Imperial College London

Department of Computing

Risk and Trust in Open Systems

Author:
Shu Peng Loh

Supervisor:
Sophia Drossopoulou

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

Contents

Risk and Trust in Open Systems:
Towards formalising Permission and Authority, and specifying policies for Object Capability Patterns

Shu-Peng Loh

Imperial College London

Abstract

We revisit the concepts of *permission* and *authority* in the works of Miller[Mil06] and Drossopoulou et al. [DNMM16], and we contribute to the literature by proposing new formal definitions of permission and authority (MayAccess and MayCall predicates in our paper). We show how our definitions can be used to reason about object-to-object interactions, and in particular, cooperation between objects and protection of objects from unknown code. Another contribution we made is that we propose and show how the concept of domination over objects can be used to specify and reason about safe or vulnerable cooperation, which is novel but inspired by the work of Clarke et al. on ownership types [CPN98]. Our paper then show how using a mix of our proposed formalisations of permission and authority, and Hoare logics, we can formally specify well-established Object-Capability (OCap) patterns in the form of *OCap policies* [DN13, DN14, DNM15, DNMM15]. In particular, we reason about protecting the properties of nodes in a DOM Tree, inspired by the works of Maffeis et al.[MMT10] and Devriese et al.[DBP16], but we do so within the capability-safe language Pony [CDBM15]. We believe our methodology and specifications of OCap patterns are less complex than other methodologies in the literature[DBP16, SGD17], which will allow a programmer to reason convincingly how the use of attenuating objects in capability-safe languages can help to facilitate cooperation betweeen known and unknown code, while still preventing vulnerabilities by preserving desired properties in an open system. Lastly, we end our paper with some preliminary insights on thinking about security and risk in a non-OCap system by using the Ethereum[Woo14] blockchain as a case study.

1 Introduction

The power of distributed modern computing lies in facilitating cooperation between multiple agents, but it comes with risk as an agent is vulnerable to *unexpected* outcomes[1]. 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]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.

It is therefore not surprising that *thinking* about security while building a complex system of cooperation that involves the interplay between trusted(known) and untrusted(unknown) code components is extremely challenging to any programmer. To some programmers, security, is often treated as a *separate* mental burden to be considered separately from the functionality of a system. 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[Mil06] 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. This is done through leveraging the object-oriented programming paradigm and imposing certain prohibitions or restrictions. Within the OCap literature, there are two central concepts that describe how objects interact: permission and authority. The OCap literature also features prominently how using an OCap system would accomplish cooperation without vulnerability through Object-Capability Patterns (OCap patterns). OCap patterns are important, because they demonstrate how objects can cooperate with unknown code while preserving desirable properties in a system.

1.1 Our contributions

While there have been considerable work done on OCap patterns in programming languages [MS03, Mil06, Mur10], less work has been done on formally specifying them. The only works we know so far are [DNMM15, DBP16, SGD17], which are all very recent in the literature. Our paper builds on the works of [DNMM15], because we claim that the methodology of using OCap policies to specify OCap patterns to be intuitive and straightforward. While they may seem less sophisticated than the methodologies in other works, we believe they are powerful enough to reason convincingly about the interactions between trusted code and code of unknown provenance. To formally specify OCap patterns in the style of OCap policies, our paper contributes to the literature by proposing formal predicates for permission and authority that describe access dynamics between objects. Miller first proposed the concepts of permission and authority in his PhD thesis[Mil06], following which the formal definitions of these concepts were proposed and refined in the work of Drossopoulou et al. [DNMM16]. Our paper defines permission and authority in a formal way that is different from the aforementioned works, but we generally adopt the same informal meanings (that having permission means the right to invoke an object's behaviours, and authority to mean the ability to cause effects). More precisely, our paper defines *permission* as having the reference of another object (right to invoke that object's behaviours), either directly or indirectly—through multiple object reference(s)—and either currently or eventually—in some current or future state of the system; and defines *authority* to mean being *able* to invoke an object's behaviours; i.e. we distinguish between having the *right to do* something (permission) and *being able to do* something (authority)[2]. Furthermore, our paper implicitly treats invoking behaviours of objects as 'effects' in the system. We claim that these insights are novel and our motivation for defining them can be found in §3.2.

[2]An anology would be having the right to perform an impossible task does not imply the task becomes possible.

To put our formal definitions of permission and authority in practice and demonstrate their usefulness, we further propose the novel use of the concept of domination over objects that is inspired by the works of [CPN98] on ownership types. Our domination predicate describes a special set of objects that dominate a particular object—all permissions to use the particular object have to involve the permission of a member of the dominating set, and all authorities have to involve the authority of a member of the dominating set.

With our formal definitions of permission, authority, and domination, our paper then argues how we can use them to reason and distinguish concepts of *isolation*, *cooperation*, *vulnerability* and *protection*. We claim that our framework allows one to understand and reason about these concepts in a highly straightforward and intuitive way without treating security as a separate burden from the functionality of the system, which we hope would encourage programmers to seek the use of the object-capability model to build robust and secure systems.

To show this in practice, our paper provides an illustration of three OCap patterns found in sections 5.1.4, 5.3.1 and 5.3.2, but we focus only on providing the OCap policy specifications for the DOM Tree pattern in §5.1.5. We choose the DOM Tree pattern because we are convinced of the importance of reasoning about unknown code using the object-capability framework in the web security space, where guest/unknown code (such as third party scripted advertising or widgets) are pervasive in the web today (details can be found in the work by Maffeis et al.[MMT10]). We show how our policies can specify the preservation of properties of nodes, in the presence of unknown code execution. Lastly, in §6, we offer some preliminary insights on the security properties of smart contracts on the Ethereum blockchain system that has experienced explosive growth in recent years, and we recognise that further work has to be done in this exciting new field.

1.2 Overview of Paper

In our paper, we first give an overview background history on the OCap model in §2. Readers familiar with the object-capability model and its languages are encouraged to skim through this section. In §3, we seek to formally specify the definitions of permission and authority and use our specifications to describe and reason how objects interact with each other. Our formal specifications are broad enough to reason in general about how two objects interact in our small specification language. That said, there are benefits to using these concepts to reason within an OCap system, which we demonstrate in §4, because the restrictions placed by OCap languages or what we call capability-safe languages allow us to enforce protection of objects more easily. We further solidify how our methodology can be used to reason convincgly in an OCap system by applying it on a DOM Tree pattern within an OCap framework, in §5.

2 Object-Capability 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[MS03]. 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 for an object to call the methods of another object, an object needs to explicitly obtain the reference of the object it is calling. 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[DVH66]'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[WCC$^+$74] into a protected ability to invoke arbitrary services provided by other processes. Capabilities were later implemented on computer and operating systems such as CAP[WN79], KeyKOS[Har85], and EROS[SSF99]. The development of capabilities and a detailed examination of the object-capability model with the capability-safe programming language E, can be found in Miller's PhD thesis[Mil06]. Capability-safe programming languages that allow building a system based on the object-capability model are:

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

2.2.1 Difference between capability and object-capability

In his seminal paper *Protection*[Lam74], Lampson defined protection as a general term for all the mechanisms that control the access of a program to other things in the system. Specifically, Lampson introduced the idea of an Access Control Matrix, an abstract, formal security model that describes the rights of each subject with respect to every object in the system, at a given point in time. The matrix contains one row per subject and one column per object, and each cell entry (for a particular subject-object pair) indicates the access mode that the subject is permitted to exercise on the object. Each column represents an "access control list" (ACL) for the object; and each row represents an "access profile" for the subject. Since a capability model associates each subject with a list of capabilities, it would appear that the model can be represented by a row-based view of the matrix.

However, in the technical report *Capability Myths Demolished*[MYS$^+$03], Miller et al. identified that a capabilities-as-rows interpretation assumes an ambient authority security property—subjects are not required to indicate a specific authority in order to exercise it—and in total identified seven security properties that capture the distinction between ACL, capabilities-as-rows, capabilities-as-keys, and object-capabilities. In the same report, Miller et al. argue that the "true" capability model *is* the object-capability model, because all known major capability systems take the object-based approach and capability-based systems are explicitly characterised as "object-based". The object-based model of computation can be seen as a dynamic reference graph of objects, where there is no distinction between subjects and objects. The object-capability model uses the reference graph as the access graph, requiring that objects can interact only by sending messages on references. An object's state therefore consists of both data and the capabilities—unforgeable object references that allow its holder to send messages to the object it references by invoking the capability.

2.3 Capability-safe Language Restrictions

In this section, we briefly mention three critical restrictions required in a capability-safe language: memory safety, no global mutable variables (objects can only interact with each other on the reference graph), and the taming of reflection.

2.3.1 No Capability Forging/Memory-safety

The object-capability model requires a capability to be *unforgeable*. Without memory safety in the programming language, a reference to a particular object in a memory can potentially be forged externally and anywhere. For example in C++, because the language allows memory manipulation, designation is not opaque and hence forgeable. Let us assume a pointer ptr that holds the designation of a newly created int object `int* ptr = new int()`. The int object can thus be accessed through the use of ptr. However C++ allows the programmer access to the memory and we can obtain the a string of bits that represent the memory address of the int object. Let us assume this memory address is represented by the bits 0x12345678. We say Designation is forgeable and accessible by bits because any object can perform the following operation:

```
int* ptr2 = reinterpret_cast<int*>(0x12345678)
```

to obtain a new pointer variable to the same int object. On the other hand, in java, object designations are completely opaque/indivisible–even with knowledge of the bits that constitute a pointer (obtained by Object.toString()); there is no operation for obtaining the corresponding pointer from this bit information and therefore designation cannot be forged.

2.3.2 No Global Mutable Variables

This restriction is necessary because if there are global mutable variables, then objects can now communicate without needing the object capabilities of each other. Global mutable variables can behave like a message carrier, such that two objects can both communicate by monitoring and modifying the state of the global variable. The capability-safe language Pony, which we use to show the code of OCap patterns, do not allow global variables.

2.3.3 No 'Untamed' Reflection

Reflection is the ability of a piece of code to "reflect" on the structure of other code in the same system or itself. More concretely in object-oriented systems, an object that can perform untamed reflection is about to examine and modify the code of other objects or itself. Take for example in Java, reflection allows code to perform operations that would otherwise be illegal in non-reflective code, such as accessing private fields and methods. Let us assume a simple Java program where object A possesses the reference capability of object B, and object B possesses the reference capability of object C. Object B stores the reference capability of object C in a private field and does not have any public method to divulge this data. With reflection, we cannot guarantee that object A will never obtain the capability of object C - this is because with the capability of object B, object A can reflect on object B's code and get all information stored in object B - including the private field that holds the reference capability of object C. In order to ensure capability-safety, the reflection feature in a language has to be 'tamed' or restricted. Joe-E[MWC10], a security oriented subset of Java, has such a feature.

2.4 Related Works

Drossopoulou et al. [DNMM15] propose that reasoning about object-capability patterns require extensions in the logics that talk about programs, and those extensions are not about what currently is in the heap, but about what a program might do in the future. They use predicates modelling trust and risk, and used those predicates to specify a capability-based version of Miller's escrow exchange example[MMF00]. Drossopoulou and Noble[DN13] first re-wrote the escrow exchange example in Joe-E and E, and in the process of writing the code, they argue that code written in existing capability safe languages is too focused on the low-level mechanism rather than higher-level security policies. Furthermore, the code required to describe such security policies tend to be interwoven and mixed with the parts implementing the functionality, resulting in these policies being *implicit*. Thus, they propose a specification language where policies are made explicit, to define policies required in the escrow exchange example[DN14, DNM15, DNMM15]. More recently, Drossopoulou et al.[DNMM16]'s work further introduced formalisations with regards to the distinction between permission and authority in the object-capability literature.

Maffeis et al.[MMT10]'s formal system defines authority in a topological manner where objects are represented as nodes in a graph, and a path between two nodes implies that the source node can access the destination node. Their work brings object-capability to web security, which is also a motivation for our paper's work on specifying the DOM Tree OCap pattern. They formalized concepts of capability safety and authority safety, and proved that capability safety implies authority safety, which in turn implies resource isolation. They demonstrated these guarantees in a Caja-based subset of JavaScript.

Devriese et al.[DBP16] present a formalisation of reasoning about object capabilities that is based on a Kripke logical relations model, in a language with higher-order state. Using a simple core calculus for JavaScript, they allow bounding the behaviour of a program fragment based on the capabilities it has access to, and use their model to verify and reason about several simple capability patterns, including the DOM Tree pattern.

Swasey et al.[SGD17] present a logics for OCP called OCPL, a formal system for compositionally specifying and verifying the security guarantees provided by OCPs, with a language that has higher-order functions, state, and concurrency. OCPL allows modular reasoning about OCP implementations and user code that depends on them, and to specify a general property on user code that ensures such code can be safely shared with untrusted code without having its internal invariants violated.

3 Formal specifications

In this section we describe a small object-oriented language in §3.1 and define predicates that allow us to specify object-capability policies in §3.2. For our language, we use the definitions of modules, classes, runtime state, interpretation, execution and arising configurations from the specification language in the appendix of [DNMM15]. To express more precisely how an object can interact with another object, we use the ideas of permission and authority in Miller's[Mil06] and Drossopoulou et al.'s[DNMM16] works. In their papers and our paper, we generally try to capture the informal meaning of:

- permission: access right of an object to invoke a behaviour of an object
- authority: ability to cause effects

In the approach of our paper however, we contribute to the literature by taking a novel approach in formally defining both permission and authority, though we argue they generally capture the same informal meanings. For permission, we have four different modes permissions specified by four MayAccess predicates. Given a module M and state σ,:

MayAccess^{Dir,Now}(o,o′): o has an object reference of o′
MayAccess^{Ind,Now}(o,o′): o contains a reference that points to an object that has a reference to another object—thus forming a series of object reference(s)— where the last object in the series has an object reference of o′
MayAccess^{Dir,Eve}(o,o′): there will be some eventual state σ′ arising from σ where M,σ′ ⊨ MayAccess^{Dir,Now}(o,o′) holds
MayAccess^{Ind,Eve}(o,o′): there will be some eventual state σ′ arising from σ where M,σ′ ⊨ MayAccess^{Ind,Now}(o,o′) holds

For authority, we propose a new MayCall predicate that deviates from the MayAffect predicate proposed in Drossopoulou et al.[DNMM16]. While their MayAffect predicate captures the idea of a field of an object being modified, our MayCall predicate is much weaker in the sense that mean the ability to invoke a method of an object *being*. Given a module M and state σ,:

MayCall(o,o′): there exist a method in o that can call a method in o′, such that there will be some eventual state σ′ arising from σ, where the receiver in the stack has transitioned from o in σ, to o′ in σ′

We further elaborate these predicates in §3.2.1 and §3.2.2.

3.1 Specification Language

In our small specification language, we briefly describe the defintions of module, class, runtime state, interpretation, execution, arising configurations, and assertion. **Module:**
A module maps class identifiers to class descriptions:

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

Class:
Class definitions describe how a class contributes to the runtime behaviour of a program for which we use its field and method declarations. We define method bodies as consisting a sequences of statements; these statements can be field read or internal field assignments (only allowed if the object is `this`), conditionals, and method calls. We also define all classes to be private which mean only objects belonging to class C may construct objects of class C.

```
ClassDescr   ::= [private] class ClassId
                          implements SpecId*
               { (field FieldId)*
                 (methBody)*
                 (FunDescr)*
                 (PredDescr)* }
methBody     ::= method m (ParId*)
               { Stmts; return Arg }
Stmts        ::= Stmt | Stmt; Stmts
Stmt         ::= var VarId := Rhs
               | VarId := Rhs
               | this.FieldId := Rhs
               | if Arg then Stmts else Stmts
               | skip
Rhs          ::= Arg.MethId( Arg* ) | Arg
               | new ClassId( Arg* )
Arg          ::= ParId | VarId | this
               | this.FieldId
```

Note that we work with all fields being private in our model - this restricts each object to being able to modify only its own fields.

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

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

Interpretation:
For a state $\sigma = (\varphi, \chi)$, we define the partial function:

$\lfloor \_ \rfloor_{\_}$: state $\times$ Path $\rightarrow$ Value

as follows:

$\lfloor \textbf{null} \rfloor_\sigma \ = \textbf{null}$
$\lfloor \textbf{true} \rfloor_\sigma \ = \textbf{true}$
$\lfloor \textbf{false} \rfloor_\sigma \ = \textbf{false}$
$\lfloor x \rfloor_\sigma \ = \varphi(x)$ for $x \in$StackId,
undefined, otherwise.
$\lfloor p.f \rfloor_\sigma \ = \chi(\lfloor p \rfloor_\sigma)(f)$ if $\lfloor p \rfloor_\sigma$ is defined,
and f is a field of $\lfloor this \rfloor_\sigma$
undefined, otherwise.

where,
$p \in$path ::= $x \mid p.f$

We also define the lookup of the class of an expression:
$\text{Class}(e)_\sigma = (\sigma\downarrow_2 \lfloor e \rfloor_\sigma)\downarrow_1$ if $\lfloor e \rfloor_\sigma$ defined
undefined, otherwise.

Execution:
Execution uses module M, and maps a runtime state σ and statements stmts onto a new state σ′. We keep our model simple by not giving execution rules for outcomes like null-pointer-exception or stuck execution. Execution of statements and expressions has the following shape:

$\leadsto$: Module $\times$ state $\times$ Stmts $\rightarrow$ state
$\leadsto$: Module $\times$ state $\times$ Rhs $\rightarrow$ heap $\times$ val

Arising Configurations:
We define Arising(M, σ) as the set of runtime configurations which may be reached during execution of some initial context (σ_0, stmts_0). The mappings of Init and Arising are as follows:

Init : Module $\rightarrow$ P(State $\times$ Stmt)
Arising : Module $\rightarrow$ P(State $\times$ Stmts)

We say a context is initial if its heap contains only objects of class Object, and initial configuration should be kept as minimal as possible, with a heap that has only one object and executing a method call on a newly created object with another newly created object as an argument:

$$
\begin{aligned}
&\mathsf{Init}\{(\mathsf{M},\sigma) = (\sigma_0,\ \mathbf{new}\ \mathsf{c.m}(\mathbf{new}\ \mathsf{c'}))\ |\ \mathsf{c,c'} \in \mathsf{dom(M)}\\
&\quad \mathsf{where}\ \sigma_0 = ((\iota,\mathbf{null}),\chi_0),\\
&\quad \mathsf{and}\ \chi_0(\iota) = (\mathsf{Object},\ \emptyset)\}\\
&\mathsf{Arising}(\mathsf{M},\sigma) = \bigcup_{(\sigma,\mathsf{stmts})\in \mathsf{Init(M)}} \mathsf{Reach}(\mathsf{M},\sigma,\mathsf{stmts})
\end{aligned}
$$

We also say a configuration is reachable from another configuration, if the former may be required for the evaluation of the latter after any number of steps. Reach takes the following shape:

$$
\begin{aligned}
&\mathsf{Reach} : \mathsf{Module} \times \mathsf{state} \times \mathsf{Stmts}\\
&\quad\quad \to \mathsf{P}(\mathsf{Stmts} \times \mathsf{state})
\end{aligned}
$$

The set Reach(M,σ,stmts) collects all configurations reachable during execution of σ,stmts. Note that the function Reach(M,σ,stmts) is defined even when execution diverges so it may be an infinite set. We therefore can give meaning to capability policies without requiring termination.

Assertion
The syntax of assertion can be found in the works of [DNMM15], where we have placed the relevant section in Appendix §9.2 of our paper , and is more or less as expected. Validity of assertion has the shape:

$$\mathsf{M},\sigma \models \mathsf{A}$$

and holds in context of a module M and runtime configuration σ, where more details can be found in the mentioned appendix section above.

We say that a class C satisfies an assertion A in the context of a module M, if in all runtime configuration σ' arising from execution of any module M' linked with module M, all objects of class C satisfies A:

$$\mathsf{M} \models \mathsf{C{:}A} \iff \forall \mathsf{M}',\sigma\in\mathsf{Arising}(\mathsf{M}{*}\mathsf{M}',\sigma).\ [\ \mathsf{M}{*}\mathsf{M}',\sigma \models \mathsf{x{:}C} \implies \mathsf{A[x/this]}\ ]$$

We define linking of modules, $M{*}M'$ to be the union of their respective mappings, provided that the domain of the two modules are disjoint with respect to classes.We need module linking to allow us to reason about policies in the execution of unknown code, where module M' can be unknown. More details on module linking can be found in Appendix §9.1.

3.2 Permission and Authority

The concepts of permission and authority are central to reasoning about object-to-object interactions, and are especially important for reasoning about OCap patterns. Miller first proposed the concepts of permission and authority in his PhD thesis[Mil06], following which the formal definitions of these concepts were proposed and refined in the work of Drossopoulou et al. [DNMM16]. Our paper defines permission and authority in a formal way that is different from the aforementioned works, but we generally adopt the same informal meanings (that having permission means the right to invoke an object's behaviours, and authority to mean the ability to cause effects).

3.2.1 Permission - MayAccess Definition

We define in total four flavours of MayAccess predicates that are inspired by the work in Drossopoulou et al.[DNMM15], but our four MayAccess predicates extend and modify the definition in [DNMM15] 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 general enough to describe generic object-oriented systems:

Definition—[MayAccess]

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

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

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

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

---

where, we say 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 MayAccess^{Ind,Now}(x,e) holds *iff* in some current state σ, we have a *series of fields* from 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.

Table 1: MayAccess Relations

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

We surmarise the relationships between the four flavours of MayAccess in table 1. Note that in an object-to-object paradigm, all four flavours of MayAccess involve object capabilities. $\text{MayAccess}^{Dir,-}(o,o')$ says object o has the direct reference of o', while $\text{MayAccess}^{Ind,-}$ says object o has a path of reference(s) that lead to o' (that is, o has the reference of some object x_1, and object x_1 has the reference of object x_2 ... and object x_n has the reference of object o'). The 'now' and 'eventual' flavours simply state that the "direct" and "indirect" predicates either they hold at σ (Now) or they can hold at some arising eventual state σ' from σ (eventual).

3.2.2 Authority - MayCall Definition

With our access predicates in place, we define a MayCall predicate that describe what it means for x to be able to call e. Note that our definition of MayCall is different from the MayAffect predicate in [DNMM16]. Our MayCall predicate captures the meaning of an object o having authority over another object o' without o necessarily modifying the fields of o'. In that sense, our MayCall predicate is weaker than the MayAffect predicate in [DNMM16]:

Definition—[MayCall]

$$M,\sigma \models \text{MayCall}(x,e) \iff$$
$$\lfloor \texttt{this} \rfloor_\sigma = \lfloor x \rfloor_\sigma \land \exists \sigma' \in \text{Arising}(M,\sigma).\ \land \lfloor \texttt{this} \rfloor_{\sigma'} = \lfloor e \rfloor_{\sigma'}$$

where we say MayCall(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'.

Definition—[MayAffect from Drossopoulou et al.[DNMM16]]
M,σ $\models$ MayAffect(x,e) if there exists a method m, argument(s) $\bar{a}$, state σ', such that:
M,σ e.m($\bar{a}$) $\rightsquigarrow$ σ', and
$\lfloor e \rfloor_{M,\sigma} \neq \lfloor e' \rfloor_{M,\sigma'}$

Motivation for MayCall: The motivation for defining authority as MayCall in our paper is that we would like to describe a *chain* of authority between objects, without needing objects in such a chain to necessarily modify the fields of other objects in the chain. In other words, when object A wants to use object B to modify a field in C but can only do so through object B, we want to express that we can deny A from modifying C either by denying A from calling B, or deny B from calling C. Note that object A does not need to be able to modify the fields of B to eventually modify the state of C. Hence, the MayAffect predicate used in [DNMM16] is too strong to decompose a long chain of authority into smaller object-to-object authorities.

Lemma—[Object as receiver requires method invocation]
M,σ $\models$ σ(this) = $\lfloor o \rfloor_{\sigma} \Longrightarrow$
$\exists\sigma'$,m,$\bar{a}$. [M,σ', o.m($\bar{a}$) $\rightsquigarrow$ σ]

We 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.

Our MayCall(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 has authority of 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. Hence, the question of whether an *untrusted* object can modify a private field in an object o′ can be expressed as a question of whether the untrusted object can invoke a method of o′. Given our definitions of MayCall(o,o′), it follows that a necessary condition for modification of a private field in o′ is the invocation of a method of o′. Proving the falsity of our MayCall(o,o′) predicate is therefore sufficient to prove that for a field f in o′, MayAffect(o,o′.f) does not hold.

Lemma—[Field Modification Requires Authority]
M,σ $\models$ $\forall\sigma'$,f. [$\sigma'\in$Arising(M,σ) $\wedge$ $\lfloor o'.f \rfloor_{\sigma'} \neq \lfloor o'.f \rfloor_{\sigma} \Longrightarrow$
$\exists$o. M,σ $\models$ MayCall(o,o′)]

3.3 Domination

Here, we introduce the concept of domination that is inspired by the work on ownership types by Clarke et al.[CPN98], 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.

We say that a set S dominates an object x *iff* for every object y in the system that has a path to x, y must have a path to some arbitrary object o that is a member of the set S. The power of having such a predicate will be demonstrated in §5, where we show how attenuating objects that is meant to protect an object o can ensure certain properties of members in a dominating set over s under any unknown code execution.

Definition—[Domination]

$$M,\sigma \models \mathsf{Dom}(S,x) \iff$$
$$\forall y.\bar{f},n.\ [\ x \neq y \land \lfloor y.f_1...f_n \rfloor_\sigma = \lfloor x \rfloor_\sigma \implies$$
$$\exists k,o.\ \lfloor y.f_1...f_k \rfloor_\sigma = \lfloor o \rfloor_\sigma \land o \in S\ ]$$

The usefulness of domination will be made more clear in the next section when we argue about cooperation without vulnerability.

4 Reasoning in an OCap System

In this section, we formally describe the ideas of *isolation* and *cooperation*, and how *cooperation* can either be vulnerable or protected, within the OCap system. We use the word *capability* (of an object), interchangeably with reference (of an object), to represent that both object reference and the right to use the object reference are indistinguishable in OCap; i.e., there is no global ambient authority to prevent an object from using another object reference in general. Given our definition of *permission*, it follows that o has the permission of o′ if o has the capability of o′. However, if we were to use *our* formal definition of *authority* (causing effects where effects are method invocations), we cannot say that an object reference embodies authority - we elaborate more in §4.2. We use the word *path* interchangeably with indirect capability or permission, to emphasise that an object o need not necessarily require the direct capability of o′ to invoke the behaviours of o′—invocation of o′ can be done by o through a network of paths consisting of other object capabilities.

In §4.1 we first explain that objects require permission (direct capabilities or a chain of capabilities) to communicate, i.e. permission forms the necessary condition for authority. We also propose a lemma that states that under an OCap system, we can reason about eventual indirect permission, from a current permission configuration of an initial closed system[3]. That

[3]We require a closed system because unknown object or code might introduce capabilities of objects in the system that the unknown object or code might already have.

is, given a system where we know the entire system's present reference graph, OCap rules confine the possible reference graphs that can be born eventually from the present reference graph. Next in §4.2, we reason about *isolation* and *cooperation* using our formal definition of permission. Following which in §4.3, we expand the concept of *cooperation* to include cooperation that is vulnerable, and cooperation that is protected, i.e. *vulnerability* and *protection*. *Protection* and *vulnerability* are built on our formal definitions of authority and domination.

4.1 Eventual permission from present permission configuration

Here, we propose a lemma EPC that states how eventual permission can be born from a constellation of direct permissions within an OCap system. We use paths interchangeably with direct and indirect permission, to emphasise that an object o need not necessarily have the direct capability of o′ to invoke the behaviours of o′, invocation can be done through a network of paths consisting of permissions.

Lemma—[OCap Eventual Paths from Current Paths (EPC)]

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

LEMMA 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. EPC is actually a formal representation of the well-known OCap idiom of *connectivity begets connectivity*:

- **Initial Conditions or Endowment:** o has the capability of o′ in σ due to initiate conditions or endowment in the system, 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 some object that is not o (x≠o).

Note that for o to have an eventual path to o′, we only require now that o have 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. 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 x′? Note that in such a case, x′ must have the capability of object x (for x′ to even introduce x to o in the first place), and therefore x′ can also have an eventual path to o′. Also x′ must also be able to introduce itself to o. With these observations, we note that there is hence no logical difference between x′ and x in our formal description and x′ might simply be referred to as x.

There is one final critical result from LEMMA EPC. Notice how, there is a 'recursive' MayAccess^{Ind,Eve}(x,o′) in our condition for 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 MayAccess^{Dir,Now} predicates, where the terminating MayAccess^{Ind,Eve}(o′,o′) can be determined to be true or false based on whether o′ exists in some arising state. 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 show an example using LEMMA EPC with 3 objects o, y, o′. Let us assume o, y, 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 (where x refers to the intermediate object in EPC):

1) o has the capability of y, and y has an eventual path to o′—
by y having the direct capability of o′
2) o has the capability of y, and y has an eventual path to o′—
by o′ having the direct capability of y
3) y has the capability of o, and y has an eventual path to o′—
by y having the direct capability of o′
4) y has the capability of o, and y has an eventual path to o′—
by o′ having the direct capability of y
5) o has the direct capability of o′
6) o′ has the direct 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.

4.2 Isolation and Cooperation

In this section, we reason about isolation and cooperation using our formal definition of permission and authority. What exactly is object isolation when we say that object o′ is isolated from object o, and what exactly is cooperation?

We say isolation is simply that there is no eventual directed path from o to o′:

Isolation of o′ from o:
$\models \neg$MayAccess^{Ind,Eve}(o,o′)

Given LEMMA EPC, we can enforce isolation of o′ from o given the present path configuration of a system at σ, and when the closed system is eventually opened such that an unknown object has permission and authority of *only* object o in some arising state σ′, we know that the unknown code will not have permission of o′, since o is isolated from o′.

When we have isolation, we know from contraposition of the corollary from our definitions of permission and authority:

Corollary—[Permission(Indirect,Eventual) is necessary for Authority]
$M,\sigma \models \mathsf{MayCall}(o,o') \implies$
$\quad \mathsf{MayAccess}^{Ind,Eve}(o,o')$

Therefore,

Isolation implies no authority
$\models \neg\mathsf{MayAccess}^{Ind,Eve}(o,o') \implies \neg\mathsf{Maycall}(o,o')$

It is good to know we can guarantee the internal state and behaviour of an object o′ in the presence of unknown code on object o through isolation, but in practice this is not entirely useful. The quintessential question in OCap systems is whether we can allow cooperation without vulnerability. What then is cooperation?

We say cooperation between two objects o and o′ is simply that o has permission of o′:

Cooperation between o and o′ (o → o′):
$\models \mathsf{MayAccess}^{Ind,Eve}(o,o')$

Cooperation is necessary but not sufficient for authority:
$\models \mathsf{MayAccess}^{Ind,Eve}(o,o') \not\Longrightarrow \mathsf{MayCall}(o,o')$

We provide some analogies as to why cooperation does not imply authority. Object o can have the reference(capability) of o′, and hence there is cooperation, but o contains no methods that can invoke(use) the capability of o′, and thus o has the capability of o′ but not the authority of o′. Another analogy, from the other viewpoint of o′, would be that o′ has no available methods that can be invoked by o —this can happen if o has no methods, or o only contains private methods or methods that are restricted for objects inherited from the same class as o of which o′ is not.

The problem with cooperation is when we know o′ has exposed methods (or public methods), and when object o is of unknown provenance. Since we assume object o is unknown, then we must assume that given the permission of o′, o will contain some method that will try to invoke the behaviours of o′. This then leads us to our next section on vulnerability and protection in the context of cooperation.

4.3 Vulnerability and Protection

- We assume that object o′ will always have some exposed public method for the rest of this section, and is the object of interest that we want to protect.

For the concepts of vulnerability and protection, our arguments are based primarily on whether the source object is trusted (known) or untrusted (unknown).

While we can always protect o′ through isolation from o (o will have no paths to o′), what is probably more useful is whether cooperation between o′ and o, is possible in a *protected way*. What does that mean? Notice that if we *know* or *trust* the code of o, then **cooperation can actually be safe**, even if o contains methods that can eventually invoke behaviours of o′.

Lemma—[Known-Safe cooperation]
o:KnownObj $\wedge$ o′:ExposedObj $\wedge$ MayAccess^{Ind,Eve}(o,o′)

Then regardless of whether MayCall(o,o′) holds or not, since we know or trust the code of o, we know that o will not do something that will lead to undesirable method invocations of o′.

However, if we have another object o* that is of unknown provenance such that:

Lemma—[Unknown-Vulnerable cooperation]
o*:UnknownObj $\wedge$ o′:ExposedObj $\wedge$ MayAccess^{Ind,Eve}(o*,o′)

---

then we must always be prepared for the worst in terms of what o* will do, and cooperation becomes vulnerable if o* can invoke behaviours of o such that MayCall(o*,o′) holds.

So, this then leads us to the most important question, how do we enable safe cooperation in the face of unknown code?

4.3.1 Attenuating objects as trusted messengers

Notice how we already have a result about a Known-Safe cooperation. In this section, we show how can make use of it, and four corollaries from our formal specifications, including the concept of domination that we have proposed.

Corollary—[Indirect permission is made of direct permissions]
M,σ ⊨ [$\mathsf{MayAccess}^{Ind,Now}$(o,o′)
    ⟺
    ∃x. [$\mathsf{MayAccess}^{Ind,Now}$(o,x) ∧
        $\mathsf{MayAccess}^{Dir,Now}$(x,o′)]]

Corollary—[Authority can be a chain of smaller authorities]
M,σ ⊨ MayCall(o,o′)
    ⟹
    ∃x. [MayCall(o,x) ∧ MayCall(x,o′)]

The above two corollaries tell us that we do not need to give a direct permission or capability of o′ to the unknown object o* for cooperation to occur between the two objects. Cooperation between o* and o′ is possible with *intermediate* objects.

Corollary—[Domination of Permission]
M,σ ⊨ $\mathsf{MayAccess}^{Ind,Now}$(o,o′) ∧ Dom(S,o′)
    ⟹
    ∃x∈S. [$\mathsf{MayAccess}^{Ind,Now}$(o,x) ∧ $\mathsf{MayAccess}^{Ind,Now}$(x,o′)]

Corollary—[Domination of Authority]
M,σ ⊨ MayCall(o,o′) ∧ Dom(S,o′)
    ⟹
    ∃x∈S. [MayCall(o,x) ∧ MayCall(x,o′)]

The above two corollaries on domination tell us that if we know a set S that dominates our object of interest o′, then an unknown object o* *must* use a member of this dominating set in the path of permissions to reach o′ and in the chain of authority to invoke o′.

With these four corollaries, and from knowing that cooperation between trusted code is safe, we have:

Lemma—[Unknown-Protected cooperation]
o*:UnknownObj ∧ o′:ExposedObj ∧ $\mathsf{MayAccess}^{Ind,Eve}$(o*,o′)
∧ Dom(S,o′) ∧ ∀s. [s∈S ⟹ s:KnownObj]

where objects s are known as attenuating objects in the OCap literature. To see why Unknown-Protected cooperation is safe, see how it follows that:

Lemma—[Unknown-Protected cooperation implies Known-Safe cooperation]
$\mathsf{o^*{:}UnknownObj} \land \mathsf{o'{:}ExposedObj} \land \mathsf{MayAccess}^{Ind,Eve}(\mathsf{o^*,o'})$
$\land\ \mathsf{Dom(S,o')} \land \forall \mathsf{s}.\ [\ \mathsf{s}{\in}\mathsf{S} \implies \mathsf{s{:}KnownObj}\ ]$
$\implies \exists \mathsf{s'{:}KnownObj}.\ [\ \mathsf{MayAccess}^{Ind,Eve}(\mathsf{o^*,s'}) \land \mathsf{MayAccess}^{Ind,Eve}(\mathsf{s',o'})\ ]$

This says that if we have a configuration where object o′ is dominated by known objects, then the permission to use o′ by o* must involve at least a known object. This shows that we can use attenuating objects to mitigate vulnerability between unknown code and known code in OCap systems.

Therefore, this shows that we can always protect some object of interest by:

- being careful about our present path configurations so that we can ensure eventual path-isolation between the untrusted object and the object of interest
- ensuring that trusted objects dominate the object of interest and attenuates the permission and authority of the untrusted 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 authority *cannot* enforce security through pure path isolation, which is a distinctive feature of OCap models.

Protection strategies for attenuating objects can be implemented using defensive programming techniques like encapsulation and checking of conditions of method arguments. At the extreme ends, attenuating objects can take the shapes of either deciding to forward messages from the unknown object o* in an intact way, thereby allowing MayCall(o*,o′) to hold; or if some condition fails, the attenuating object can choose not to forward the message at all such that MayCall(o*,o′) does not hold. Somewhere in the middle, attenuating objects can also choose to modify the message from o* such that it becomes safe to forward, where MayCall(o*,o′) holds but since it has been attenuated by the attenuating object, we would know it is safe. What and how the attenuating objects decide to do with messages from unknown objects before forwarding them can be best expressed as OCap policies for the attenuating object.

In the OCap pattern examples we will go through in the next section, protection of an object is accomplished through some attenuating object that will never leak the capability of the object of interest. A safe configuration is also expressed as attenuating objects dominating the object of interest in such a way that unknown code is forced to use them to cooperate.

5 OCap Patterns

An OCap pattern is a a set of objects connected to each other by capabilities in such a way that some effects can only happen under certain conditions. 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′. Indeed, in §5.1.4, §5.3.1, and §5.3.2, we represent such diagrams for three important OCap patterns in the literature. These three OCap patterns are: the DOM Tree pattern, the Caretaker pattern, and the Membrane pattern. In addition to the visual diagrams of these patterns, we also show the code of these three patterns in the capability-safe language Pony[CDBM15]. To make use of our formal specifications, we focus exclusively on the DOM Tree pattern, where we show how we can use our formal specifications to write policies that can guarantee the preservation of properties of specific nodes in the pattern.

5.1 The DOM Tree Pattern

According to the World Wide Web Consortium (W3C)[4], the Document Object Model (DOM) is a platform- and language-neutral interface that allow programs and scripts to dynamically access and update the content, structure and style of documents. A DOM Tree describes a model where objects in a system are arranged according to some hierarchical configuration in the structure of a tree. In this pattern, permission and authority of nodes can be reasoned according to whether one object can traverse up or down the tree to access or call other nodes in the tree. This type of reasoning is useful when we need to protect the integrity of more sensitive objects higher up in the hierarchy of the tree, while at the same time, we need to allow untrusted external objects the ability to affect less-sensitive objects lower in the hierarchy of the tree. Our contribution in our paper is that we implement the pattern and give the code for the pattern classes Nodes and ReNodes in the capability-safe language Pony. Another contribution of ours is that provide the formal specifications of such a pattern using our methodology, which then allows us to assert conditions when properties of a Node in the tree are modified.

We first illustrate the vulnerability of a Node object in the Tree in §5.1.1. Subsequently, we show how we can create an attenuating object ReNode that protects access to a Node, in §5.1.2. In §5.1.3, we show the code of a simulation where we have a JavaScript web document (simulated in Pony) where we need to give some arbitrary 3rd party advertisement company external access to a particular node called AdNode in the document (so that the website is able to run advertisements and make money). In the simulation, we show how using an attenuating restricted node RAdNode with a configuration of depth = 0, is able to protect all other nodes higher in the hierarchy of the DOM Tree, while *still* allowing the 3rd-party advertiser the ability to modify properties of the AdNode. We then show a graphical illustration of AdNode and RAdNode simulation fig. 1. Following which, we write the object-capability policies of the DOM Tree pattern using our methodology, in §5.1.5. Lastly, we show how we can begin to reason the

[4]https://www.w3.org/DOM/

protection of a node using those policies.

5.1.1 Vulnerability of Node in Tree

We call objects in the hierarchical tree to be a Node, and by default, nodes are vulnerable because they have unprotected methods that can access other nodes in the tree. In particular, the methods parent() and getChild(id:String) return the object references of other nodes in the tree.

```
class Node
    let name: String
    let _parent: (Node ref|None)
    let _children: collections.Map[String val, Node ref] = _children.create()
    let _props: collections.Map[String val, String val] = _props.create()

   new ref create(name':String, parent': Node ref)=>
       name = name'; _parent = parent'
   new ref createRoot(name':String)=>
       name = name'; _parent = None
   fun ref getChild(id:String val): Node ref?=>
       try _children(id) else error end
   fun ref parent(): Node ref?=>
       match _parent
       | let x: Node ref => x
       else error end
   fun box getProp(id:String val):String val ?=>
       try _props(id) else error end
   fun ref setProp(id:String val,prop:String val) =>
       _props(id)=prop
   fun ref addChild(id:String val):Node?=>
       _children(id) = Node.create((name+"-child-"+id.string()), this)
       _children(id)
   fun ref delChild(id:String val) ?=>
       try _children.remove(id) else error end
```

Internal state modification of a node is possible because the class Node contains a public method setProp(key,value) that can be called by anyone holding a node's capability. The method will modify the internal state properties of the node (by creating or modifying a key-value pair in a map within the node). Consequently, this means that having the direct capability of a node implies having the authority to modify the node's properties. Also, since having a direct capability of a node can eventually translate to having the direct capability and authority over all other nodes in the tree, this means that an untrusted object o that obtains access to a node n can eventually obtain authority over *all* other nodes n' in the same tree:

$$\forall \text{o:Object}, \forall \text{n,n':Node}. \ [\ \text{MayAccess}^{Dir,Now}(\text{o,n}) \wedge \text{MayAccess}^{Ind,Now}(\text{n,n'}) \implies \text{MayCall(o,n')}\ ]$$

In the OCap literature, letting an object o have access to an object o′ (o having the reference of o′) *usually* implies we want to let object o use or call methods on o′. More precisely, in such a configuration, we are typically prepared to let o have the ability to modify the state of o′. However, there are situations where object o′ has not just the ability to modify its own internal state, but in addition, the ability to call methods and modify the state of other objects o″. This then becomes problematic, because while we are typically prepared to let o modify the state of o′, we typically are **not** prepared to give o the authority to modify the state of all other objects o″ that are reachable from o′.

Nodes are problematic precisely because of the same reasons described in the general case above, because while we are typically prepared to give some unknown and untrusted object the authority to modify the state of a particular node n, we are **not** prepared to give the untrusted object the authority to modify all other nodes reachable from n. Notice the implication MayCall(o,n′) where n′ refers to all other nodes in the tree, such that the untrusted object can now call all other nodes in the tree, which is an extremely undesirable outcome in terms of security. Therefore, we need to find ways where we can give to an untrusted object o access and authority to a node n so that o can modify or call methods on node n, *but* at the same time, we do *not* compromise the properties or state of other nodes in the tree.

5.1.2 Protection using Restricted Node

In the OCap literature, the use of attenuating objects help to mediate authority of a protected object, and help to remedy the security problem we have just described above. In the DOM Tree pattern, we make use of an attenuating restricted node class ReNode whose objects serve to attenuate the authority of objects belonging to class Node. A restricted node rn of class ReNode attenuates the capability of a node of class Node in the following ways:

- A restricted node rn of class ReNode contains a field which points to a node n of class Node that rn is meant to attenuate, and contains a depth field d of type integer.
- It protects n such that with an untrusted object having only the capability of the restricted node rn, rn **only allows** the properties of n, all of n's descended child nodes in the tree, and all of n's ancestor parent nodes up to depth d in the tree, to be modified.

The restricted node accomplishes this primarily by specifying the conditions on which method calls are allowed to be *forwarded* to n, and specifying how the results of those method calls to n should be *processed*. Specifically, the sensitive parent() method in n that returns the direct capability of the parent node of n, is now protected by the restricted node rn. By calling parent() method on rn, rn will only forward the parent method call to n **only** when the condition that rn's depth level stored in the integer variable d must have a value of at least more than 0. Even if rn has a depth level more than 0, the restricted node does not return the direct capability of the parent node of n to the method caller on rn, but instead *processes* the result to the method caller on rn, by only returning a new restricted node object rn' with a field that points to the parent node of n, where rn' inherits the depth level of rn decremented by 1. Lastly, the restricted node does not have any methods that return the direct capability of the node that it protects and all the node's children. Method calls done by the restricted node on the underlying node that return the capabilities of the child nodes are always processed by the restricted node such that the restricted node only returns newly created restricted nodes of those child nodes configured with a depth level incremented by 1.

```
class ReNode[5]
    let \_node: Node ref
    let _d: U32
    new ref create(node':Node ref, d':U32)=>
        _node = node'; _d = d'
    fun ref getChild(id:String val):ReNode ref?=>
        try ReNode.create(_node.getChild(id), _d+1) else error end
    fun ref parent():ReNode ref?=>
        if \_d > 0 then ReNode.create(\_node.parent(), \_d-1) else error end
    fun box getProp(id:String val):String val ?=>
        try _node.getProp(id) else error end
    fun ref setProp(id:String val,prop:String val)=>
        _node.setProp(id, prop)
    fun ref addChild(id:String val):ReNode ref?=>
        try ReNode.create(_node.addChild(id), _d+1) else error end
    fun ref delChild(id:String val)?=>
        try _node.delChild(id) else error end
```

5.1.3 DOM Tree Example

We will now demonstrate how ReNode attenuates authority over Node. We assume we only have two nodes in the tree, where the root node n′ has a child node n. In this example, we want to allow an untrusted object o the authority to modify the properties of n, but not the root node n′. To illustrate this, we use a DOM Tree and call the root note n′ a document node, and the child node n an adnode. Typically, a website that wants to make money from having advertisements will have a particular node (adnode) within the DOM Tree, so that a dynamic advertisement object can be rendered properly in the document. Suppose that the website owner is called Alice and she is also required to give an external party (advertisement company) the authority to modify the properties of adnode in her HTML document. Obviously, Alice will not want the external party to modify the properties of the other nodes in her document—she would want to only allow the external party the authority to modify the adnode and its descended nodes. Alice therefore needs a way to protect the other nodes in the DOM Tree from the external party, but still allow the untrusted external party the authority to modify adnode.

```
actor Main
    let env: Env
    let main: Main tag
```

[5]In the Pony language, a field is declared private with a leading underscore "_", so we know that the capability of the protected node stored in a restricted node cannot be read directly by external objects using the restricted node, and can only do so through some method. Also, a field is declared using "let" to restrict the field from being modified. Therefore, we know that the node that a restricted node protects cannot change.

```
    new create(env':Env)=>
        env = env'
        main = this
        let document: Node = Node.createRoot("Document")
        let access: Bool = initWebPage(document, {(radnode: ReNode ref)=>
                try radnode.parent().setProp(" title ", "Bob website") end
                }ref)
        env.out.print(access.string())

fun ref initWebPage(document: Node, ad_lambda:{(ReNode)}):Bool val=>
    try
        document.setProp("title","Alice website")
        let adnode = document.addChild("ad_div")
        let radnode = ReNode(adnode, 0)
        ad_lambda(radnode)
        if document.getProp("title") is "Alice website" then return true
            else return false end
    else return false end
```

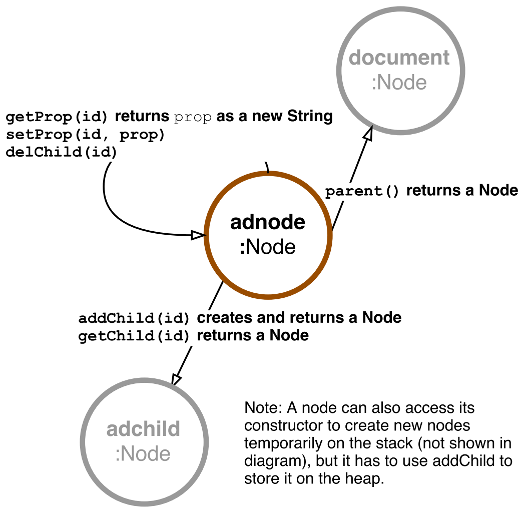
In the DOM Tree -adnode- code example above, we simulate initialising a webpage with a root document node that we set the property of a map-key "title" having a map-value of "Alice website". We then create a node for an advertisement called adnode. To allow the external party the authority to modify the properties of adnode, we create a restricted node object radnode with a depth level of 0 that points to adnode. In the simulation, we execute a method ad_lambda that attempts to call code on radnode, where in our simulation, ad_lambda attempts to modify the root Document node's properties by attempting to change the value of the root node's map-key "title", to "Bob's website". This would fail, because we have constructed radnode with a depth level of 0, so we know that radnode will not be able to traverse up to the root node of the tree. Therefore, we know the title of the rootdocument node remains intact as "Alice website" after execution of ad_lambda.

5.1.4 DOM Tree Pattern Illustration

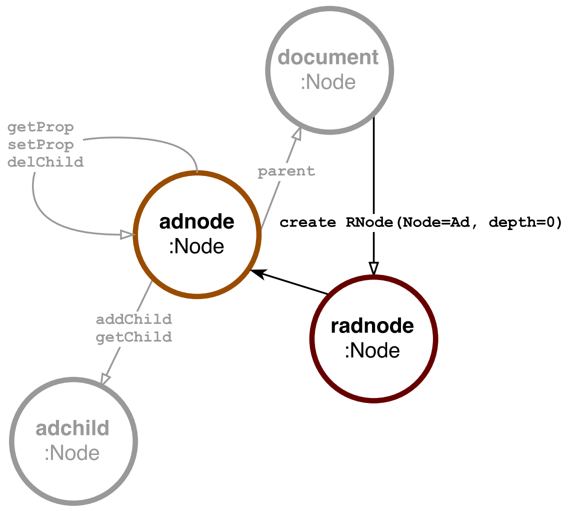
Here, we illustrate in more detail the example of the DOM Tree pattern in a visual diagram.

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 **adnode** to a third-party is unsafe, because using the parent() function call on the **adnode** 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.

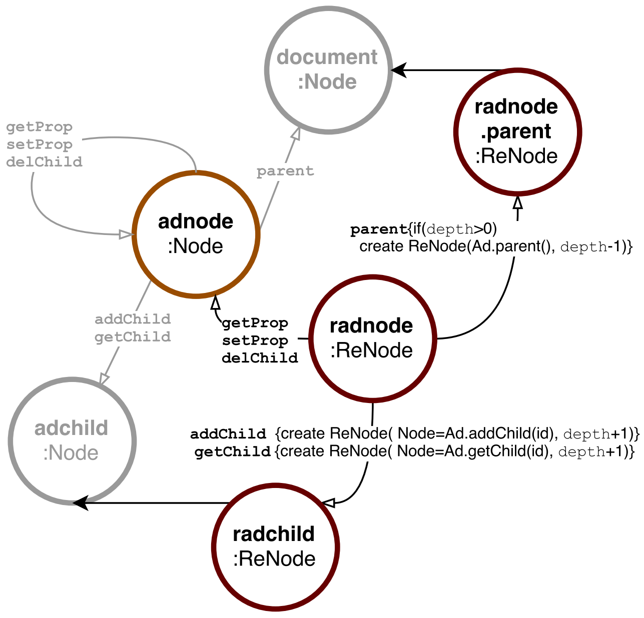


document :Node
getProp(id) returns prop as a new String
setProp(id, prop)
delChild(id)
parent() returns a Node
adnode :Node
addChild(id) creates and returns a Node
getChild(id) returns a Node
adchild :Node
Note: A node can also access its constructor to create new nodes temporarily on the stack (not 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
radnode .parent :ReNode
getProp
setProp
delChild
parent
adnode :Node
parent{if(depth>0) create ReNode(Ad.parent(), depth-1)}
addChild
getChild
getProp
setProp
delChild
radnode :ReNode
adchild :Node
addChild {create ReNode(Node=Ad.addChild(id), depth+1)}
getChild {create ReNode(Node=Ad.getChild(id), depth+1)}
radchild :ReNode

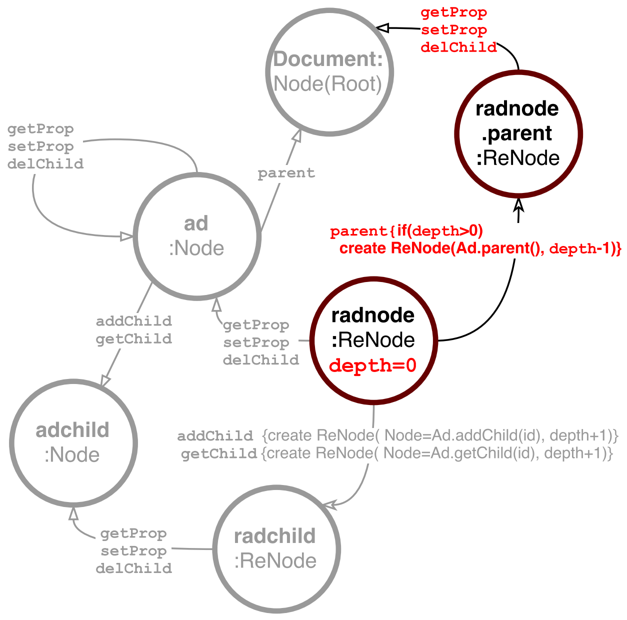


Figure 1: DOM Tree Example

5.1.5 DOM Tree Policies

Here, we provide the policies for the DOM Tree, using our methodology and Hoare tuples. The use of ":" indicates the variables of the LHS of ":" belonging to a primitive type or class stated in the RHS of ":". Before we dive into the policies, we define the lookup of the k-th parent of a restricted node as follows:

Definition —[ReNode.parent]

Given rn,rn′:ReNode, k:ℕ, then [$\text{rn.parent}^{k} = \text{rn}' \implies (k > 1 \wedge \text{rn.parent.parent}^{k\text{-}1} = \text{rn}') \vee$

$(k = 1 \wedge \text{rn.parent} = \text{rn}') \vee$

$(k = 0 \wedge \text{rn} = \text{rn}')$]

This definition says that given a superscript ^{k} placed on a parent, it means that the result on the RHS of such a look up is the restricted node's parent on the LHS being called k multiple times; if rn' refers to the same restricted node as rn, then k = 0.

5.1.5.1 Policies for calling and modifying nodes

The following two policies describe the condition under which an Object or ReNode may call methods on a node.

Policy —[Object calls Node]

$\forall$o:Object, $\forall$n:Node. [MayCall(o,n) $\wedge$ Dom(S,n) $\wedge$ $\forall$s. [s$\in$S $\implies$ s:ReNode]

$\implies$

$\exists$rn:ReNode. [rn$\in$S $\wedge$ MayCall(o,rn) $\wedge$ MayCall(rn,n)]]

Policy —[ReNode calls Node (Necessary Condition)]

$\forall$rn:ReNode, $\forall$n:Node. [MayCall(rn,n) $\implies$ $\exists$rn′:ReNode. [MayCall(rn,rn′) $\wedge$ rn′.node = n]]

Policy —[ReNode calls Node (Sufficient Condition)]

$\forall$rn:ReNode, $\forall$n:Node. [rn.node = n $\implies$ MayCall(rn,n)]

Policy [Object calls Node] says that if we have an object o that can call the methods of node n, and we have a set S that dominates n such that all members in S are of class ReNode, then it implies that object o must be able to call methods of a restricted node rn out of S and that rn must able to call methods of n. This just means that the chain of authority from o to n must involve rn in the chain.

Policy [ReNode calls Node] says something about the last restricted node rn′ at the end of the chain of authority to n: that a restricted node rn can call n by calling another restricted node rn′ that points to n. That is, a restricted node rn that points to a node n (rn.node = n)

5.1.5.2 Policy for restricted node calling other restricted nodes

Policy —[ReNode calls ReNode (Necessary Condition)]

$$
\begin{aligned}
&\forall \mathsf{rn},\mathsf{rn}'{:}\mathsf{ReNode}.\ [\ \mathsf{MayCall}(\mathsf{rn},\mathsf{rn}')\\
&\qquad \Longrightarrow\\
&\qquad \exists \mathsf{k},\mathsf{j}{:}\mathbb{N}.\ [\ (\mathsf{rn.parent}^{\mathsf{k}} = \mathsf{rn}'.\mathsf{parent}^{\mathsf{j}}) \wedge \mathsf{rn.depth} \geq \mathsf{k}\ ]\ ]
\end{aligned}
$$

Policy [ReNode calls ReNode] specifies a necessary condition and a sufficient condition for which a restricted node rn may call another restricted node rn′. This policy is subtle and says that a restricted node rn is allowed to navigate to all the descendants from all its accessible ancestors. *k* is a natural number that indexes all the accessible ancestors of rn *k* levels up the tree from rn, while we use another natural number *j* that describes all the descendant nodes *j* levels down the tree from a particular accessible ancestor. If rn′ is a direct descendant from rn, then k = 0. If rn′ is a direct ancestor of rn, then j = 0. An ancestor *k* levels up the tree is only accessible if rn has sufficient depth access (rn $\geq$ k); there are no restrictions on accessing the descendants *j* levels down the tree from a particular accessible ancestor.

5.1.5.3 Policy for reasoning about new paths from ReNodes

Policy —[Domination No Leaked Paths from ReNode]

$$
\begin{aligned}
&\forall \mathsf{n}{:}\mathsf{Node},\ \mathsf{RND}_{old}.\ [\ \mathsf{Dom}(\mathsf{RND}_{old},\mathsf{n}) \wedge \forall \mathsf{s}.\ [\ \mathsf{s}\in\mathsf{RND}_{old} \Longrightarrow \mathsf{s}{:}\mathsf{ReNode}\ ]\\
&\qquad\qquad \{\ \mathtt{code}\ \}\\
&\exists \mathsf{RND}_{new}.\ [\ \mathsf{Dom}(\mathsf{RND}_{new},\mathsf{n}) \wedge \forall \mathsf{s}'\in\mathsf{RND}_{new}.\ [\ \mathsf{s}'{:}\mathsf{ReNode} \wedge\\
&\qquad \exists \mathsf{s}''\in\mathsf{RND}_{old},\ \exists \mathsf{k},\mathsf{j}{:}\mathbb{N}.\ [\ (\mathsf{s}'.\mathsf{parent}^{\mathsf{k}} = \mathsf{s}''.\mathsf{parent}^{\mathsf{j}}) \wedge (\mathsf{s}''.\mathsf{depth} \geq \mathsf{j}) \wedge\\
&\qquad\qquad (\mathsf{s}'.\mathsf{depth} = \mathsf{s}''.\mathsf{depth} - \mathsf{j} + \mathsf{k})\ ]\ ]\ ]\ ]
\end{aligned}
$$

This policy is a very important security policy of the DOM Tree pattern. The policy confines all possible paths to a node n that might arise from running *any* code, if all members in the initial set that dominates n belong to class ReNode. For all restricted nodes in the new dominating set, they must have been descended from some restricted node ($\mathsf{rn}''.\mathsf{parent}^{\mathsf{j}}$), where ($\mathsf{rn}''.\mathsf{parent}^{\mathsf{j}}$) is accessible from a restricted node rn′′ in the old dominating set (rn′′.depth $\geq$ j). The policy says that an accessible ancestor rn′′ from a restricted node rn′ in the old dominating set is indexed by a depth level j so that the accesible ancestor would inherit a depth level j higher than rn′′′s depth level—(rn′′.parent$^{\mathsf{j}}$.depth = rn′′.depth - j). For a newly created restricted node rn′ in the new dominating set that is descended k levels down from one of these accessible ancestors, it would thus inherit a depth of (rn′′.depth - j + k). In other words every restricted node in the new dominating set must fall into one of these categories:

- a newly created restricted node descended k levels down directly from a restricted node in the old dominating set—**(j = 0, k > 0)**
- a newly created restricted node descended k levels down from some restricted node that is an accessible ancestor j levels up of a restricted node from the old dominating

set—**(j > 0, k > 0)**.

- a new created restricted node that is a direct ancestor from a restricted node in the old dominating set—**(j > 0, k = 0)**
- a restricted node in the new dominating set is also a member of the old dominating set—**(j = 0, k = 0)**

The argument for the policy is as follows: given a set of initial restricted nodes, we know by definition of the callable methods of a restricted node such that it can only create, store and return the capabilities of newly created restricted nodes, or return a null value, or return a String value, in **all** of its methods. The policies for these methods can be found in § 5.1.5.4. The execution policies in § 5.1.5.4 imply that all newly created restricted nodes from existing restricted nodes must obey certain policies (such as inheriting an incremented depth from its creator restricted node if its a child, or inheriting a decremented depth if it is a parent). Therefore, all restricted nodes in the new dominating set, if created during the unknown code execution, must be connected to a creator restricted node in the old dominating set through some path formed from the policies in § 5.1.5.4, or is already a member of the old dominating set, no matter what kind of code execution.

5.1.5.4 Policies for methods of ReNode

The six methods in ReNode are described in the policies as follows:

Policy —[Execution of ReNode.parent]

∀rn:ReNode, ∃i:String. ∀s. [(rn.depth > 0)
{`s = rn.parent()`}
(s:ReNode) ∧ (s = rn.parent) ∧ (s.child(i) = rn) ∧
(s.node = rn.node.parent) ∧
(s.depth = rn.depth - 1)
∨
s = `null`]

This policy says that the pre-condition creating a parent restricted node requires that the creator restricted node must have a depth level of more than 0. The post-conditions are that the newly created parent restricted node would be pointing towards a node that is the parent node of the node that its creator is pointing towards, and the newly created parent restricted node inherit the depth level of its creator restricted node decremented by 1.

Policy —[Execution of ReNode.setProp]

∀rn:ReNode, ∀i,j:String. [*true* {`rn.setProp(i,j)`} rn.node.prop(i) = j]

Policy —[Execution of ReNode.getProp]

∀rn:ReNode, ∀i,j:String. [*true* {`j = rn.getProp(i)`} j = rn.node.prop(i)]

Policy —[Execution of ReNode.addChild]
$\forall$rn,rn′:ReNode, $\forall$i:String. [*true*
`{rn′ = rn.addChild(i)}`
(rn′ = rn.child(i)) $\wedge$ (rn′.parent = rn) $\wedge$
(rn′.node = rn.node.child(i)) $\wedge$
(rn′.depth = rn.depth + 1)]

Policy —[Execution of ReNode.getChild]
$\forall$rn:ReNode, $\forall$i:String, $\forall$s. [*true*
`{s = rn.getChild(i)}`
(s:ReNode) $\wedge$ (s = rn.child(i)) $\wedge$ (s.parent = rn) $\wedge$
(s.node = rn.node.child(i)) $\wedge$
(s.depth = rn.depth + 1)]
$\vee$
s = `null`]

Policy —[Execution of ReNode.delChild]
$\forall$rn:ReNode, $\forall$i:String. [*true* `{rn.delChild(i)}` rn.child(i) = `null` $\wedge$
rn.node.child(i) = `null`]

5.1.5.5 Policy for Node Property Modification

This policy uses the Lemma [Field Modification requires Authority] and says that if a property of a node n is modified after the execution of some code by object o, then it implies o must be able to invoke a method on n.

Policy —[Node Property Modification]
$\forall$n:Node, $\forall$o:Object. [[n.prop(i) = j { o.*code* } n.prop(i) $\neq$ j] $\Longrightarrow$ MayCall(o,n)]

5.1.5.6 Policy for ReNode Constructor

The constructor for ReNode is private to the module and class ReNode. This means that an unknown object cannot directly use the constructor of ReNode even if it holds the capability of an object belonging to class ReNode, unless the unknown object is in the same module as ReNode.

Policy —[Constructor of new ReNode]
$\forall$rn:ReNode, $\forall$n:Node, $\forall$k:$\mathbb{N}$. [*true* `{rn = ReNode(n,k)}` rn.node = n $\wedge$
rn.depth = k $\wedge$
rn.parent = `null`]

5.1.5.7 Policies on invariants and immutable properties

The following policies describe the properties that hold in any state of execution. Namely, a node that a restricted node points to will never change [Immutability of ReNode.node], its ancestor(s) will never change [Immutability of ReNode.parent^k], its depth will never change [Immutability of ReNode depth], and the depth of a restricted node will never be less than 0 [Invariant of minimum ReNode depth].

Policy —[Immutability of ReNode.node]
$\forall$rn:ReNode, $\forall$n:Node. [rn.node = n {code} rn.node = n]

Policy —[Immutability of ReNode.parent^k]
$\forall$rn,rn$'$:ReNode, $\forall$k:$\mathbb{N}$. [rn.parent^k = rn$'$ {code} rn.parent^k = rn$'$]

Policy —[Immutability of ReNode depth]
$\forall$rn:ReNode, $\forall$k:$\mathbb{N}$. [rn.depth = k {code} rn.depth = k]

Policy —[Invariant of minimum ReNode depth]
$\forall$rn:ReNode. [*true* {code} rn.depth $\geq$ 0]

5.2 Reasoning about the DOM Tree with unknown code using our policies

In this subsection, we will now discuss how the policies we have developed in §5.1.5 allow us to reason about the use of restricted nodes of class ReNode to attenuate authority of nodes of class Node is a DOM tree. In particular, we can argue that authority is attenuated in the presence of the execution of unknown code. We also believe that the argument we have put forth using our logics is relatively simple, and natural to a programmer, compared with the corresponding works by Devriese et al.[DBP16] that uses Kripke world logics and Swasey et al.[SGD17] that builds on the Iris framework for concurrent separation logic. Furthermore, our framework allows us to reason about unknown code without needing to deploy more sophisticated tools of more advanced logic frameworks. In the code below, we create a DOM tree with the following structure:

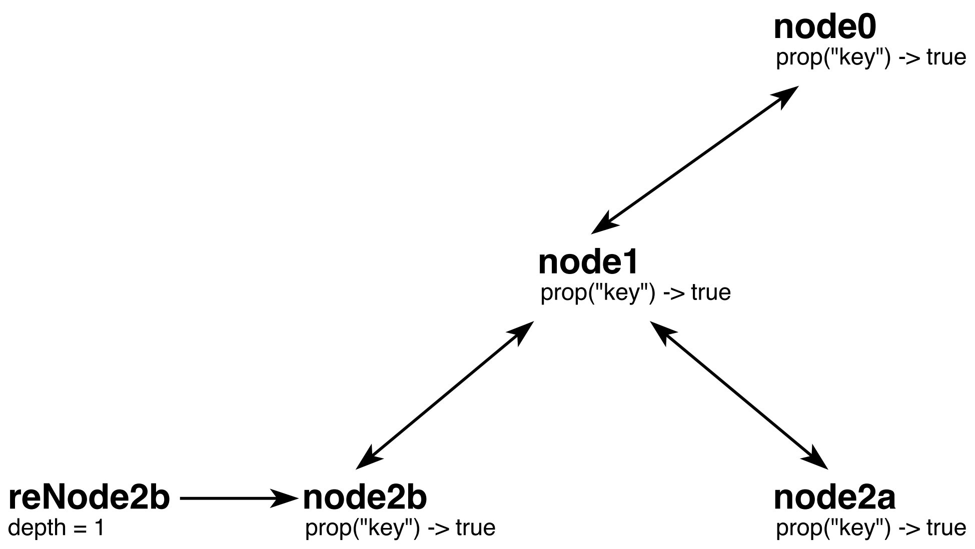


Figure 2: DOM Tree Example With Unknown Code

```
node0 = Node.createRoot("node0")
node0.setProp("key", "true")

node1 = node0.addChild("node1")
node1.setProp("key", "true")

node2a = node1.addChild("node2a")
node2a.setProp("key", "true")

node2b = node1.addChild("node2b")
node2b.setProp("key", "true")

reNode2b = ReNode(node2b, 1) //reNode2b given depth level of 1

// we have a mystery object o, which we do not trust
// o has a mystery method that takes an object reference as an argument
// we give o the capability of node2b, and o attempts to execute a mystery method on renode2b:

o.mystery0(renode2b)

Assertion 1: node0.getProp("key") = "true"

```

Figure 3: Code for DOM Tree Pattern with unknown provenance

It is unsafe to hand the capability of node2b directly to an untrusted agent, because it is possible the properties and state of the entire DOM tree would be compromised, in particular the root node node0 whose properties and state we want to protect in this example. Note that an untrusted agent having the direct capability of node2b implies being that the untrusted agent is able to obtain the direct capability of node1, node2a, and node0, because node2b has methods that freely gives those capabilities to any caller.

How then, can we allow an untrusted agent whose object and code we do not know, to safely modify the properties of node2b, and some other nodes (node2a and node1), whose properties are not critical and we do not mind being modified, but **not** allow the untrusted agent to modify the properties of node0 which are critical and which we want to protect?

We will accomplish this by creating an attenuating restricted node object called reNode2b, which points to node2b and has a depth level of 1 (reNode2b.node = node2b, reNode2b.depth = 1). On `line 19`, we pass reNode2b to an object o, of unknown provenance. While we do not know exactly how o might make use of reNode2b, to represent unknown code execution on reNode2b by o, we can assume that o would have a function called mystery that executes some unknown code on reNode2b. Using our policies, we can comfortably argue that the properties of node0 have been preserved after the execution of the unknown mystery function by o. However, because we have configured reNode2b to allow the unknown object to potentially modify the properties of node2a, node2b and node1, we do not know if the properties of node2a, node2b and node1 have been preserved.

We argue that **Assertion 1** at `line 21` holds, such that after execution of a piece of unknown code, node0.getProp("key") = "true". We begin by stating several facts at `line 18`, before the execution of mystery; we know that:

- **F1**: node2b.getProp("key") = true
- **F2**: reNode2b.depth = 1
- **F3**: reNode2b.node = node2b
- **F4**: reNode2b.parent.node = node1
- **F5**: reNode2b.parent.child("node2a").node = node2a
- **F6**: reNode2b.parent.parent.node = node0

Then at `line 21`, we consider the effects of the mystery function. We argue that:

- **(A)**: **F1** is preserved, unless **MayCall(o,node0)** holds at line 21. This is because to modify the value of "key" in node2b, **Policy [Node Property Modification]** at § 5.1.5.5 tells us that a method of node0 has to be invoked by o.

Therefore, to show that **Assertion 1** at `line 21` holds, it is sufficient to show that at `line 19`, the assertion MayCall(o,node0) does not hold. We will show this by contradiction.

Let us assume:

- **(B)**: MayCall(o,node0)

During execution of the method call mystery, the receiver is o, and the method argument is reNode2. Moreover, node0 was created before o. Therefore, within this frame, it holds that :

- **(C)**: Dom({reNode2b}, node0)
- **(D)**: reNode2b:ReNode

Applying **Policy [Object calls Node]** on **(C)** and **(D)**, we obtain:

- **(E)**: $\mathsf{MayCall(o,node0)} \implies \mathsf{MayCall(o,reNode2b)} \land \mathsf{MayCall(reNode2b,node0)}$

From **(D)** and **(E)** and applying **Policy [ReNode calls Node]**, we obtain that there exists a rn:ReNode such that:

- **(F)**: $\mathsf{MayCall(renode2b,node0)} \implies \mathsf{MayCall(renode2b,rn)} \land \mathsf{rn'.node} = \mathsf{node0}$

From **(D)** and by applying **Policy [ReNode calls ReNode]** on **(F)**, we know that there exists a k,j:$\mathbb{N}$ such that:

- **(G)**: $\mathsf{MayCall(renode2b,rn)} \implies \mathsf{reNode2b.parent}^{\mathsf{k}} = \mathsf{rn.parent}^{\mathsf{j}} \land \mathsf{reNode2b.depth} \geq \mathsf{k}$

From **(G)**, and **(F2)** which says $\mathsf{reNode2b.depth} = 1$, we obtain that:

- **(H)** k = 0 ∨ k = 1
- Using proof by cases,
  1st case $\mathsf{k} = 0$, then for all $\mathsf{rn'}$ such that $\mathsf{rn'.parent}^{\mathsf{j}} = \mathsf{reNode2b}$,
  we have that $\mathsf{rn'}$ are descendants of reNode2b,
  and therefore from **(F3) and (F6)**, we also know:
  $\mathbf{rn'.node \neq node0}$
  2st case $\mathsf{k} = 1$, then for all $\mathsf{rn'}$ such that $\mathsf{rn'.parent}^{\mathsf{j}} = \mathsf{reNode2b.parent}$,
  we have that $\mathsf{rn'}$ are descendants of reNode2b.parent,
  and therefore from **(F4) and (F6)**, we also know:
  $\mathbf{rn'.node \neq node0}$
- Therefore, rn′.node ≠ node0 from proof by cases.

We thus obtain a contradiction from **(F)**, and we know that assumption **(B)** must be false:

- $\mathbf{\neg MayCall(o,node0)}$

***Consequently, from **(A)**, we know the properties of node0 is preserved.

Note that in the above steps we used without mentioning explicitly that the parent and depth fields of all restricted nodes belonging to ReNode are immutable. Particularly, our **Policies on invariants and immutable properties** in § 5.1.5.7 tell us that the following must hold in all states of exection: **(F2-F6)**, and the minimum depth of all restricted nodes must be greater than 0.

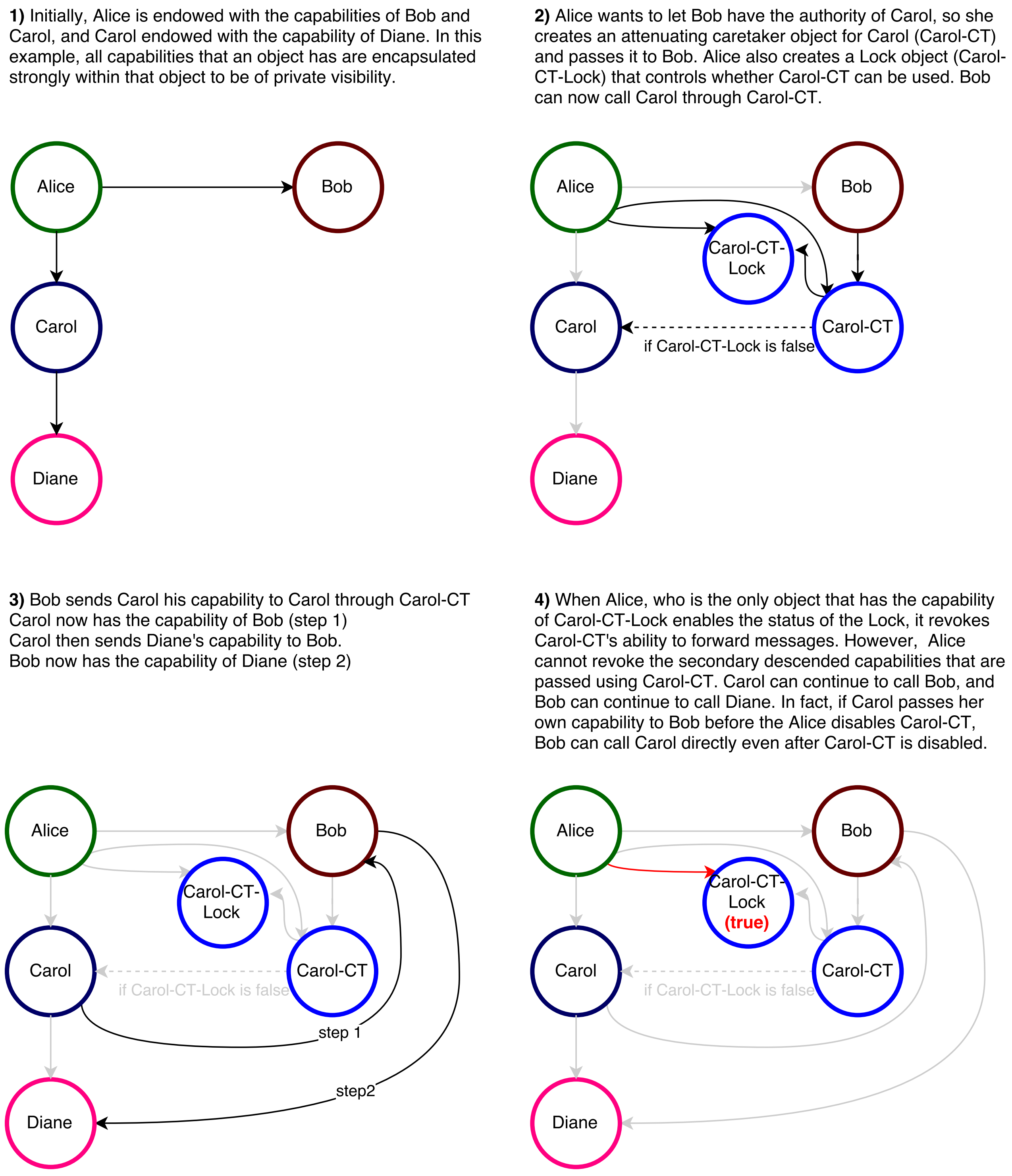
5.3 Caretaker and Membrane Patterns

The Caretaker pattern originally appeared in Redell's 1974 work [Red74], but features prominently along with its more advanced Membrane pattern in the works of [Mil06, Mur10, SGD17]. Unlike the previous DOM Tree pattern, protected objects in the Caretaker and Membrane pattern are now no longer necessarily ordered in some hierarchical access structure. Consequently, the attenuating 'caretaker' and 'membrane' objects do not try to maintain the integrity of some hierarchy structure, as compared to the attenuating ReNode object in the DOM Tree, where it maintains hierarchy of the Tree by adjusting the depth of ReNodes that it can return (parent ReNodes have decremented depths while child ReNodes have incremented depths).

The Caretaker merely fulfils the simple task of 1) masking the protected object's direct capability, and 2) only forwarding messages to the protected object according to some condition in an associated lock object. Note that a simple caretaker object forwards messages without analysing the contents of the messages, i.e. messages can contain capabilities. The Membrane pattern on the other hand, shares some similarities from both the DOM Tree pattern and Caretaker pattern. The Membrane pattern, is identical to the Caretaker pattern in the sense that it is meant to mask the protected object's direct capability and forward messages to the protected object, but with the exception that it does not forward messages that contain capabilities without first 'wrapping' those capabilities within new membranes. The membrane attenuating object can create more membranes (or deep attenuating objects) that is similar to the DOM Tree pattern where a restricted node creates more restricted nodes when accessing different parts of the tree.

We show the visual illustration of the differences between the Caretaker and Membrane patterns in fig. 4 and fig. 5. In the Caretaker pattern, while the status of the lock object Carol-CT-Lock for the caretaker object Carol-CT is disabled (Carol-CT.status = false), object Bob can pass his own capability to Carol using Carol-CT, and consequently, with Bob's capability, Carol can pass the capability of Diane, which Carol initially holds, to Bob. *After* the Lock object is enabled (Carol-CT.status = true), Carol-CT no longer forwards messages to Carol. However, because Bob has transferred his own capability to Carol, and Carol has transferred the capability of Diane to Bob during Carol-CT.status = false, even after Carol-CT.status = true, Carol can continue to communicate with Bob, and Bob can continue to communicate with Diane, because these objects can store the capabilities they have received for future use. For the Membrane example however, for all messages that contain capabilities, those messages will be attenuated before being forwarded. When Bob tries to send his own capability to Carol through the membrane Carol-M, Carol-M will wrap the capability of Bob within the message in a new membrane Bob-M and lock Bob-M-Lock, and forward the new membrane Bob-M to Carol, and when Carol tries to send the capability of Diane to Bob using Bob-M, a similar membrane Diane-M and lock object is created by Bob-M. When Alice decides to disable Carol-M by setting Carol-M-Lock = true, the disabling action can propagate to all membranes descended from Carol-M, thereby preventing communication from Carol to Bob and Bob to Diane.

5.3.1 Caretaker Pattern Illustration



1) Initially, Alice is endowed with the capabilities of Bob and Carol, and Carol endowed with the capability of Diane. In this example, all capabilities that an object has are encapsulated strongly within that object to be of private visibility.

2) Alice wants to let Bob have the authority of Carol, so she creates an attenuating caretaker object for Carol (Carol-CT) and passes it to Bob. Alice also creates a Lock object (Carol-CT-Lock) that controls whether Carol-CT can be used. Bob can now call Carol through Carol-CT.

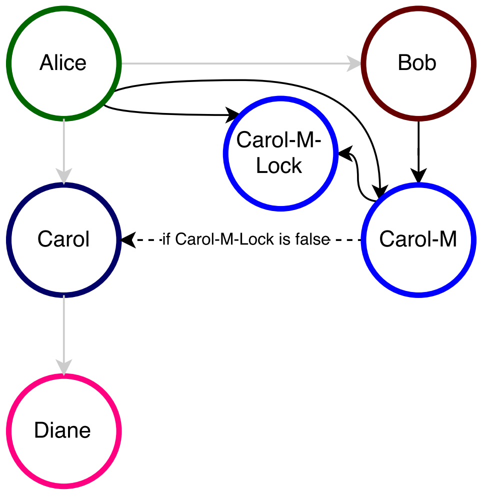
3) Bob sends Carol his capability to Carol through Carol-CT
Carol now has the capability of Bob (step 1)
Carol then sends Diane's capability to Bob.
Bob now has the capability of Diane (step 2)

4) When Alice, who is the only object that has the capability of Carol-CT-Lock enables the status of the Lock, it revokes Carol-CT's ability to forward messages. However, Alice cannot revoke the secondary descended capabilities that are passed using Carol-CT. Carol can continue to call Bob, and Bob can continue to call Diane. In fact, if Carol passes her own capability to Bob before the Alice disables Carol-CT, Bob can call Carol directly even after Carol-CT is disabled.

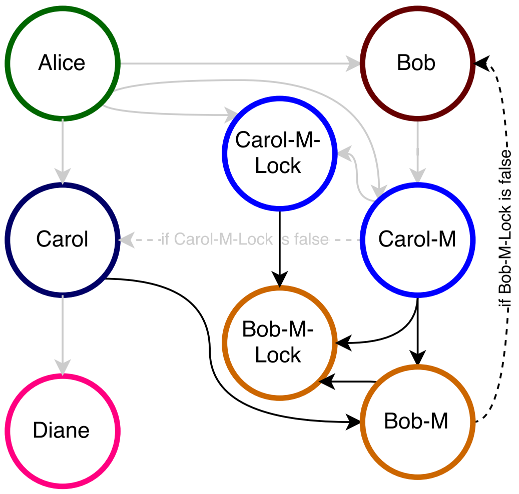
Figure 4: Caretaker Pattern

5.3.2 Membrane Pattern Illustration

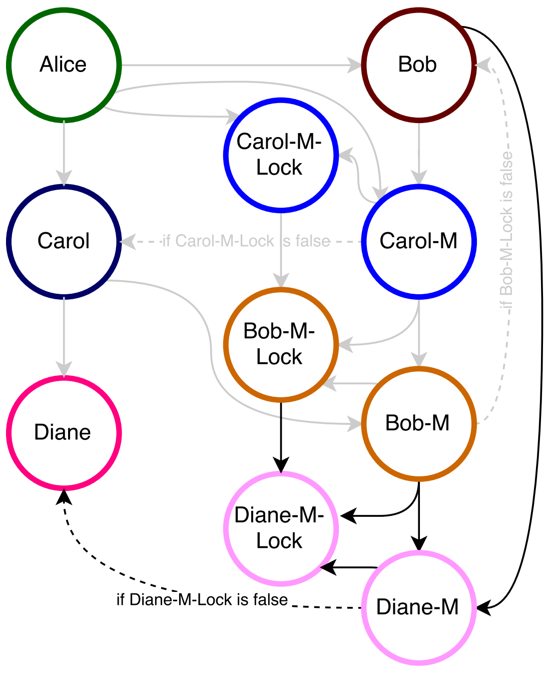
1) Alice wants to introduce Bob to Carol through attenuation, so she creates a deep attenuating membrane object for Carol (Carol-M), and its associated Lock object (Carol-M-Lock) that has a default boolean status=false. Alice passes only Carol-M to Bob. Bob can now call Carol through Carol-M.



2) Bob sends Carol his capability to Carol through Carol-M. Carol-M detects a capability being forwarded and modifies the message. Carol-M creates a Lock object (Bob-M-Lock) and constructs a membrane for Bob (Bob-M). Carol-M then introduces Bob-M-Lock to Carol-M-Lock. Lastly, Carol-M forwards to Carol the capability of Bob-M instead of Bob. Carol can now call Bob through Bob-M.



2) Carol sends Diane's capability to Bob through Bob-M. Bob-M detects a capability being forwarded, and creates a Lock object (Diane-M-Lock) and constructs a membrane for Diane (Diane-M). Bob-M then introduces Diane-M-Lock to Bob-M-Lock. Lastly, Bob-M forwards to Bob the capability of Diane-M instead of Diane. Bob can now call Diane through Diane-M.



4) Alice decides to do a deep revocation of Carol-M by calling lockall() on Carol-M-Lock. This results in a chain of calls (Carol-M-Lock calls the same method on Bob-M-Lock, which Bob-M-Lock then calls the same method on Diane-M-Lock). Alice has effectively revoked Bob's authority to Carol and all descended authorities from Carol-M. Carol-M and its chain of descended membranes is represented by the dotted black circle.

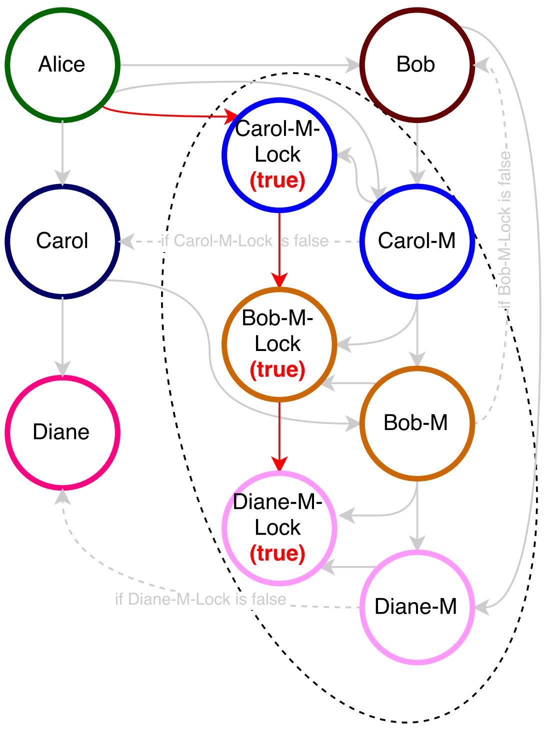


Figure 5: Membrane Pattern

6 Case Study of Ethereum / Solidity as a Non-OCap Model

6.1 Motivation

Recent widespread adoption of distributed ledger technology (blockchain) has created multiple decentralised, distributed computational platforms where millions of dollars are transacted over codified constructs called smart contracts. For example, as of 7 September 2017, Ethereum[Woo14] is approximately a US$30 billion blockchain platform with an in-built Turing-complete programming language that can be used to create and deploy such contracts.

6.2 Solidity - Language for smart contracts

Solidity[6] is a high-level language with a syntax similar to that of JavaScript and is designed to write smart contracts on the Ethereum Virtual Machine (EVM), the Turing-complete 256bit runtime environment of the Ethereum blockchain[7].

6.3 Objects in Solidity

On the Ethereum blockchain, there are generally two types of accounts: "externally-owned accounts"(external accounts) and "contract accounts"(contracts). We define these entities as objects because they encapsulate both state and behaviour. In terms of state, both types of accounts have an Ether balance (the currency of Ethereum) associated with them, while contract accounts differ from external accounts by having additional persistent contract storage that can store data such as integers and strings. In terms of behaviour, both types of accounts come with a predefined set of methods to transfer Ether, but contract accounts can have additional specified methods. The methods of external accounts can only be called by a person who has authenticated himself as the owner of the external account using a private key, while methods of contract accounts that are deployed on the blockchain can be called based on any conditions defined in the contract. Hence, external accounts are controlled by public-private key cryptography, while contract accounts are controlled by their contract code. An external account has no code but can send messages to other external accounts or contract accounts by creating or signing a transaction. For contract accounts, everytime the contract account receives a message, its code activates, allowing it to read and write to its internal storage and sending other messages to other external accounts or contracts. Transactions can be triggered from both types of accounts, though contracts can only trigger transactions in response to other transactions that they have received. Therefore, all transactions in Ethereum can only originate from external accounts.

6.3.1 Object References/Addresses

In Ethereum, the designations of all external account and contract objects are made of up a 20 bytes value. Hence, if we treat smart contracts and Ethereum accounts as objects, then these

[6]https://solidity.readthedocs.io/en/develop/

[7]Details on the Ethereum blockchain platform in our paper are based on the Etheruem white-paper: https://github.com/ethereum/wiki/wiki/White-Paper

object references are non-opaque—because anybody can specify an address using a 20 bytes string value. That is, anybody on Ethereum only needs the bit information of a particular address of an object to start interacting with the object. Unlike in Java where knowing the bit information of an object address does not imply knowing the designation/reference of an object, in Ethereum, the bit information of an address *is* the object reference.

6.4 Object Protection in Solidity

Because all smart contract/account object references in Solidity are available to everybody, the concept of isolating objects do not exist in Ethereum. Consequently, the concept of protecting objects through eventual path-isolation does not exist, because:

$$\forall \mathsf{o}{:}\mathsf{ExtAcct},\ \forall \mathsf{o}'{:}\mathsf{Object}.\ [\ \mathsf{MayAccess}^{Dir,Now}(\mathsf{o},\mathsf{o}')\ ]$$

where we define o as objects representing external accounts and o′ as objects representing external accounts or contracts in Solidity. In other words, on the Solidity platform, every external account has a path to every object on the blockchain. Therefore, in Ethereum, private-public key cryptography is used to determine whether one has the authority to use an external account, while for contract accounts, we can only use a form of stack-based access control that allow whether a method called is allowed to succeed or fail—the protected contract object o′ can only enforce policies in a state σ *after* it has been called by another object since:

$$\forall \mathsf{o}{:}\mathsf{ExtAcct},\ \forall \mathsf{o}'{:}\mathsf{Object}.\ [\ \mathsf{MayCall}(\mathsf{o},\mathsf{o}')\ ]$$

6.4.1 Protection through Message Sender Authentication

In Ethereum, protection of an external account is done through private key authentication and protection of a smart contract is done through specifying the conditions on whether a particular method call would be successful. For external accounts, once a person has provided the correct private key, the person is allowed to call any of the predefined methods in an external account that facilitate ether transactions. For contracts, things become more complex because the code of all contracts are transparent and publicly available to everyone. What then, are the kind of protection mechanisms that a smart contract can deploy? As addresses of smart contracts are transparent and globally available, contracts *cannot* prevent anybody from sending them messages, but they can dictate *how* they respond to these messages.

All this means is that from the receiver object's viewpoint, there must be a way for the receiver to know *who* or what is the sender address of the message. The Ethereum platform facilitates sender authentication, such then when an external account or contract account calls a method on a contract account o′ by sending a message to o′, o′ as the receiver, can always see the address of the message sender, and can trust the address of the message sender to be authentic[8].

[8]Note that we are less concerned with how the architecture of the Ethereum platform guarantees the authenticity of who the message sender is, but more with higher-level protection strategies that smart contracts can employ

In other words, when Object A sends a message to Object B, Object A *cannot* mask or hide its own address or pretend to be another Object in the message to B, and B can always guarantee that the designation of the sender is the true sender of the message. If the sender of the message is an address of an external account, we know that a person has authenticated using a private-key to be the external account's owner and fired off the transaction from the external account. If a sender of a message is a contract address, then we know the message is triggered from the contract (but that the origin of such a transaction has to be an external account).

From the receiver viewpoint, Solidity allows a method call to succeed or fail based on function modifiers. Function modifiers represent conditions written by the programmer that can be attached to methods that determine whether calls on that method would succeed or fail. They provide more flexibility than the private-public access modifiers in most object-oriented languages. They differ from access modifiers in the sense that access modifiers restrict an object's method from being called by a caller, i.e. the method *cannot be called*, while function modifiers in Solidity, they can be considered as *part of the method execution*. In a smart contract, one can use the following code in Solidity to determine who exactly is the message sender:

- `Msg.sender` gives the designation of the message sender
- `Tx.origin` allows stack introspection and gives the designation of the first sender(originator)

A common strategy used in smart contracts to protect methods meant only for the owner of the contract is to first store the 20-byte address of the deployer of the smart contract in an arbitrary variable `owner` during the constructor of the contract:

```
//Contract is called TestContract
TestContract {
        address owner;
        function TestContract{ //constructor
                owner = msg.sender;
                }
}
```

With the owner of the contract stored as a persistent state variable, we can now write a function modifier called `ownerOnly()` in TestContract:

```
modifier ownerOnly(){ require(msg.sender == owner);_;}
```

Attaching this owner-only modifier on a method `restrictedMeth()`, we have:

```
//Contract is called TestContract
TestContract {
        address owner;
        modifier ownerOnly(){ require(msg.sender == owner);_;}
        function TestContract{ //constructor
                owner = msg.sender;
```

knowing such a guarantee exists.

```
            }
        function  restrictedMeth ()  ownerOnly { //does something  restricted  to  owner}
}
```

where we now know that while any object in Solidity can call `restrictedMeth()` in TestContract, the execution of the method will fail if the object caller's address is not the same as the owner of the contract.

6.5 Further Work

We note that our MayCall predicate is inadequate to describe the protection mechanisms of smart contracts in Ethereum. This is because, on the Ethereum platform, the protection mechanism in a smart contract essentially happens *during* the execution of a method in that object, i.e., that the receiver in the state of execution must have *already* transitioned into that object, *before* that object can protect itself. Through some condition specified in the method call (e.g. checking the identity of the message sender or other conditions), the method call will fail during execution, where the Ethereum virtual machine will, up till the 'point of failure', revert all state changes during the method execution, except the payment of the underlying 'gas' fees to execute the contract on the blockchain. Therefore, the platform on Ethereum requires stronger formal definitions of authority that does not merely mean a transition of the receiver from an object to another object—the stronger definition of authority needs to be able to describe the *successful completion* of a method call.

7 Conclusion

To formally specify OCap patterns in the style of OCap policies, our paper contributes to the literature by proposing new formal definitions for permission and authority that describe access dynamics between objects. Using our formal definition, we have reasoned in a novel way about isolation, cooperation, vulnerability, and protection of objects in the face of trusted and untrusted code. Our paper has also provided an illustration of three OCap patterns and more specifically, the OCap policy specifications using our proposed methodology for the DOM Tree pattern. Using our OCap policies in the DOM Tree pattern, we show how we can preserve properties of a system that interacts with code of unknown provenance. In the course of coming up with our OCap policies, we have also proposed the novel use of the concept of domination over objects. We have further demonstrated that it is challenging to reason about security properties of modern open systems like the Ethereum blockchain platform that does not adhere to an OCap system, and further work needs to be done in this area given the meteoric rise of capital traded and number of smart contracts on blockchain technology.

8 Bibliography

References

[CDBM15] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming*

Based on Actors, Agents, and Decentralized Control, pages 1–12. ACM, 2015.

[CPN98] 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.

[DBP16] 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.

[DN13] Sophia Drossopoulou and James Noble. The need for capability policies. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, page 6. ACM, 2013.

[DN14] Sophia Drossopoulou and James Noble. Towards capability policy specification and verification. Technical report, Technical Report ECSTR-14-05, School of Engineering and Computer Science, Victoria University of Wellington, 2014.

[DNM15] Sophia Drossopoulou, James Noble, and Mark S Miller. Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, pages 2–15. ACM, 2015.

[DNMM15] Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Reasoning about risk and trust in an open word. 2015.

[DNMM16] Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Permission and authority revisited towards a formalisation. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, page 10. ACM, 2016.

[DVH66] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[Har85] Norman Hardy. Keykos architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[Lam74] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[Mil06] Mark S Miller. Robust composition: Towards a unified approach to access control and concurrency control. 2006.

[MMF00] Mark S Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *International Conference on Financial Cryptography*, pages 349–378. Springer, 2000.

[MMT10] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125–140. IEEE, 2010.

[MS03] 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.

[Mur10] Toby Murray. *Analysing the security properties of object-capability patterns*. Oxford University, 2010.

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374, 2010.

[MYS$^+$03] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[Red74] David D Redell. Naming and protection in extendible operating systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1974.

[SGD17] David Swasey, Deepak Gark, and Dreyer Derek. Robust and compositional verification of object capability patterns. 2017.

[SSF99] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[WCC$^+$74] 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.

[WN79] Maurice Vincent Wilkes and Roger Michael Needham. The cambridge cap computer and its operating system. 1979.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

9 Appendix

9.1 Module Linking

Here we show the definition of module linking found in the appendix of [DNMM15].

Definition 6 (Linking)**.** *Linking of modules M and M' is*

$$* \ : \ Module \times Module \ \longrightarrow \ Module$$

$$M * M' = \begin{cases} M *_{aux} M', & \textit{if } WFL(M, M') \\ \bot & \textit{otherwise.} \end{cases}$$

$$(M *_{aux} M')(c) = \begin{cases} M(id), & \textit{if } M(id) \textit{ is defined} \\ M'(id) & \textit{otherwise.} \end{cases}$$

$$\begin{array}{l} WFL(M, M') \ \equiv \\ \quad dom(M) \cap dom(M') \cap ClassId = \emptyset \ \land \\ \quad \forall S \in SpecId \cap dom(M) \cap dom(M').\ M(S) = M(S') \\ \quad WFP(M, M') \ \land \ WFP(M', M) \\ WFP(M, M') \ \equiv \\ \forall C.\ M(C) = \textbf{private}.... \ \rightarrow \ \textbf{new}\ C... \textit{ does not appear in } M' \end{array}$$

In the above, the predicate $WFL(M, M')$ asserts that linking of the modules M and M' is well-defined. It requires that 1) classes are not defined more than once, 2) specifications may have been defined more than once, but then their bodies must be identical[5], and 3) no module can call private constructors[6] from another module. This means, that the runtime system enforces this form of privacy. For example, if in module M_{mp} we define `Purse` as **private** and `Mint` as not private, the call of the `Purse` constructor is restricted to only insider the module M_{mp}, while the call of `Mint` is unrestricted. This means, that only objects of classes defined in the module M_{mp} may create `Purse`s, while clients of M_{mp} may create objects of class `Mint`. In effect, the creation of `Mint` is publicly available, but the creation of `Purses` is restricted

9.2 Assertion

Here we show the definition of assertion found in the appendix of [DNMM15].

Definition 12 (Validity of one-state assertions)**.** *Given an oracle $\mathcal{O} \subseteq Module \times ClassId$, the validity of an assertion A, is defined through the* partial *judgments:*

$$\models \subseteq Module \times state \times Oracle \times Assertion$$
$$\not\models \subseteq Module \times state \times Oracle \times Assertion$$

using the notations $M, \sigma \models_{\mathcal{O}} A$ and $M, \sigma \not\models_{\mathcal{O}} A$:

- $M, \sigma \models_{\mathcal{O}} \mathtt{e}$, *if* $\lfloor \mathtt{e} \rfloor_{M,\sigma} = \mathbf{true}$,
  $M, \sigma \not\models_{\mathcal{O}} \mathtt{e}$, *if* $\lfloor \mathtt{e} \rfloor_{M,\sigma} = \mathbf{false}$,
  undefined, otherwise.
- $M, \sigma \models_{\mathcal{O}} \mathtt{e}_1 \geq \mathtt{e}_2$, *if* $\lfloor \mathtt{e}_1 \rfloor_{M,\sigma} \geq \lfloor \mathtt{e}_2 \rfloor_{M,\sigma}$,
  $M, \sigma \not\models_{\mathcal{O}} \mathtt{e}$, $\lfloor \mathtt{e}_1 \rfloor_{M,\sigma} < \lfloor \mathtt{e}_2 \rfloor_{M,\sigma}$,
  undefined, if $\lfloor \mathtt{e}_1 \rfloor_{M,\sigma}$ *or* $\lfloor \mathtt{e}_1 \rfloor_{M,\sigma}$ *is undefined, or not a number.*
- $M, \sigma \models_{\mathcal{O}} Q(\mathtt{e}_0, \mathtt{e}_1, ...\mathtt{e}_n)$, *if*
  $M, \sigma \models_{\mathcal{O}} A[\mathtt{e}_0/this, \mathtt{e}_1/p_1, ...\mathtt{e}_n/p_n]$
  $M, \sigma \not\models_{\mathcal{O}} Q(\mathtt{e}_0, \mathtt{e}_1, ...\mathtt{e}_n)$, *if*
  $M, \sigma \not\models_{\mathcal{O}} A[\mathtt{e}_0/this, \mathtt{e}_1/p_1, ...\mathtt{e}_n/p_n]$
  if $\mathcal{P}(M, \lfloor \mathtt{e}_0 \rfloor_\sigma \downarrow_1, Q) = \mathbf{predicate}\ Q\ \mathbf{(}\ p_1...p_n\mathbf{)}\ \mathbf{\{}\ A\ \mathbf{\}}$,
  undefined, if $\mathcal{P}(M, \lfloor \mathtt{e}_0 \rfloor_\sigma \downarrow_1, Q)$ *undefined, or if* $M, \sigma \models_{\mathcal{O}} A(\mathtt{e}_0, \mathtt{e}_1, ...\mathtt{e}_n)$ *undefined.*
- $M, \sigma \models_{\mathcal{O}} A_1 \wedge A_2$, *if* $M, \sigma \models_{\mathcal{O}} A_1$ *and* $M, \sigma \models_{\mathcal{O}} A_2$,
  $M, \sigma \not\models_{\mathcal{O}} A_1 \wedge A_2$, *if* $M, \sigma \not\models_{\mathcal{O}} A_1$ *or* $M, \sigma \not\models_{\mathcal{O}} A_2$
  undefined, if $M, \sigma \models_{\mathcal{O}} A_1$ *or* $M, \sigma \models_{\mathcal{O}} A_2$ *is undefined.*
- $M, \sigma \models_{\mathcal{O}} A_1 \rightarrow A_2$, *if* $M, \sigma \models_{\mathcal{O}} A_1$ *and* $M, \sigma \models_{\mathcal{O}} A_2$,
  or $M, \sigma \not\models_{\mathcal{O}} A_1$.
  $M, \sigma \not\models_{\mathcal{O}} A_1 \rightarrow A_2$, *if* $M, \sigma \models_{\mathcal{O}} A_1$ *and* $M, \sigma \not\models_{\mathcal{O}} A_2$,
  undefined, if $M, \sigma \models_{\mathcal{O}} A_1$ *or* $M, \sigma \models_{\mathcal{O}} A_2$ *is undefined.*
- $M, \sigma \models_{\mathcal{O}} \exists x.A$ *iff for some address* ι *and some fresh variable* $z \in VarId$*, we have* $M, \sigma[z \mapsto \iota] \models_{\mathcal{O}} A[z/x]$.
  $M, \sigma \not\models_{\mathcal{O}} \exists x.A$ *iff for all address* ι *and fresh variable* $z \in VarId$*, we have* $M, \sigma[z \mapsto \iota] \not\models_{\mathcal{O}} A[z/x]$.
  undefined, otherwise.
- $M, \sigma \models_{\mathcal{O}} \forall x.A$ *iff for all addresses* $\iota \in dom(\sigma)$*, and fresh variable* z*, we have* $M, \sigma[z \mapsto \iota] \models_{\mathcal{O}} A[z/x]$.
  $M, \sigma \not\models_{\mathcal{O}} \forall x.A$ *iff there exists an address* $\iota \in dom(\sigma)$*, and fresh variable* z*, such that* $M, \sigma[z \mapsto \iota] \not\models_{\mathcal{O}} A[z/x]$.
  undefined, otherwise.
- $M, \sigma \models_{\mathcal{O}} \mathtt{e}{:}\mathtt{C}$, *if* $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 = \mathtt{C}$.
  $M, \sigma \not\models_{\mathcal{O}} \mathtt{e}{:}\mathtt{C}$, *if* $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 \neq \mathtt{C}$.
  undefined, if $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \notin dom(\sigma \downarrow_2)$.
- $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect\mathbf{(}\ \mathtt{e}, \mathtt{e}'\mathbf{)}$, *if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *and* $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ *are defined, and there exists a method* $\mathtt{m}$*, arguments* $\bar{\mathtt{a}}$*, state* σ'*, identifier* z*, such that* $M, \sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}], z\,\mathbf{.}\,\mathtt{m}(\bar{\mathtt{a}}) \rightsquigarrow \sigma'$*, and* $\lfloor \mathtt{e}' \rfloor_{M,\sigma} \neq \lfloor \mathtt{e}' \rfloor_{M,\sigma\downarrow_1,\sigma'\downarrow_1}$.
  $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect\mathbf{(}\ \mathtt{e}, \mathtt{e}'\mathbf{)}$, *undefined if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *or* $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ *are undefined.*
  $M, \sigma \not\models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect\mathbf{(}\ \mathtt{e}, \mathtt{e}'\mathbf{)}$, *otherwise.*
- $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, *if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *and* $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ *are defined, and there exist fields* $\mathtt{f}_1$*,...* $\mathtt{f}_n$*, such that* $\lfloor z.\mathtt{f}_1...\mathtt{f}_n \rfloor_{M,\sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}]} = \lfloor \mathtt{e}' \rfloor_{M,\sigma}$.
  $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, *undefined if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *or* $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ *are undefined.*
  $M, \sigma \not\models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, *otherwise.*
- $M, \sigma \models_{\mathcal{O}} \mathtt{e}$ ***obeys*** $\mathtt{S}$, *undefined, if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *undefined, unknown, if* $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ *unknown, or* $\mathcal{C}lass(\mathtt{e}, \sigma) \notin dom(M)$.
  $M, \sigma \models_{\mathcal{O}} \mathtt{e}$ ***obeys*** $\mathtt{S}$, *if* $\mathcal{O}(M, C, S) = true$
  $M, \sigma \not\models_{\mathcal{O}} \mathtt{e}$ ***obeys*** $\mathtt{S}$, *if* $\mathcal{O}(M, C, S) = false$
  where $\mathtt{C}$=$\mathcal{C}lass(\mathtt{e}, \sigma)$.

In the above, the notation $\sigma[v \mapsto \iota]$ *is shorthand for* $(\phi[v \mapsto \iota], \chi)$ *for a state* $\sigma = (\phi, \chi)$.