^{Ind,Eve}(o,o')$$

$$M,\sigma \models \neg\text{MayAccess}^{Ind,Eve}(o,o') \implies \neg\text{MayAccess}^{Dir,Eve}(o,o') \wedge \neg\text{MayAffect}^{Eve}(o,o')$$

We make the important result that protection of an object's state is path denial to the object, where path is a necessary condition for an object to affect another object. An object o having the capability of o′ is **NOT** a necessary condition for o to modify the state of o′. However, the existence of *some* object X* (that can refer to o, or other objects) having the capability of o′, is a necessary condition for o to modify o′. See NEC1 and NEC2 below which illustrates this. This is precisely why attenuating objects are so powerful and useful in OCap patterns. We can always deny o the capability of o′, but still allow o to modify o′ in some protected way. To say it in another way, denial of paths is much stronger, as it implies protection *and* denial of capability, but denial of capability does *not* imply protection and denial of paths.

---

*NECESSARY EXECUTION CONDITION 1 (NEC1)
M,σ ⊨ MayAffect^{Eve}(o,o′) $\implies$
  ∃X*∈Obj. $\text{MayAccess}^{Dir,Eve}$(X*,o′)

By contraposition and quantifier equivalence,
M,σ ⊨ [∀X*∈Obj. ¬$\text{MayAccess}^{Dir,Eve}$(X*,o′)] $\implies$
  ¬MayAffect^{Eve}(o,o′)

NEC1 says that in order for an object o to modify the state of o′, there must exist an object X* that has a direct path to o′, and that X* *can* traverse such a path. This is derived trivially from our PFA assumption. Consequently, in our contraposition result, we can deny *all* objects in our system capability of o′ to enforce the protection of o′ from o. NEC1 does not yield a very useful result, as this is just another way of implementing protection of o′ from o through denying all paths to o′ in the system reference graph. To see why denying all objects the capability of o′ is equivalent to denying all paths to o′, see the contraposition result from GPE.

In NEC1, if X* refers to the same object as o, then we have a straightforward configuration where object o has the capability of o′ and can therefore affect o′. However, very crucially, o having the capability of o′ is *not* a sufficient condition—denying o the capability of o′ does *not* deny o the ability to affect o′, since X* can refer to an object that is *not* o. NEC2 below is an expansion of NEC1 to illustrate this crucial point.

---

*NECESSARY EXECUTION CONDITION 2 (NEC2)
M,σ ⊨ MayAffect^{Eve}(o,o′) $\implies$
  [$\text{MayAccess}^{Dir,Eve}$(o,o′) ∨
  ∃X*∈Obj, X*≠o. $\text{MayAccess}^{Dir,Eve}$(X*,o′)]

By contraposition,
M,σ ⊨ [¬$\text{MayAccess}^{Dir,Eve}$(o,o′) ∧
  ∀X*∈Obj, X*≠o. ¬$\text{MayAccess}^{Dir,Eve}$(X*,o′)]
  $\implies$ ¬MayAffect^{Eve}(o,o′)

With this clarification, we then reason about the relationship between protection of state (MayAffect) and paths ($\text{MayAccess}^{Ind,*}$). The question we ask is what is the relation between state protection in eventual outcomes, and present path configurations? We build this relationship from the basics by progressively finding stronger conditions of MayAffect. We begin with the first condition, which says that for an object to affect another, an *eventual* path has to exist. This can be derived from the definitions of our predicates.

---

*NECESSARY PATH CONDITION 1 (NPC1)
M,σ ⊨ MayAffect^{Eve}(o,o′) $\implies$
  $\text{MayAccess}^{Ind,Eve}$(o,o′)

By contraposition,
M,σ ⊨ ¬$\text{MayAccess}^{Ind,Eve}$(o,o′) $\implies$
  ¬MayAffect^{Eve}(o,o′)

Here, we immediately see a first defensive outcome of the OCap model. Having no eventual paths from object o to o′ guarantees that the state of object o′ cannot be modified by object o.

So far our results serve as a good base to enforce ¬$\text{MayAffect}^{Eve}(o,o')$ but we require *stronger* necessary conditions for $\text{MayAffect}^{Eve}(o,o')$, because we need to understand the present path configurations that can deny o from modifying o′. Luckily, we have already built the necessary ingredients in our formalizations of OCap rules in *subsection 3.3*, where have painstakingly derive the relationship between present path-capability configurations and eventual paths. Since NPC1 tells us that the eventual paths is a necessary condition for object state modification, and EPC1 tells us the the relationship between eventual paths and present path-capability configurations, we combine the two to present:

---

*NECESSARY PATH CONDITION 2 (NPC2)
M,σ ⊨ MayAffect^{Eve}(o,o′)
  $\implies$
  ∃X*∈Obj.[($\text{MayAccess}^{Ind,Now}$(o,X*) ∨
      $\text{MayAccess}^{Dir,Now}$(X*,o))
        ∧
      $\text{MayAccess}^{Ind,Eve}$(X*,o′)]

By contraposition,
M,σ ⊨ ∀X*∈Obj. [¬$\text{MayAccess}^{Ind,Now}$(o,X*) ∧
      ¬$\text{MayAccess}^{Dir,Now}$(X*,o)]
      ∨
      ¬$\text{MayAccess}^{Ind,Eve}$(X*,o′)]
  $\implies$
  ¬$\text{MayAffect}^{Eve}(o,o')$

, where we can always use GPEC to expand indirect paths into direct capabilities till we reach the terminating origin o and recursively use the condition to expand the condition till we reach the terminating o′. NPC2 gives us the present path configurations that will allow an eventual path from o to o′ as a necessary condition for o to modify o′. NPC2 tells us that we can enforce protection of o′ from o by denying all eventual paths from o to o′. But if those objects are path-isolated, how do we allow them to cooperate?

3.5 What about 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 useful. Cooperation between objects require paths to exist between objects so it is rare or almost impossible to have two objects o and o′ that cooperate while $\neg\text{MayAccess}^{Ind,Eve}(o,o')$ holds. The quintessential question in OCap systems is whether we can enforce protection such as controlling whether o may affect o′ while *still* allowing paths to exist between them. This is possible, because of Rule 3 that says objects may have private encapsulation of state and behaviour. To begin, let us assume a simple object o′ that has *only* private behaviours. In such a scenario, since all fields in our system are also assumed to be private, then we know:

$\models \forall X^*{\in}\text{Obj}, X^*{\neq}o'.\ \neg\text{MayAffect}^{Eve}(X^*,o')$

Notice, how protection of o′ can occur independent of paths and capability, due to object encapsulation. This means that objects can specify security policies that it chooses to enforce *internally*. Can we further develop our necessary conditions to show this?

NEC2 and NPC2 show the necessary capability and path conditions for protection. Is there a relation that can capture the essence of both? Notice how, we have said in NEC2 that there must exist some object in the system (that is not necssarily o) that has the capability of o′, while in NPC2, we said that o must have a path to o′. Surely it must mean then that the object X* in NEC2 that has the capability of o′ refers to the last object on the path from o to o′ mentioned in NPC2. We now bring these results together, by making use of GPEC to make explicit the last object in the eventual path holding the capability of o′ in NPC2:

NECESSARY EXECUTION-PATH CONDITION
(NEPC*)

$$\begin{aligned}
&M,\sigma \models \text{MayAffect}^{Eve}(o,o')\\
&\quad\Longrightarrow\\
&\quad \exists X^*{\in}\text{Obj}.[(\text{MayAccess}^{Ind,Now}(o,X^*) \lor\\
&\qquad\qquad \text{MayAccess}^{Dir,Now}(X^*,o))\\
&\qquad\qquad \land\\
&\quad \exists Y^*{\in}\text{Obj}.\ [\text{MayAccess}^{Ind,Eve}(X^*, Y^*) \land\\
&\qquad\qquad \text{MayAccess}^{Dir,Eve}(Y^*,o')]
\end{aligned}$$

By contraposition,

$$\begin{aligned}
&M,\sigma \models \forall X^*{\in}\text{Obj}.\ [\neg\text{MayAccess}^{Ind,Now}(o,X^*) \land\\
&\qquad\qquad \neg\text{MayAccess}^{Dir,Now}(X^*,o)]\\
&\qquad\qquad \lor\\
&\quad \forall Y^*{\in}\text{Obj}.\ [\text{MayAccess}^{Ind,Eve}(X^*, Y^*) \lor\\
&\qquad\qquad \text{MayAccess}^{Dir,Eve}(Y^*,o')]\\
&\Longrightarrow\\
&\quad \neg\text{MayAffect}^{Eve}(o,o')
\end{aligned}$$

NEPC* is nice, because we can see the necessary path condition and the condition that an object must hold the capability of o′. But what does it mean to modify an object? No matter how long the path from o to o′ is, surely to modify o′ requires the last object on our path to *be* the 'final messenger' that will call the object's behaviour to modify it—which means $\text{MayAffect}^{Eve}(Y^*,o')$ *must* hold. We express this by extending NEPC1* to get NEPC2*:

NECESSARY EXECUTION-PATH CONDITION 2
(NEPC2*)

$$\begin{aligned}
&M,\sigma \models \text{MayAffect}^{Eve}(o,o')\\
&\quad\Longrightarrow\\
&\quad \exists X^*{\in}\text{Obj}.[(\text{MayAccess}^{Ind,Now}(o,X^*) \lor\\
&\qquad\qquad \text{MayAccess}^{Dir,Now}(X^*,o))\\
&\qquad\qquad \land\\
&\quad \exists Y^*{\in}\text{Obj}.\ [\text{MayAccess}^{Ind,Eve}(X^*, Y^*) \land\\
&\qquad\qquad \text{MayAccess}^{Dir,Eve}(Y^*,o') \land\\
&\qquad\qquad \text{MayAffect}^{Eve}(Y^*,o')]
\end{aligned}$$

By contraposition,

$$\begin{aligned}
&M,\sigma \models \forall X^*{\in}\text{Obj}.\ [\neg\text{MayAccess}^{Ind,Now}(o,X^*) \land\\
&\qquad\qquad \neg\text{MayAccess}^{Dir,Now}(X^*,o)]\\
&\qquad\qquad \lor\\
&\quad \forall Y^*{\in}\text{Obj}.\ [\text{MayAccess}^{Ind,Eve}(X^*, Y^*) \lor\\
&\qquad\qquad \text{MayAccess}^{Dir,Eve}(Y^*,o') \lor\\
&\qquad\qquad \text{MayAffect}^{Eve}(Y^*,o')] \quad \Longrightarrow\\
&\quad \neg\text{MayAffect}^{Eve}(o,o')
\end{aligned}$$

NEPC2* has the intuitive meaning that in order to say object o may modify object o′ in some eventual state, *all* these three conditions must hold:

- there must be some present path configuration such that an eventual path from o to o′ can arise
- there must be an object Y* that sits as the last object along the eventual path to o′, that has the capability of o′
- object Y* must be able to modify o′

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 on how *all* objects with the direct capability of our trusted object may modify the object

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 path isolation, which is a distinctive feature of OCap models.

The similarity is that both models allow us to specify policies *within* objects as to whether an object's behaviour can be called (non-OCap systems can do so 'externally' through a global ambient authority). 'Internal' object protection can be done through using defensive programming techniques like encapsulation and checking of conditions for behaviours to succeed. In a way these forms of protection can be 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.

How do we then express these internal object security policies in our logical framework? The nice thing is we already have a predicate MayAffect^{Eve}(*,*) that can accomodate this. Notice how in NEPC2*, object Y* sits as the last object in the path from o to o′ *and* has the capability of o′. This gives us a clue as to how we can enable protection of o′ without denying paths, by *specifying whether Y* can affect o′ explicitly in security policies* that are independent of paths.

3.5.1 How can security policies be enforced by objects?

In the previous example of an object with all fields and behaviours private, we have demonstrated in OCap systems that objects can, at the very least enforce protection policies on *itself*. But can we have objects enforcing security policies on *other* objects? Let us examine this with another example.

If we have some starting configuration in a system where there is some object X* that belongs to a set of objects belonging to the same class called AttenObj that has the capability of o′, and no other objects have the capability of o′, then we know any eventual path to o′ has to include and end with the sub-path from X* to o′, *unless* X* leaks the capability of o′ to some other object. If object X* has no way of leaking the capability of object o′, then it has the power to dictate the policies in all eventual states that can affect o′ through itself (since it holds the direct capability of o′), and state more precise stronger necessary conditions for the MayAffect^{Eve}(X*,o′) predicate. To put it in another way, we can now allow a path to exist from X* to o′, *and* still enforce ¬MayAffect^{Eve}(X*,o′), because the negation of these stronger security policy conditions results in weaker antecedents that imply ¬MayAffect^{Eve}(X*,o′).

- X* has no behaviour that will make use of and call the capability of o′:

  $M,\sigma \models \forall X^* \in \text{AttenObj}.\ \text{MayAffect}^{Eve}(X^*,o')$
  $\implies \bot$

- X* has only one behaviour that will modify o′ if it holds a boolean field `access` that is evaluated to true:

  $M,\sigma \models \forall X^* \in \text{AttenObj}.\ \text{MayAffect}^{Eve}(X^*,o')$
  $\implies$
  $\exists \sigma' \in \text{Arising}(M,\sigma).$
  $\lfloor \texttt{x.allow} \rfloor_{\sigma'} = \lfloor \texttt{true} \rfloor_{\sigma'}$

A key result from this analysis shows the importance of attenuating objects, objects that sit as the last line of defence in the eventual paths to our protected objects. In the OCap pattern examples we will go through in the next section, protection of an object is done through some attenuating object that the trusted object knows will never leak its capability directly (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.

If the protected object can guarantee through the system of configuration that those attenuating objects are the *only* objects in the entire system that holds its capability in *all* eventual states, then the protected object can dictate the conditions and degree of control on how it can be used by untrusted objects in all eventual states through those attenuating objects. This ability of the protected object to dictate control across time and conditions enables cooperation, where conditions on how the protected object will cooperate is written in those attenuating objects.

We now see 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.

3.6 Pattern 1: The JavaScript DOM Tree

We use a JavaScript DOM Tree OCap pattern largely inspired by the example in Devriese et al.[1] where they use a Kripke worlds framework to reason about the pattern. We define the following variables throughout our pattern:

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

*NODE VULNERABILITY
$\forall$o, n. MayAccess^{Dir,Now}(o,n)
$\quad\Longrightarrow$
$\quad\quad$MayAffect^{Now}(o,n) $\wedge$
$\quad\quad\forall$n′. MayAccess^{Dir,Now}(o,n′)

The vulnerability of a node lies in the fact that it contains a public method `setProperty(key,value)` that will modify an internal mapping data structure. A node also has a public field `parent` that will return the capability of its parent node. Consequently, this allows an object which has the 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.

RENODE TO NODE CONDITION
M,$\sigma \models \forall$rn$\in$ReNode. MayAffect^{Eve}(rn,n)
$\quad\Longrightarrow$
$\quad\exists\sigma' \in$Arising(M,σ).
$\quad\quad\lfloor$`rn.node`$\rfloor_{\sigma'} = \lfloor$e$\rfloor_{\sigma'} \wedge$ {`e is n`}

RENODE PARENTHOOD CONDITION
M,$\sigma \models \forall$rn,rn.parent $\in$ReNode.
MayAccess^{Dir,Eve}(rn,rn.parent)<—Or do we need a MayCreate predicate?
$\quad\Longrightarrow$
$\quad\exists\sigma' \in$Arising(M,σ).
$\quad\wedge\ \lfloor$`rn.depth`$\rfloor_{\sigma'} = \lfloor$e$\rfloor_{\sigma'} \wedge$ e > 0
$\quad\wedge\ \lfloor$rn.parent `depth`$\rfloor_{\sigma'} = \lfloor$e′$\rfloor_{\sigma'} \wedge$ e′ = e - 1

References

[1] 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.

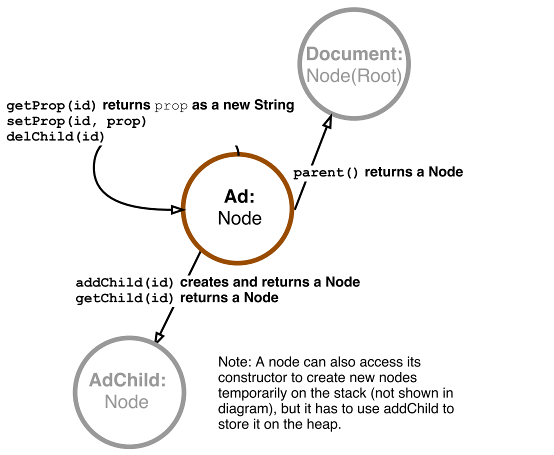
[2] Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Reasoning about risk and trust in an open word. 2015.

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

[4] 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.

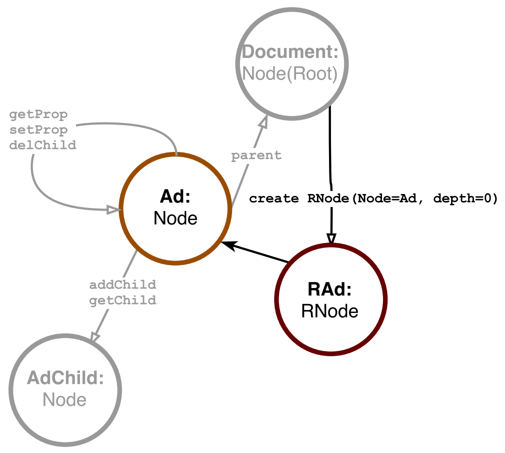
1) A simplified representation of a Javascript HTML DOM tree. A node can perform 6 functions and the result of each function call is pointed to by empty arrowheads. Notice below that giving away the capability of the **Ad** node to a third-party is unsafe, because using the `parent()` function call on 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.



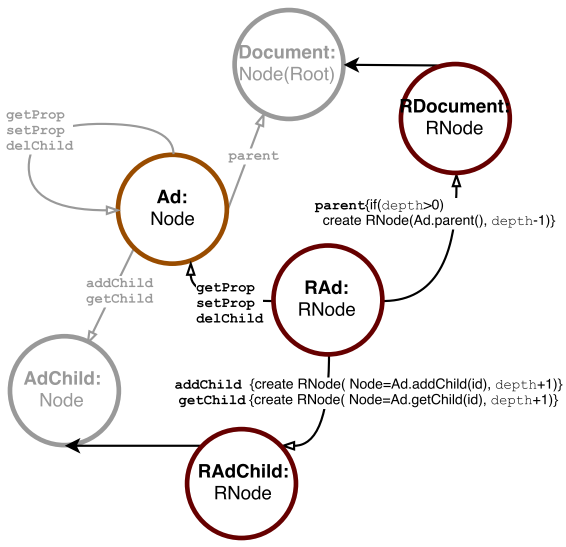
Document:
Node(Root)
getProp(id) returns prop as a new String
setProp(id, prop)
delChild(id)
parent() returns a Node
Ad:
Node
addChild(id) creates and returns a Node
getChild(id) returns a Node
AdChild:
Node

Note: A node can also access its constructor to create new nodes temporarily on the stack (not shown in diagram), but it has to use addChild to store it on the heap.

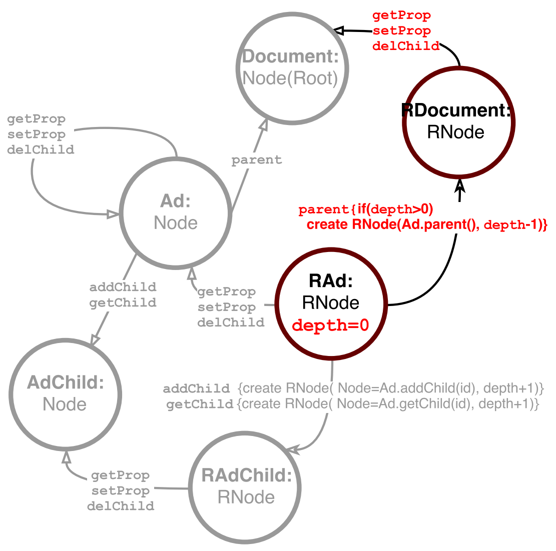


3) A `ReNode` has all the functions of a `Node`, and it forwards all capability-insensitive messages (that return a non-capability type - `getProp`, `setProp`, `delChild`) to the `Node` that it wraps over , and returns `Node`'s results. For capability-sensitive functions that return a capability (`addChild`, `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.**



Document:
Node(Root)
RDocument:
RNode
getProp
setProp
delChild
parent
Ad:
Node
parent{if(depth>0)
create RNode(Ad.parent(), depth-1)}
addChild
getChild
getProp
setProp
delChild
RAd:
RNode
AdChild:
Node
addChild {create RNode(Node=Ad.addChild(id), depth+1)}
getChild {create RNode(Node=Ad.getChild(id), depth+1)}
RAdChild:
RNode



getProp
setProp
delChild
Document:
Node(Root)
RDocument:
RNode
getProp
setProp
delChild
parent
Ad:
Node
parent{if(depth>0)
create RNode(Ad.parent(), depth-1)}
addChild
getChild
getProp
setProp
delChild
RAd:
RNode
depth=0
AdChild:
Node
addChild {create RNode(Node=Ad.addChild(id), depth+1)}
getChild {create RNode(Node=Ad.getChild(id), depth+1)}
getProp
setProp
delChild
RAdChild:
RNode