Risk and Trust in Open Systems

Towards formalizing Permission and Authority, and Policies for Object-Capability Patterns

Author: Shu-Peng Loh
Supervisor: Sophia Drossopoulou

MSC COMPUTING SCIENCE THESIS PRESENTATION

Motivation & Goals

security of complex software systems which use third-party unknown code (open system) is difficult

Object-Capability (OCAP) system compelling approach to building robust, distributed systems --*cooperation without vulnerability*.

Much work done on OCAP patterns, less work done on formalizing specifications

Example of OCAP pattern – protecting nodes in a DOM Tree

- Works by Maffeis et al.[MMT10] and Devriese et al.[DBP16]

Important to formally reason how OCAP patterns help to preserve or protect properties of an open system in the face of unknown code

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code

CON1: Permission Definition MayAccess

Describes having capability –

- either directly or indirectly
- either now or eventual ('possibly later')

Definition—[MayAccess]

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

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

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

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

CON1: Permission – Observations (1)

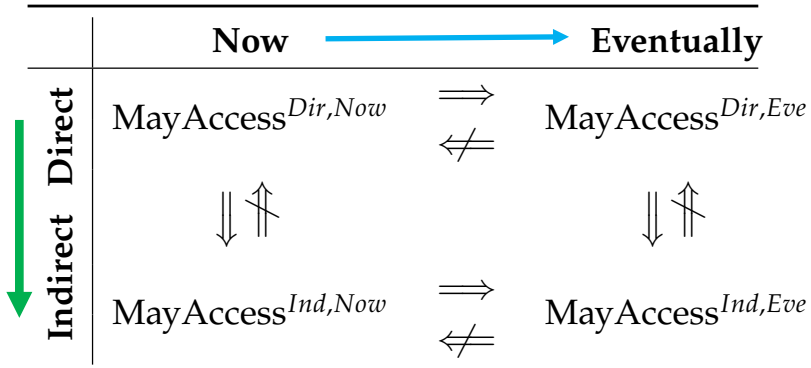
Having permission ’directly’ **implies** having permission ‘indirectly’

- A direct capability (stored in single field) is part of the definition of indirect capability (stored in a field reachable from a series of field(s))

Having permission ‘now’ **implies** having capability ‘possibly later’

- A state ’now’ is part of the definition of possibly arising states ‘later’

Table 1: MayAccess Relations - Observations



CON1: Authority – Definition MayCall

Describes *ability to exercise* capability –

New definition (MayCall) describes a change in receiver

Definition—[MayCall]

$M,\sigma \vDash \text{MayCall}(x,e) \iff$

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

Drossopoulou et al.'s MayAffect – describes execution that results in a change of a field

Definition—[MayAffect from Drossopoulou et al.[DNMM16]]

$M,\sigma \vDash \text{MayAffect}(x,e)$ if there exists a method m, argument(s) $\bar{a}$, state σ', such that:

$M,\sigma\ e.m(\bar{a}) \rightsquigarrow \sigma'$, and $\lfloor e \rfloor_{M,\sigma} \neq \lfloor e' \rfloor_{M,\sigma'}$

CON1: Authority – Why MayCall?

Weaker than MayAffect, can describe a chain of method calls over multiple objects through a chain of authorities MayCall(s)

In our language, all fields are **private**, hence MayCall is necessary for field modification (or MayAffect implies MayCall)

Lemma—[Field Modification Requires Authority]

$\forall \sigma', f.\ [\sigma' \in \text{Arising}(M,\sigma) \wedge \lfloor o'.f \rfloor_{\sigma'} \neq \lfloor o'.f \rfloor_{\sigma} \Longrightarrow \exists o.\ M,\sigma \vDash \text{MayCall}(o,o')\ ]$

Note that o can possibly refer to o′

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code

CON2: Definition – Domination

Domination – set S dominates an object x iff it is necessary for any object y with a chain (indirect) permission of x to have the permission or authority of a member *o* of the dominating set *S*

Definition—[Domination]

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

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

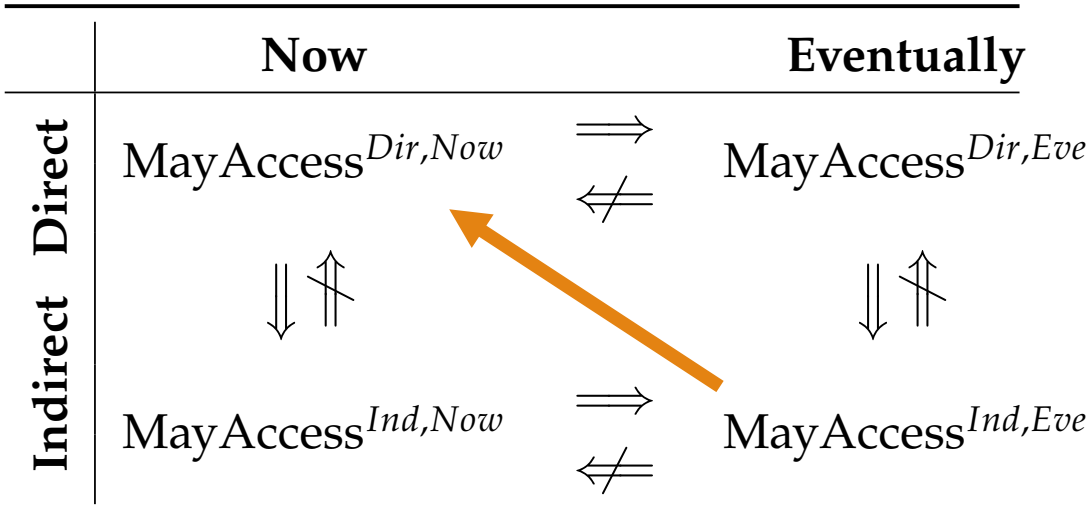
- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code

CON3: OCAP Permissions–Lemma

Can we think about what present reference graph configurations are necessary for specific future reference graph configurations?

Table 1: MayAccess Relations - Observations



CON3: OCAP Permissions–Lemma (cont.)

Can we think about what present reference graph configurations are necessary for specific future reference graph configurations?

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

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

$\Longrightarrow$

$\exists x.\ [(\mathsf{MayAccess}^{Dir,Now}(o,x)$ *o has capability of x now*

$\vee$

$\mathsf{MayAccess}^{Dir,Now}(x,o))$ *x introduces capability of itself to o*

$\wedge$

$\mathsf{MayAccess}^{Ind,Eve}(x,o')]$ *x has eventual path to o' (recursively expandable)*

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code

CON4: OCAP Authority Requires Permission

Because objects in OCAP systems can only communicate on capabilities

Method invocation requires capabilities

Observation—[Permission(Indirect,Eventual) is necessary for Authority]

$M,\sigma \vDash \mathsf{MayCall}(o,o') \implies \mathsf{MayAccess}^{Ind,Eve}(o,o')$

CON4: Isolation and cooperation

o has no possible paths to o’ in any arising state

Definition—[Isolation of o′ from o]

$M,\sigma \vDash \text{Isolated}(o,o') \iff \neg\text{MayAccess}^{Ind,Eve}(o,o')$

o has at least one possible path to o’ in an arising state

Definition—[Cooperation (o $\rightarrow$ o′)]:

$M,\sigma \vDash \text{Cooperation}(o,o') \iff M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(o,o')$

CON4: Types of cooperation

o is a *trusted* object (either we know the code of o, or we don't know the code of o but we trust the source of o to be safe)

Definition—[Safe cooperation]

$M,\sigma \vDash \text{SafeCooperation}(o,o') \iff o{:}\text{TrustedObj} \wedge \text{Cooperation}(o,o')$

o* is an un*trusted* object, then such cooperation is unsafe

Definition—[Vulnerable cooperation]

$M,\sigma \vDash \text{VulnerableCooperation}(o^*,o') \iff o^*{:}\text{UntrustedObj} \wedge \text{Cooperation}(o^*,o')$

CON4: Types of cooperation (cont.)

o* is an un*trusted* object,
but we know a set of trusted objects dominates object o' which we want to ***protect:***

Definition—[Protected cooperation]

$$M,\sigma \vDash \text{ProtectedCooperation}(o^*,o') \iff \text{VulnerableCooperation}(o^*,o') \wedge \text{Dom}(S,o') \wedge \forall s.\ [\ s{\in}S \implies s{:}\text{TrustedObj}\ ]$$

Protected cooperation implies a safe cooperation
vulnerability of o' has been shifted to s'

Lemma—[Protected cooperation implies a Safe cooperation]

$$M,\sigma \vDash \text{ProtectedCooperation}(o^*,o') \implies \exists s'{:}\text{TrustedObj}.\ [\ \text{VulnerableCooperation}(o^*,s') \wedge \text{SafeCooperation}(s',o')\ ]$$

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code

CON5: Our formal definitions help to write easy OCAP *Policies*

Our paper proposes the OCAP policies for three well-known OCAP patterns:

DOM Tree, Caretaker, Membrane

We focus on showing the policies for the **DOM Tree pattern**, and further show how those policies can be used to **reason about cooperating with unknown code while still preserving properties of the system**

CON5: Caretaker Pattern

Policies

Here we show the important policies for the Caretaker pattern, where we also say how the Caretaker might leak paths in **Policy [Possible Leaked Paths from Caretaker]**. Path leakage is possible because the caretaker might possibly return the capability of its protected object o′ from some method call forwarded to o′, when o′ has a method that exposes its own capability; i.e. the caretaker does not create deep attenuating objects to mask capabilities returned by its protected object, or messages containing capabilities passed by its caller.

Policy —[Object calls Node]

$$\forall o,o'\text{:Object}, \forall n\text{:Node}.\ [\ \text{MayCall}(o,o') \wedge \text{Dom}(S,o') \wedge \forall s.\ [\ s \in S \implies s\text{:Caretaker}\ ] \implies \exists ct\text{:Caretaker}.\ [\ ct \in S \wedge \text{MayCall}(o,ct) \wedge \text{MayCall}(ct,o')\ ]\ ]$$

Policy —[Possible Leaked Paths from Caretaker]

$$\neg\ [\ \forall o\text{:Object}, S_{old}.\ [\ \text{Dom}(S_{old},o) \wedge \forall s.\ [\ s \in S_{old} \implies s\text{:Caretaker}\ ]\ \{\ code\ \}\ \exists S_{new}.\ [\ \text{Dom}(S_{new},n) \wedge \forall s' \in S_{new}.\ [\ s'\text{:Caretaker}\ ]\ ]\ ]\ ]$$

Policy —[Caretaker calls Caretaker.target (Necessary Condition)]

$$\forall c\text{:Caretaker}.\ [\ \text{MayCall}(c,c.\text{target}) \implies c.\text{lock.status} = \texttt{false}\ ]$$

Policy —[Execution of Lock.lock()]

$$\forall k\text{:Lock}.\ [\ \textit{true}\ \{\texttt{k.lock()}\}\ k.\text{status} = \texttt{true}\ ]$$

Policy —[Constructor of new Caretaker]

$$\forall ct\text{:Caretaker}, \forall o\text{:Object}, \forall k\text{:Lock}.\ [\ \textit{true}\ \{\texttt{ct = Caretaker(o,k)}\}\ ct.\text{target} = o \wedge ct.\text{lock} = k\ ]$$

Policy —[Immutability of Caretaker.lock]

$$\forall c\text{:Caretaker}.\ [\ c.\text{lock} = k\ \{code\}\ c.\text{lock} = k\ ]$$

Policy —[Immutability of Caretaker.target]

$$\forall c\text{:Caretaker}.\ [\ c.\text{target} = o\ \{code\}\ c.\text{target} = o\ ]$$

CON5: Membrane Pattern

Policies

Here we show the important policies for the Membrane pattern. Unlike the Caretaker pattern, a Membrane does not leak capabilities because the membrane creates deep attenuating new membranes to mask capabilities returned by its protected object, and do so also for capabilities in messages passed by its caller. A membrane that creates new membranes also creates the associated lock objects for the new membranes, and passes the capabilities of these new locks to its own associated lock object k. Therefore when some other object calls the method lockall() on the lock object k, k will recursively call the same method on all lock objects that it has. Therefore, every membrane and lock object has a ghost field parent:

- The ghost field parent of a membrane object m′ points to a membrane m if the constructor for m′ is called by m, otherwise it points to null
- The ghost field parent of a lock object k′ points to a lock k if the constructor for k′ is called by a membrane m, where m.lock = k, otherwise it points to null

Policy —[Object calls Object′]

$$\forall o,o'{:}\mathsf{Object}.\ [\ \mathsf{MayCall}(o,o') \land \mathsf{Dom}(S,o') \land \forall s.\ [\ s{\in}S \implies s{:}\mathsf{Membrane}\ ]$$
$$\implies$$
$$\exists m{:}\mathsf{Membrane}.\ [\ m{\in}S \land \mathsf{MayCall}(o,m) \land \mathsf{MayCall}(m,o')\ ]\ ]$$

Policy —[Domination No Leaked Paths from Membrane]

$$\forall o{:}\mathsf{Object},\ \mathsf{MEM}_{old}.\ [\ \mathsf{Dom}(\mathsf{MEM}_{old},o) \land \forall s.\ [\ s{\in}\mathsf{MEM}_{old} \implies s{:}\mathsf{Membrane}\ ]$$
$$\{\ code\ \}$$
$$\exists \mathsf{MEM}_{new}.\ [\ \mathsf{Dom}(\mathsf{MEM}_{new},n) \land \forall s'{\in}\mathsf{MEM}_{new}.\ [\ s'{:}\mathsf{Membrane} \land$$
$$\exists s''{\in}\mathsf{MEM}_{old},\ \exists k{:}\mathbb{N}.\ [\ (s'.\mathsf{parent}^k = s'')\ ]\ ]\ ]\ ]$$

Policy —[Membrane calls Membrane.target (Necessary Condition)]

$$\forall m{:}\mathsf{Membrane}.\ [\ \mathsf{MayCall}(m,m.\mathsf{target}) \implies m.\mathsf{lock.status} = \texttt{false}\ ]$$

Policy —[Constructor of new Membrane]

$$\forall m{:}\mathsf{Membrane}, \forall o{:}\mathsf{Object}, \forall k{:}\mathsf{Lock}.\ [\ \mathit{true}\ \{\texttt{m = Membrane(o,k)}\}\ m.\mathsf{target} = o \land m.\mathsf{lock} = k\ ]$$

Policy —[Each Membrane in the Tree has an associated Lock in the same Tree]

$$\forall m,m'{:}\mathsf{Membrane}, \forall j{:}\mathbb{N}.\ [\ m.\mathsf{parent}^j = m' \iff m.\mathsf{lock.parent}^j = m'.\mathsf{lock}]$$

Policy —[Execution of Lock.lock()]

$$\forall k{:}\mathsf{Lock}.\ [\ \mathit{true}\ \{\texttt{k.lock()}\}\ k.\mathsf{status} = \texttt{true}\ ]$$

Policy —[Execution of Lock.lockall()]

$$\forall k{:}\mathsf{Lock}.\ [\ \mathit{true}\ \{\texttt{k.lockall()}\}\ \forall k'{:}\mathsf{Lock}, \forall j{>}0.\ [\ k'.\mathsf{parent}^j = k \implies k'.\mathsf{status} = \texttt{true}\ ]$$

CON5: DOM Tree Pattern

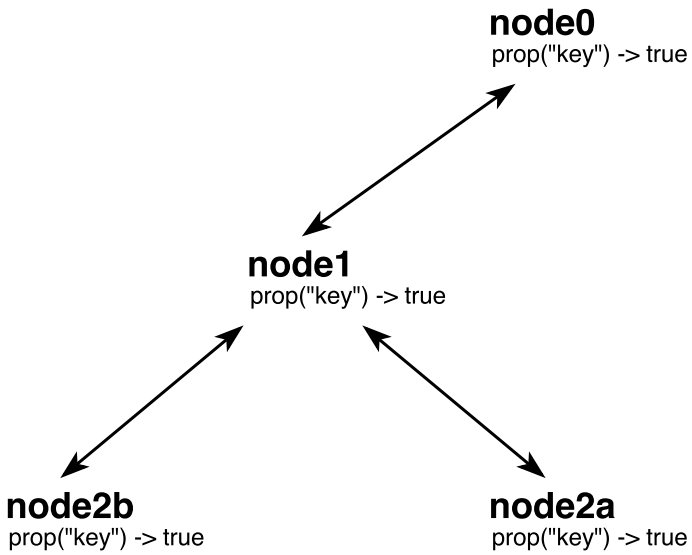
Nodes are ordered in a hierarchical tree

Nodes have properties and methods that can access other nodes in the tree:

.setProp(i,j) sets the property i (key) to be j (value)

.parent() gives the reference(capability) of the node above

.getChild(*id*) gives the capability of a child node with the identifier *id*



Any untrusted object which obtains the direct capability of a node *n* can obtain the permission and authority of all other nodes reachable from *n*

Consequence: properties of all nodes in the same tree are modifiable

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

So how do we allow an untrusted object to only modify the properties of some nodes of the tree, but protect the properties of other nodes?

CON5: DOM Tree Nodes

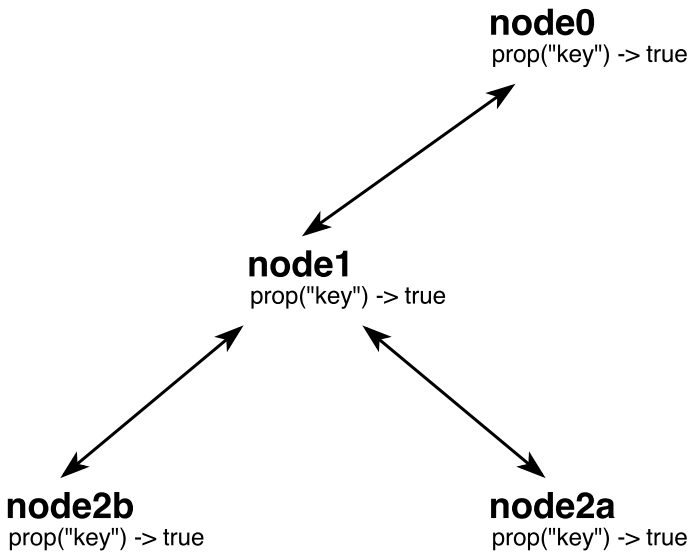
Nodes are ordered in a hierarchical tree

Nodes have properties and methods that can access other nodes in the tree:

.setProp(i,j) sets the property i (key) to be j (value)

.parent() gives the reference(capability) of the node above

.getChild(*id*) gives the capability of a child node with the identifier *id*



Any untrusted object which obtains the direct capability of a node *n* can obtain the permission and authority of all other nodes reachable from *n*

Consequence: properties of all nodes in the same tree are modifiable

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

So how do we allow an untrusted object to only modify the properties of some nodes of the tree, but protect the properties of other nodes?

CON5: DOM Tree Attenuating Object: Restricted Node (ReNode)

We use a restricted node as an attenuating object

ReNode Specification - Classical

Policy —[Execution of ReNode.parent]

$\forall$rn:ReNode, $\exists$i:String. $\forall$s. [(rn.depth $>$ 0)
`{s = rn.parent()}`
(s:ReNode) $\wedge$ (s = rn.parent) $\wedge$ (s.child(i) = rn) $\wedge$
(s.node = rn.node.parent) $\wedge$
(s.depth = rn.depth - 1)
$\vee$
s = `null`]

Policy —[Execution of ReNode.setProp]

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

Policy —[Execution of ReNode.getProp]

$\forall$rn:ReNode, $\forall$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]

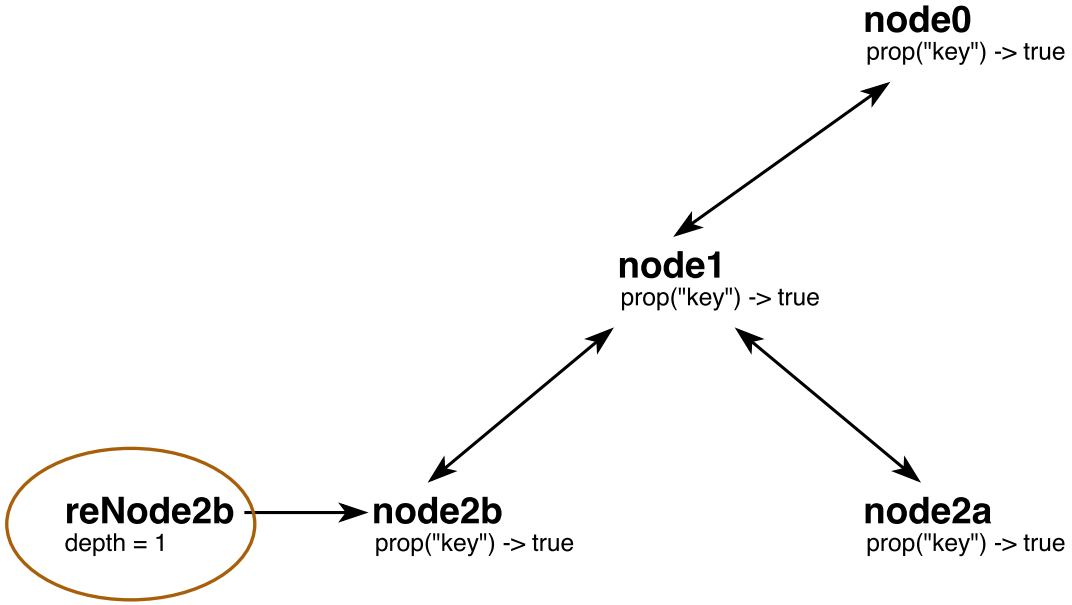
$\forall$rn:ReNode, $\forall$i:String, $\forall$s. [*true*
`{s = rn.getChild(i)}`
(s:ReNode) $\wedge$ (s = rn.child(i)) $\wedge$ (s.parent = rn) $\wedge$
(s.node = rn.node.child(i)) $\wedge$
(s.depth = rn.depth + 1)]
$\vee$
s = `null`]

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`]

CON5: DOM Tree Attenuating Object: Restricted Node (ReNode)

Policies on authority of ReNodes



A restricted node *rn* can only call a node n' through a further restricted node *rn'* which points to *n* through its node field (*rn'.node = n)*

The depth of a ReNode restrricts its authority

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 ReNode (Necessary Condition)]

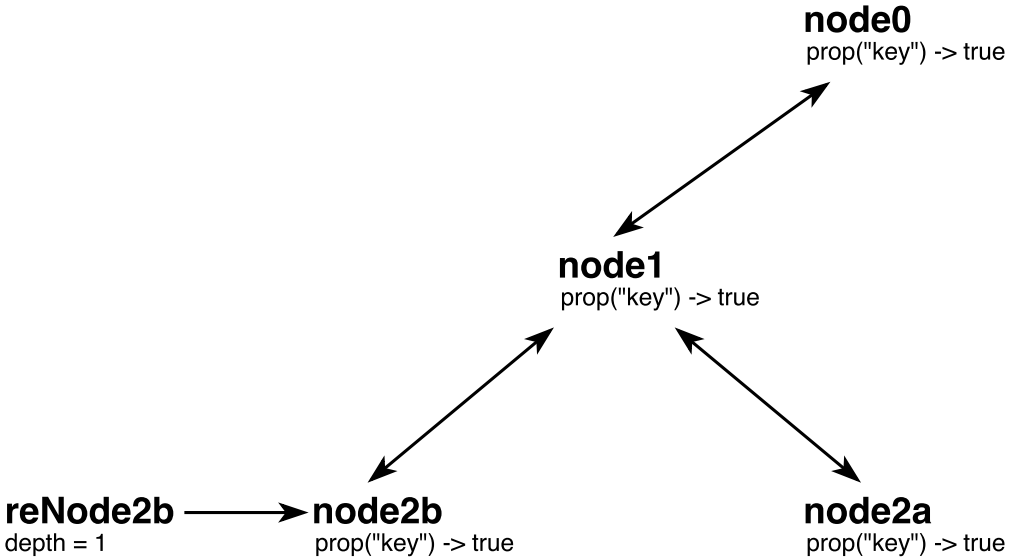
$\forall$rn,rn′:ReNode. [MayCall(rn,rn′)

$\implies$

$\exists$k,j:$\mathbb{N}$. [(rn.parent^k = rn′.parent^j) $\wedge$ rn.depth $\geq$ k]]

CON5: DOM Tree Attenuating Object: Restricted Node (ReNode)

Policy on domination



No rights amplification: If node n is dominated by a set of restricted nodes, then execution of any code will result in a new dominating set which will consist of restricted nodes only, and whose members will not have more authority over n than the original one.

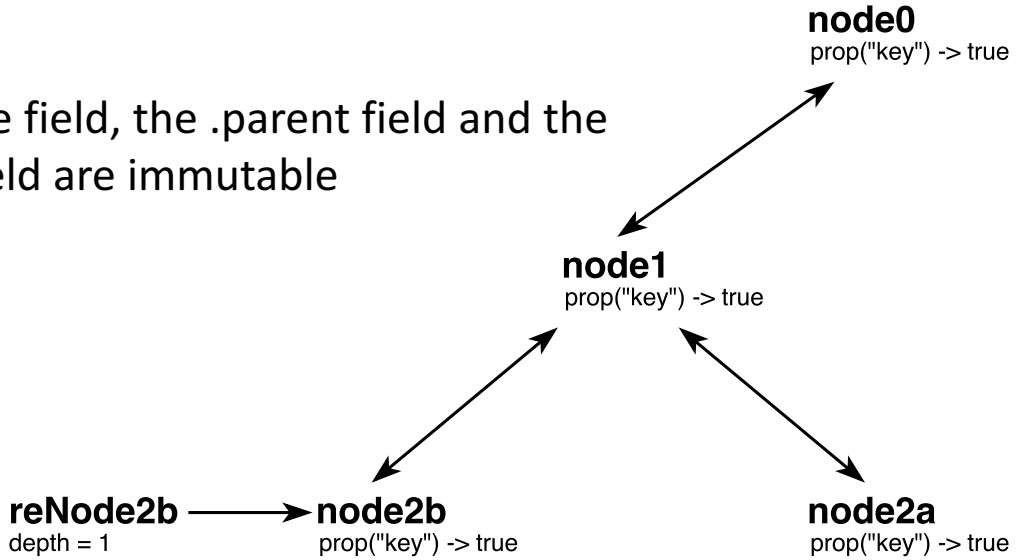
Policy —[Domination No Leaked Paths from ReNode]

$$\forall n{:}Node,\ RND_{old}.\ [\ Dom(RND_{old},n) \wedge \forall s.\ [\ s \in RND_{old} \Longrightarrow s{:}ReNode\ ]$$
$$\{\ code\ \}$$
$$\exists RND_{new}.\ [\ Dom(RND_{new},n) \wedge \forall s' \in RND_{new}.\ [\ s'{:}ReNode \wedge$$
$$\exists s'' \in RND_{old},\ \exists k,j{:}\mathbb{N}.\ [\ (s'.parent^{k} = s''.parent^{j}) \wedge (s''.depth \geq j) \wedge$$
$$(s'.depth = s''.depth - j + k)\ ]\ ]\ ]\ ]$$

CON5: DOM Tree Attenuating Object: Restricted Node (ReNode)

Policy on immutability

The .node field, the .parent field and the .depth field are immutable



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]

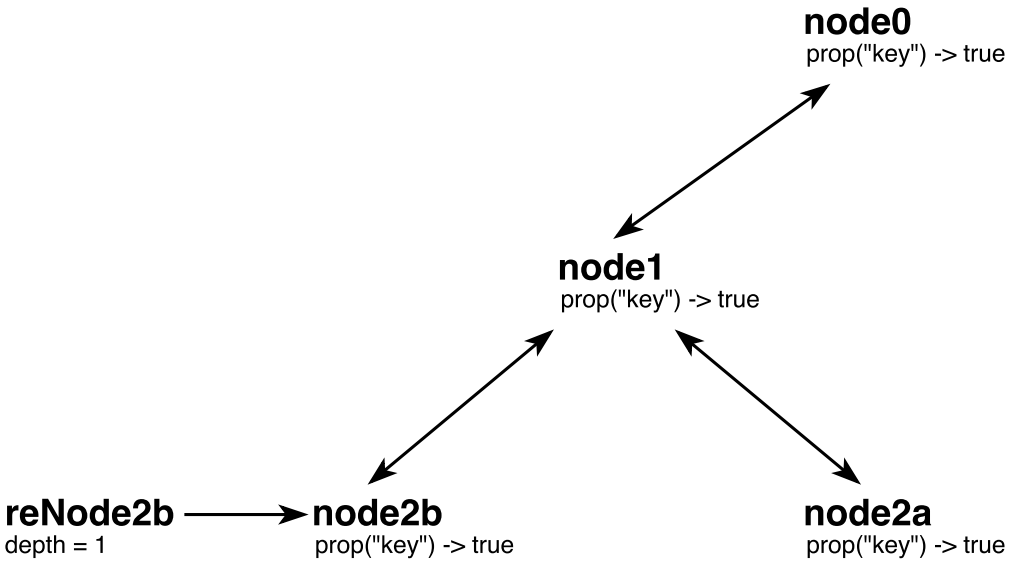
$\forall$rn:ReNode, $\forall$k:$\mathbb{N}$. [rn.depth = k {code} rn.depth = k]

Policy —[Immutability of minimum ReNode depth]

$\forall$rn:ReNode. [*true* {code} rn.depth $\geq$ 0]

CON5: DOM Tree Attenuating Object: Restricted Node (ReNode)

System-wide Policy



Lastly, if we have a dominating restricted node set *S* over node *n*, then authority over *n* must also have authority of a restricted node *rn* from *S*

Policy —[Object calls Node]

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

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

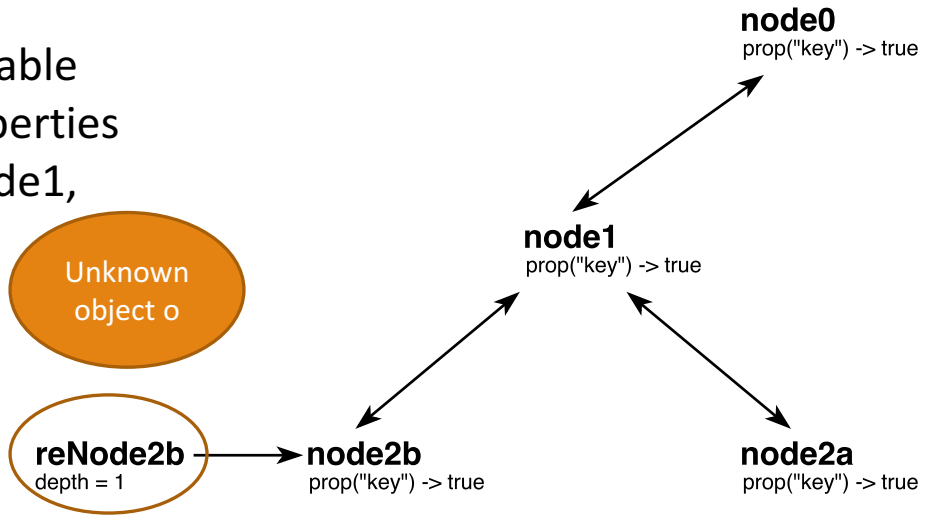
→ **CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code**

Reasoning about unknown code (1)

Suppose we want to give a untrusted 3rd party client the authority to modify node2b, node1 and node2a, but NOT node0.

How can we reason that the properties of node0 are protected?

System specs:
o needs to be able to modify properties of node2b, node1, and node2a.



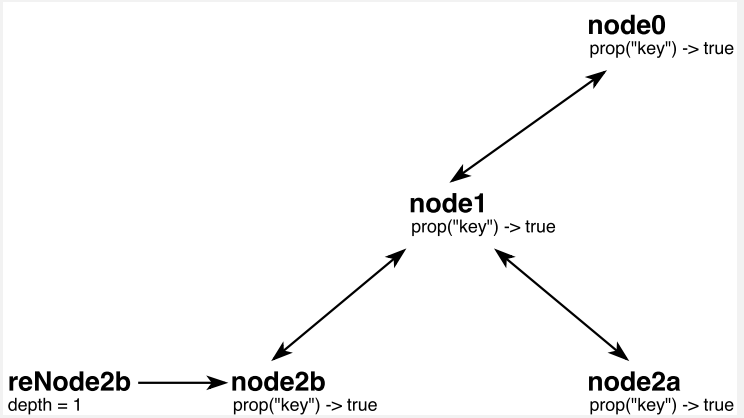

```
1  node0 = Node.createRoot("node0")
2  node0.setProp("key", "true")
3
4  node1 = node0.addChild("node1")
5  node1.setProp("key", "true")
6
7  node2a = node1.addChild("node2a")
8  node2a.setProp("key", "true")
9
10 node2b = node1.addChild("node2b")
11 node2b.setProp("key", "true")
12
13 reNode2b = ReNode(node2b, 1) //reNode2b given depth level of 1
14
15 // we have a mystery object o, which we do not trust
16 // o has a mystery method that takes an object reference as an argument
17 // we give o the capability of node2b, and o attempts to execute a mystery method on renode2b:
18
19 o.mystery0(renode2b)
20
21 Assertion 1: node0.getProp("key") = "true"
22
```

Reasoning about unknown code (2) - Some facts prior to mystery call

Suppose we want to give a untrusted 3rd party client the authority to modify node2b, node1 and node2a, but NOT node0.

How can we reason that the properties of node0 are protected?

```
1  node0 = Node.createRoot("node0")
2  node0.setProp("key", "true")
3
4  node1 = node0.addChild("node1")
5  node1.setProp("key", "true")
6
7  node2a = node1.addChild("node2a")
8  node2a.setProp("key", "true")
9
10 node2b = node1.addChild("node2b")
11 node2b.setProp("key", "true")
12
13 reNode2b = ReNode(node2b, 1) //reNode2b given depth level of 1
14
15 // we have a mystery object o, which we do not trust
16 // o has a mystery method that takes an object reference as an argument
17 // we give o the capability of node2b, and o attempts to execute a mystery method on renode2b:
18
19 o.mystery0(renode2b)
20
21 Assertion 1: node0.getProp("key") = "true"
22
```



We argue that **Assertion 1** at `line 21` holds, i.e. after execution of a piece of unknown code, we know that node0.getProp("key") = "true". We now sketch the proof of this assertion below.

We begin by stating several facts at `line 18`, before the execution of mystery. namely:

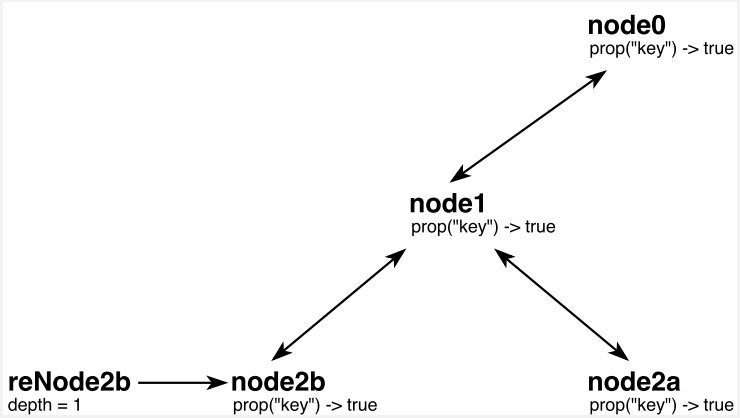
- **F1**: node2b.getProp("key") = true
- **F2**: reNode2b.depth = 1
- **F3**: reNode2b.node = node2b
- **F4**: reNode2b.parent.node = node1
- **F5**: reNode2b.parent.child("node2a").node = node2a
- **F6**: reNode2b.parent.parent.node = node0

Reasoning about unknown code (3) - modification of node0 requires authority

Suppose we want to give a untrusted 3rd party client the authority to modify node2b, node1 and node2a, but NOT node0.

How can we reason that the properties of node0 are protected?

```
1  node0 = Node.createRoot("node0")
2  node0.setProp("key", "true")
3
4  node1 = node0.addChild("node1")
5  node1.setProp("key", "true")
6
7  node2a = node1.addChild("node2a")
8  node2a.setProp("key", "true")
9
10 node2b = node1.addChild("node2b")
11 node2b.setProp("key", "true")
12
13 reNode2b = ReNode(node2b, 1) //reNode2b given depth level of 1
14
15 // we have a mystery object o, which we do not trust
16 // o has a mystery method that takes an object reference as an argument
17 // we give o the capability of node2b, and o attempts to execute a mystery method on renode2b:
18
19 o.mystery0(renode2b)
20
21 Assertion 1: node0.getProp("key") = "true"
22
```



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 19. This is based on the observation below:

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

  ---

  which follows from our Lemma [Field Modification Requires Authority], which 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.

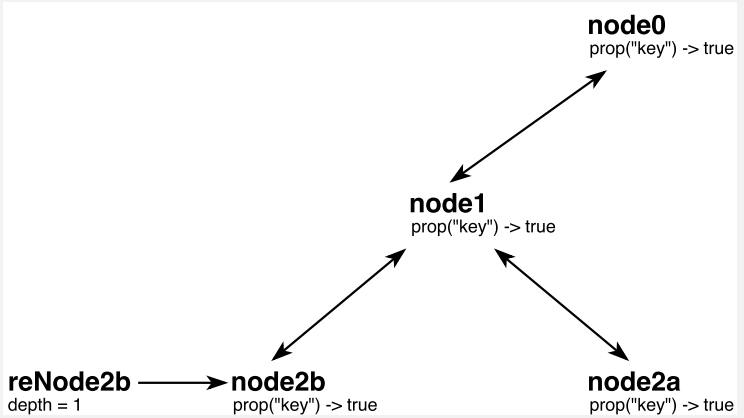
Therefore, to show that **Assertion 1** at `line 21` holds, it is sufficient to show that at `line 19`, the assertion MayCall(o,node0) does not hold. We will show this by contradiction.

Reasoning about unknown code (4) -Proof by contradiction

Suppose we want to give a untrusted 3rd party client the authority to modify node2b, node1 and node2a, but NOT node0.

How can we reason that the properties of node0 are protected?

```
1  node0 = Node.createRoot("node0")
2  node0.setProp("key", "true")
3
4  node1 = node0.addChild("node1")
5  node1.setProp("key", "true")
6
7  node2a = node1.addChild("node2a")
8  node2a.setProp("key", "true")
9
10 node2b = node1.addChild("node2b")
11 node2b.setProp("key", "true")
12
13 reNode2b = ReNode(node2b, 1) //reNode2b given depth level of 1
14
15 // we have a mystery object o, which we do not trust
16 // o has a mystery method that takes an object reference as an argument
17 // we give o the capability of node2b, and o attempts to execute a mystery method on renode2b:
18
19 o.mystery0(renode2b)
20
21 Assertion 1: node0.getProp("key") = "true"
22
```



Let us assume during execution of the mystery function in `line 19`; at some point:

- **(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

Reasoning about unknown code (5) -Finishing the proof

Suppose we want to give a untrusted 3rd party client the authority to modify node2b, node1 and node2a, but NOT node0.

How can we reason that the properties of node0 are protected?

Results from previous slides:

(A): **F1** is preserved, unless **MayCall(o,node0)** holds at line 19.

- **F1**: node2b.getProp("key") = true
- **F2**: reNode2b.depth = 1
- **(C)**: Dom({reNode2b}, node0)
- **(D)**: reNode2b:ReNode

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

- **(E)**: MayCall(o,node0) $\implies$ MayCall(o,reNode2b) $\wedge$ MayCall(reNode2b,node0)

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

- **(F)**: MayCall(renode2b,node0) $\implies$ MayCall(renode2b,rn) $\wedge$ rn′.node = 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)**: MayCall(renode2b,rn) $\implies$ reNode2b.parent^k = rn.parent^j $\wedge$ reNode2b.depth $\geq$ k

From **(G)**, and **(F2)** which says reNode2b.depth = 1, we obtain that:

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

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

- **¬MayCall(o,node0)**

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

Contributions / Outcomes

CON1: Proposed **new** formal definitions of Permission and Authority, inspired but not identical from the works of Drossopoulou et al.[DNMM16]

CON2: Proposed **novel use** of Domination, inspired by the work by Clarke et al. on ownership types[CPN98]

CON3: Proposed a **eventual paths lemma** in OCAP that describes necessary present reference graph configurations for specific future reference graph configurations

CON4: Made **observations using our methodology** on concepts such as Isolation and Cooperation;

- Safe Cooperation, Vulnerable Cooperation, and Protected Cooperation

CON5: Presented 3 OCAP patterns: DOM Tree, Caretaker, Membrane Pattern

- **(new) wrote them in the capability-safe language Pony**
- **(new) proposed OCAP policies for all three patterns**

→ **CON6: Shown how properties of nodes in DOM Tree can be preserved in the face of unknown code**

Appendix

CON1: Permission – Observations (2)

Having ‘indirect’ permission **does not imply** having ’direct’ permission

Example: **MayAccess^{Ind,_}(A,C) but not MayAccess^{Dir,_}(A,C)**

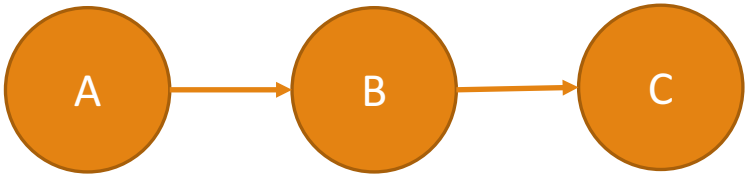


Table 1: MayAccess Relations - Observations

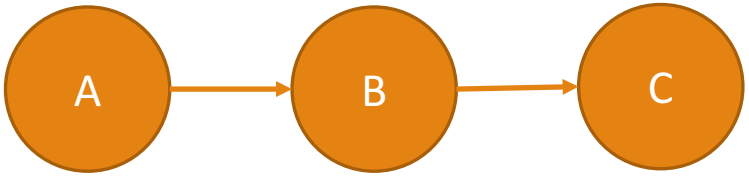
		Now		Eventually
Direct		$MayAccess^{Dir,Now}$	$\Rightarrow \not\Leftarrow$	$MayAccess^{Dir,Eve}$
		$\Downarrow \not\Uparrow$		$\Downarrow \not\Uparrow$
Indirect		$MayAccess^{Ind,Now}$	$\Rightarrow \not\Leftarrow$	$MayAccess^{Ind,Eve}$

CON1: Permission – Observations (3)

‘Possibly later’ permission **does not imply** having permission ’now’

Example: **$\text{MayAccess}^{\text{Dir,Eve}}(A,C)$ but not $\neg\text{MayAccess}^{\text{Dir,Now}}(A,C)$**

‘Now’ **$\neg\text{MayAccess}^{\text{Dir,Now}}(A,C)$**



If B passes capability of C to A, then…

‘Later’ **$\text{MayAccess}^{\text{Dir,Eve}}(A,C)$**

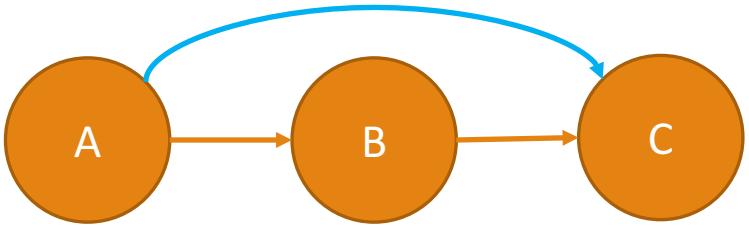


Table 1: MayAccess Relations - Observations

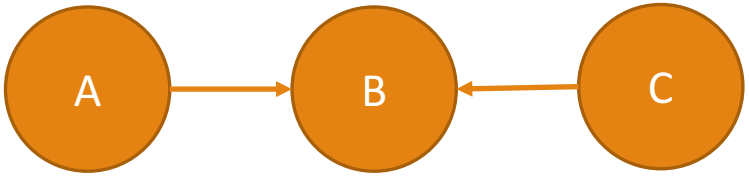
	Now		Eventually
Direct	$\text{MayAccess}^{Dir,Now}$	$\Rightarrow$ $\not\Leftarrow$	$\text{MayAccess}^{Dir,Eve}$
	$\Downarrow \not\Uparrow$		$\Downarrow \not\Uparrow$
Indirect	$\text{MayAccess}^{Ind,Now}$	$\Rightarrow$ $\not\Leftarrow$	$\text{MayAccess}^{Ind,Eve}$

CON1: Permission – Observations (4)

‘Possibly later’ permission **does not imply** having permission ’now’

Example: **MayAccess^{Ind,Eve}(A,C) but not MayAccess^{Ind,Now}(A,C)**

‘Now’ **¬MayAccess^{Ind,Now}(A,C)**



If C passes capability of itself to B then…

‘Later’ **MayAccess^{Ind,Eve}(A,C)**

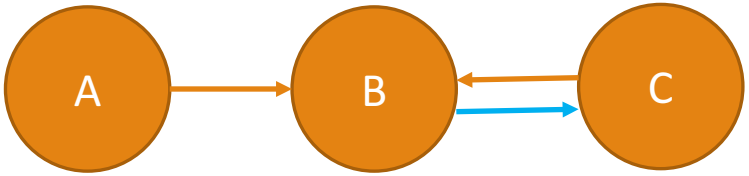
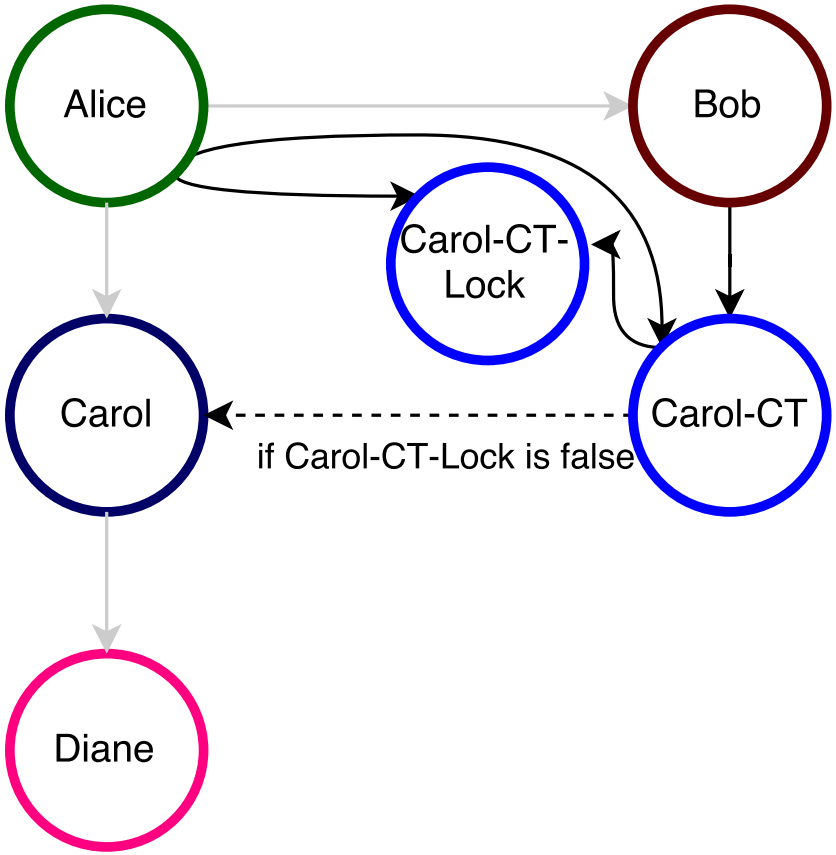


Table 1: MayAccess Relations - Observations

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

CON5: Caretaker Pattern

Illustration



Membrane Pattern

Illustration