TLA+ Specification & Model Checking of the Agoric Smart Contracts Kernel

TLA+ Conference

30.09.2021

Andrey Kuprianov & Daniel Tisdall

informal SYSTEMS

Talk outline

How we approach Correctness Assurance with TLA+

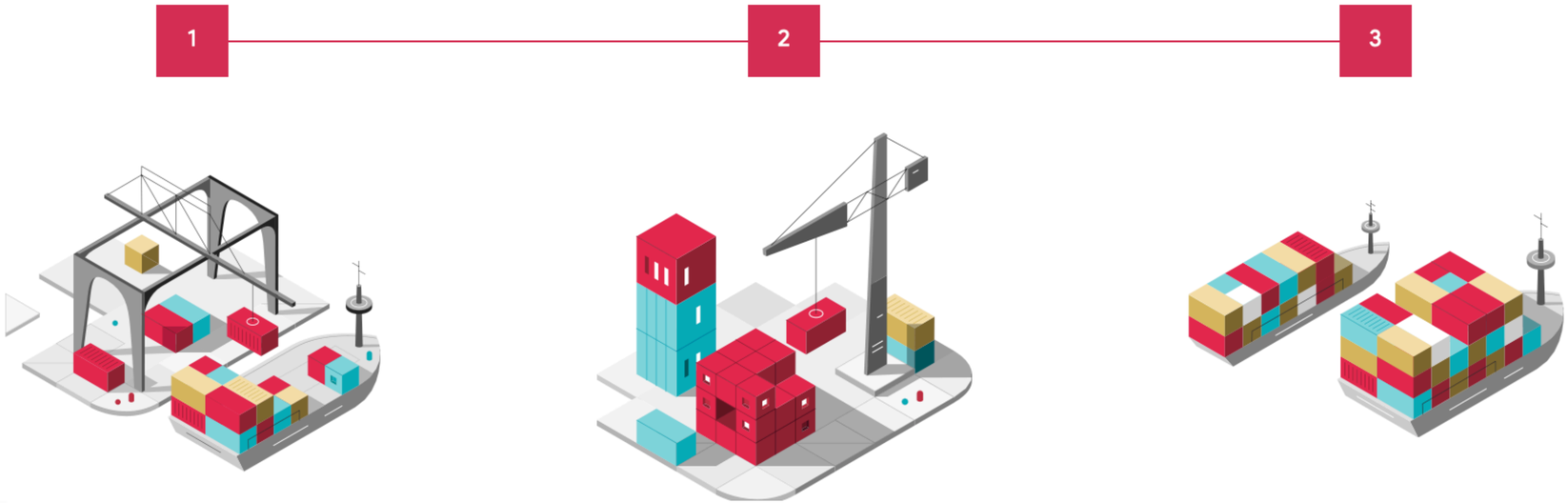
1. Agoric smart contracts engine
   - *Object capabilities for secure DEFI in JavaScript*
2. TLA+ master kernel model
   - *How TLA+ helps to understand the system*
   - *Experience with typed TLA+ using Apalache*
3. Kernel garbage collector
   - *Abstraction for practical verification*
4. Model-based testing of inter-VAT communication
   - *Generating and running thousands of tests from a model*

Agoric Smart Contracts Engine

Object Capabilities for secure DEFI in JavaScript

An Object-Capability (OCAP) is a transferrable, unforgeable authorization to use the object it designates.

- All communication between smart contracts is asynchronous and mediated
- Messages can only be sent along OCAP references
- Completely prevents certain kinds of smart contract attacks (such as Ethereum DAO)



Agoric

Stack Structure & Protocols

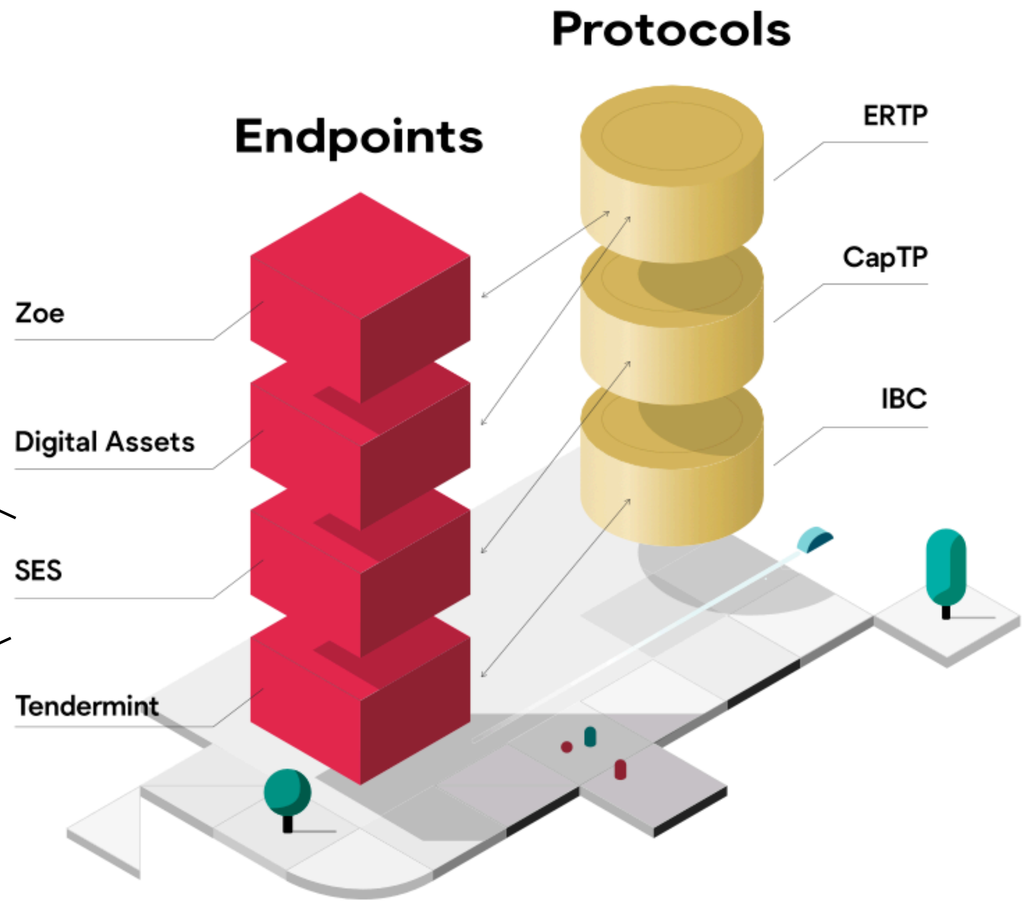
... administered by

... live in

... communicate via

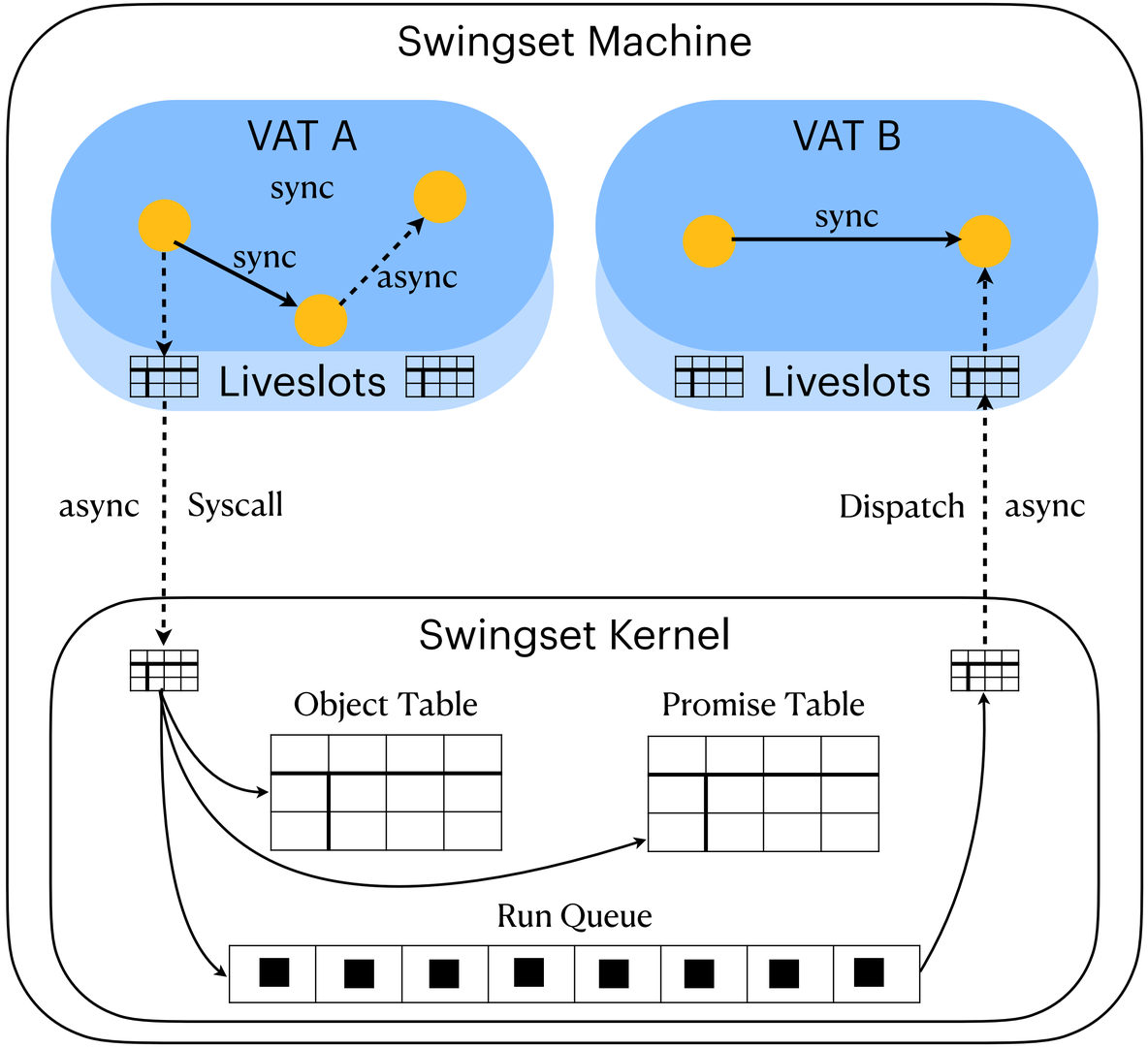
... that manages

Objects
Liveslots
VATs
Swingset Kernel
Swingset machine



Objects, VATs, Liveslots, Swingsets, Kernel

- **Objects** are normal JavaScript objects. They are submerged in SES (Secure ECMAScript) environment.
- **VATs** are units of synchrony. Synchronous communication occurs only within a single VAT.
- **Liveslots** mediate access of user-space VAT code with external world. Every access comes through translation.
- **Swingset Machine** may contain multiple VATs, and a shared kernel. It is a unit of determinism.
- **Swingset Kernel** orchestrates communication within a Swingset Machine, very much like a Unix kernel. Every *Syscall* (VAT to Kernel) or *Dispatch* (Kernel to VAT) comes through translation tables.



Agoric Swingset Kernel Model

How TLA+ helps to understand the system

Implementation vs. TLA+ model

Implementation / Documentation	Files	LOC
JavaScript	43	10000
Markdown	20	5000

vs.

TLA+ model	LOC	Comments / Types
Main model	350	250
Execution environment	350	200

Note: not all of the above implementation and documentation files were needed, but a large number of them had to be inspected for the model construction.

TLA+ model structure

- kernel_typedef.tla: types & definitions
- kernel.tla: main model, for each protocol action
  - Type
  - Pre-condition
  - Post-condition (update)
  - Changed/unchanged variables
- kernel_exec.tla: execution environment, a step for each protocol action
  - Existentially quantifies over action inputs
  - Stores inputs, checks the pre-condition, executes the update
- kernel_test.tla: model unit tests (sanity checks)

TLA+ model example: Send

Precondition

```
\* vatTranslator.js translateSend(): https://git.io/JZ8RN
\* In the real code the kernel slots are created on the fly by applying mapVatSlotToKernelSlot.
\* In the model we require this to be a separate Export action,
\* and assume that the kernel slots are already created when Send is done.
\* @type: (VAT_ID, KERNEL_SLOT, VAT_MESSAGE) => Bool;
SendPre(vatID, target, msg) ==
  /\ KnownVat(vatID)
  /\ target \in (DOMAIN kernelPromises \union DOMAIN kernelObjects)
  /\ target \in DOMAIN kernelToVat[vatID]
  /\ \/ msg.result = NULL_KERNEL_SLOT
     \/ /\ msg.result \in DOMAIN kernelPromises
        /\ kernelPromises[msg.result].state = UNRESOLVED
        /\ kernelPromises[msg.result].decider = vatID
  /\ \A slot \in msg.slots :
      /\ slot \in (DOMAIN kernelPromises \union DOMAIN kernelObjects)
      /\ slot \in DOMAIN kernelToVat[vatID]
```

TLA+ model example: Send

Changed variables & post-condition

```
SendChanges == <<kernelPromises, runQueue>>
SendNotChanges == <<await, terminationTrigger, koNextId, kpNextId, kernelObjects,
                    vats, kernelToVat, vatToKernel, vatReachSlots>>

\* kernelSyscall.js doSend(): https://git.io/JZllI
\* vatTranslator.js translateSend(): https://git.io/JZ8RN
\* @type: (VAT_ID, KERNEL_SLOT, VAT_MESSAGE) => Bool;
Send(vatID, target, msg) ==
  /\ \/ msg.result = NULL_KERNEL_SLOT /\ UNCHANGED kernelPromises
     \/ kernelPromises' = [kernelPromises EXCEPT ![msg.result].decider = NULL]
  /\ AddToRunQueue(SendMessage(target, msg))
```

TLA+ model example: Send

Protocol action & step

```
SendAction(vatID, target, msg) ==
  UNCHANGED SendNotChanges /\
    action' = [ type |-> "Send", vatID |-> vatID, target |-> target, msg |-> msg ] /\
    IF SendPre(vatID, target, msg) THEN
       /\ Send(vatID, target, msg)
       /\ error' = NULL
    ELSE
       /\ UNCHANGED SendChanges
       /\ error' = "Send"

SendStep ==
  UNCHANGED <<await, terminationTrigger>> /\
  \E vatID \in VAT_IDS:
  \E target \in KERNEL_SLOTS:
  \E msg \in VAT_MESSAGES:
    SendAction(vatID, target, msg)
```

How type checking ensures TLA+ model sanity

It is easy to make typing mistakes in TLA+ specs

```
@@ -174,9 +201,74 @@ CreateVat(vatID, enablePipelining) ==
174 201     /\ vats' = [k \in DOMAIN vats \union {vatID} |->
175 202         IF k = vatID THEN NEW_VAT(enablePipelining) ELSE vats[k] ]
176 203     /\ vatToKernel' = [k \in DOMAIN vatToKernel \union {vatID} |->
177     -       IF k = vatID THEN EMPTY_VAT_TO_KERNEL ELSE vatToKernel[k] ]
    204 +       IF k = vatID THEN EMPTY_VAT_TO_KERNEL_ENTRY ELSE vatToKernel[k] ]
178 205     /\ kernelToVat' = [k \in DOMAIN kernelToVat \union {vatID} |->
179     -       IF k = vatID THEN EMPTY_KERNEL_TO_VAT ELSE kernelToVat[k] ]
    206 +       IF k = vatID THEN EMPTY_KERNEL_TO_VAT_ENTRY ELSE kernelToVat[k] ]
```

The model checking with untyped specs (e.g. using TLC) may go fine.

```
Semantic processing of module kernel_typedef
Semantic processing of module kernel
Semantic processing of module kernel_exec
Semantic processing of module kernel_test
Starting... (2021-09-27 11:55:13)
Computing initial states...
Finished computing initial states: 1 distinct state generated at 2021-09-27 11:55:13.
Model checking completed. No error has been found.
```

Can you trust the results?

How type checking ensures TLA+ model sanity

Apalache type-checking to the rescue!

```
andrey@informal models % apalache-mc typecheck kernel.tla
# Tool home: /Users/andrey/work/apalache-0.15.11
# Package:   /Users/andrey/work/apalache-0.15.11/mod-distribution/target//apalache-pkg-0.15.11-full.jar
# JVM args:  -Xmx4096m -DTLA-Library=/Users/andrey/work/apalache-0.15.11/src/tla
#
# APALACHE version 0.15.11 build 2803ef0
#
PASS #0: SanyParser                                                  I@15:12:38.264
Parsing file /Users/andrey/work/agoric-sdk/packages/SwingSet/src/kernel/models/kernel.tla
Parsing file /Users/andrey/work/agoric-sdk/packages/SwingSet/src/kernel/models/kernel_typedef.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/Integers.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/FiniteSets.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/Sequences.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/TLC.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/Apalache.tla
Parsing file /private/var/folders/zf/cxj_wql52054hx66w0jtbh180000gn/T/Naturals.tla
PASS #1: TypeCheckerSnowcat                                          I@15:12:40.744
 > Running Snowcat .::.                                              I@15:12:40.744
[kernel.tla:169:7-169:63]: No match between operator signature ((Bool, a256, a256) => a256) and arguments
Bool and (Str -> ([exported: Bool, id: Int, type: Str] -> [id: Int, type: Str])) and ([exported: Bool, id:
 Int, type: Str] -> [id: Int, type: Str]) E@15:12:42.143
[kernel.tla:165:1-173:50]: Error when computing the type of CreateVat E@15:12:42.159
 > Snowcat asks you to fix the types. Meow.                          I@15:12:42.159
Type checker [FAILED]                                                I@15:12:42.160
```

How type checking ensures TLA+ model sanity

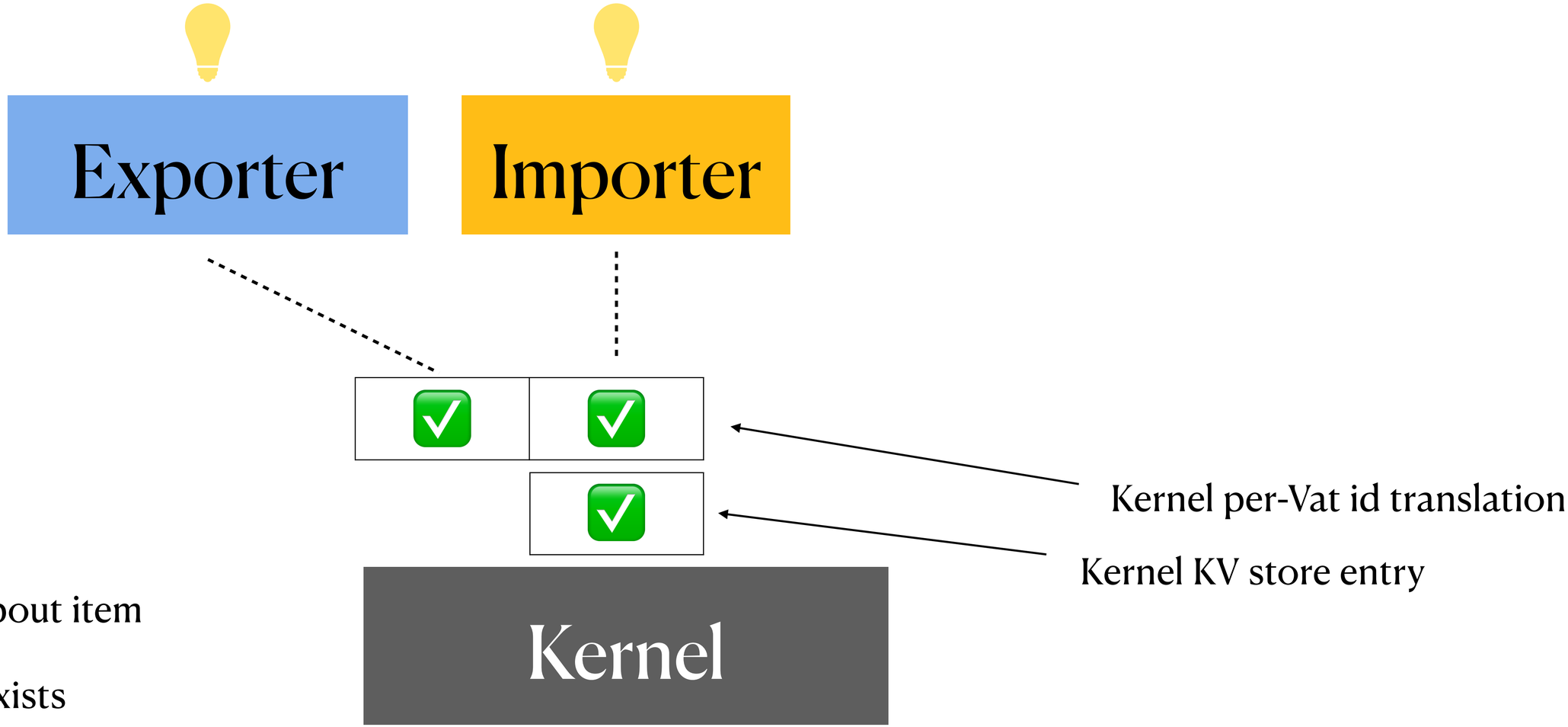
The type bug (one of many!) discovered by the Apalache type-checker

```
\* @typeAlias: VAT_TO_KERNEL = VAT_ID -> VAT_SLOT -> KERNEL_SLOT;

\* @type: () => VAT_SLOT -> KERNEL_SLOT;
EMPTY_VAT_TO_KERNEL_ENTRY == [ slot \in {} |-> NULL_KERNEL_SLOT ]
\* @type: () => VAT_TO_KERNEL;
EMPTY_VAT_TO_KERNEL == [ id \in {} |-> EMPTY_VAT_TO_KERNEL_ENTRY ]

VARIABLE
  \* A capability list from Vat slots to kernel slots: VAT_ID -> VAT_SLOT -> KERNEL_SLOT
  \* @type: VAT_TO_KERNEL;
  vatToKernel

\* kernel.js processCreateVat(): https://git.io/JnvdR
\* @type: (VAT_ID, Bool) => Bool;
CreateVat(vatID, enablePipelining) ==
  /\ vats' = [k \in DOMAIN vats \union {vatID} |->
      IF k = vatID THEN NEW_VAT(enablePipelining) ELSE vats[k] ]
  /\ vatToKernel' = [k \in DOMAIN vatToKernel \union {vatID} |->
      IF k = vatID THEN EMPTY_VAT_TO_KERNEL ELSE vatToKernel[k] ]
  /\ kernelToVat' = [k \in DOMAIN kernelToVat \union {vatID} |->
      IF k = vatID THEN EMPTY_KERNEL_TO_VAT ELSE kernelToVat[k] ]
  /\ vatReachSlots' = [k \in DOMAIN vatReachSlots \union {vatID} |->
      IF k = vatID THEN {} ELSE vatReachSlots[k] ]
```

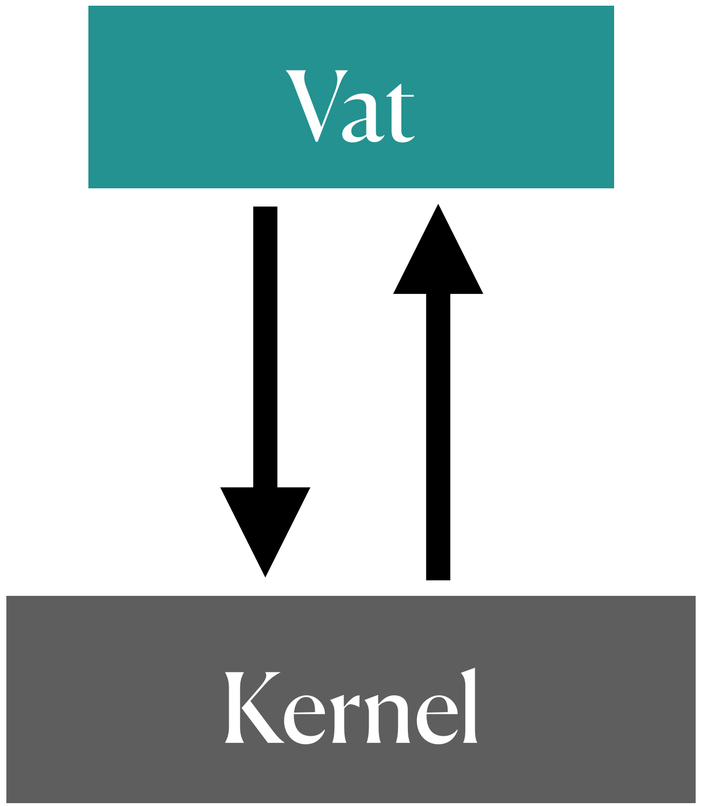
Kernel Garbage Collector

Abstraction for practical verification

Overview


Exporter
Importer
Kernel per-Vat id translation
Kernel KV store entry
Kernel
= Vat knows about item
= table entry exists

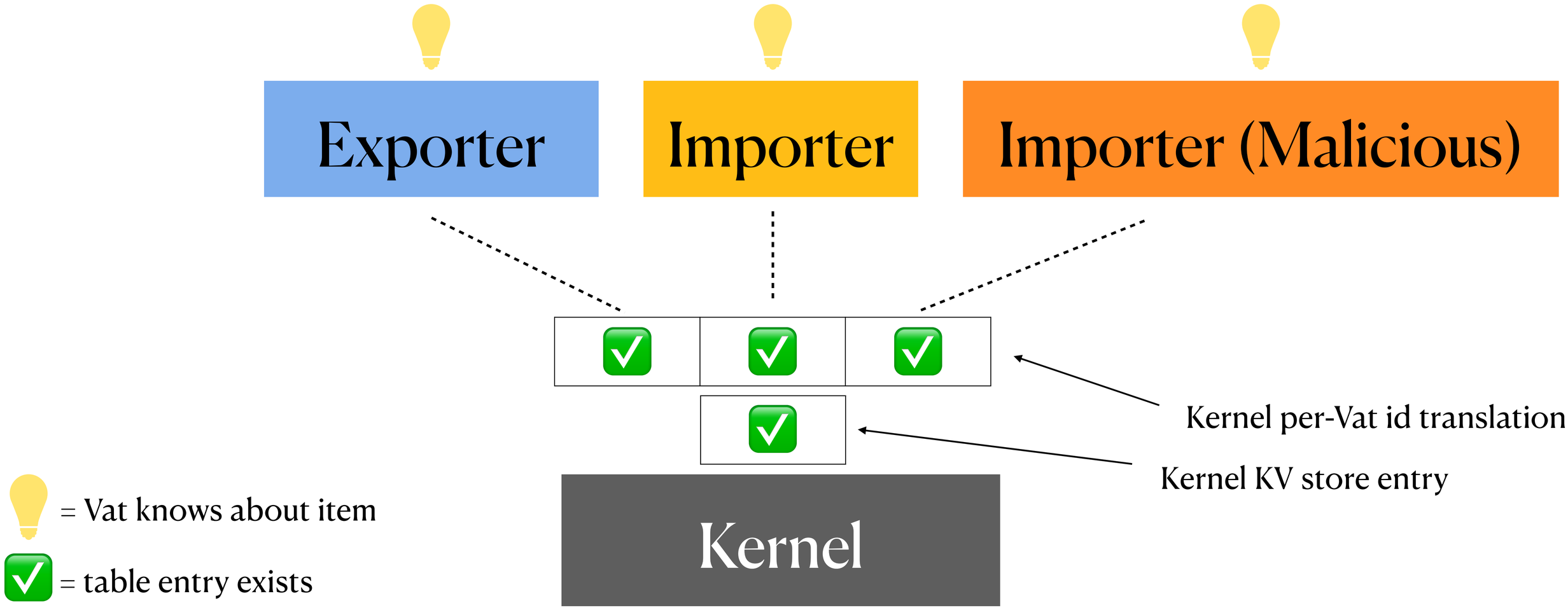
Overview



Syscalls:
- dropImport
- retireImport
- retireExport

Dispatch:
- dropExport
- retireImport
- retireExport

Malicious importer



Model structure

```
Init ==
    \* Important for TLA model only
    /\ mode = "tryGcAction"
    /\ step = "init"

    \* Kernel implementation sees these
    /\ gc_actions = {}
    /\ reachableCnt = 2
    /\ recognizableCnt = 2
    /\ exporter_reachableFlag = TRUE
    /\ importer_reachableFlag = TRUE
    /\ venomous_reachableFlag = TRUE
    /\ exporter_clist = TRUE
    /\ importer_clist = TRUE
    /\ venomous_clist = TRUE
    /\ kernel_obj = TRUE
    /\ maybeFreeKRef = FALSE

    \* Code of relevant vat sees these, kernel implementation does not
    /\ exporter_state = Known
    /\ importer_state = Known
    /\ exported_remotable = TRUE
```

```
Next ==

    \/ /\ mode = "tryGcAction"
       /\ mode' = "trySyscall"
       /\ TryGcAction

    \/ /\ mode = "trySyscall"
       /\ mode' \in {"handleKRef", "trySyscall"}
       /\ \/ Syscall
          \/ /\ UNCHANGED SemanticVars
             /\ step' = "skipSyscall"

    \/ /\ mode = "handleKRef"
       /\ mode' = "tryGcAction"
       /\ HandleKRef
```

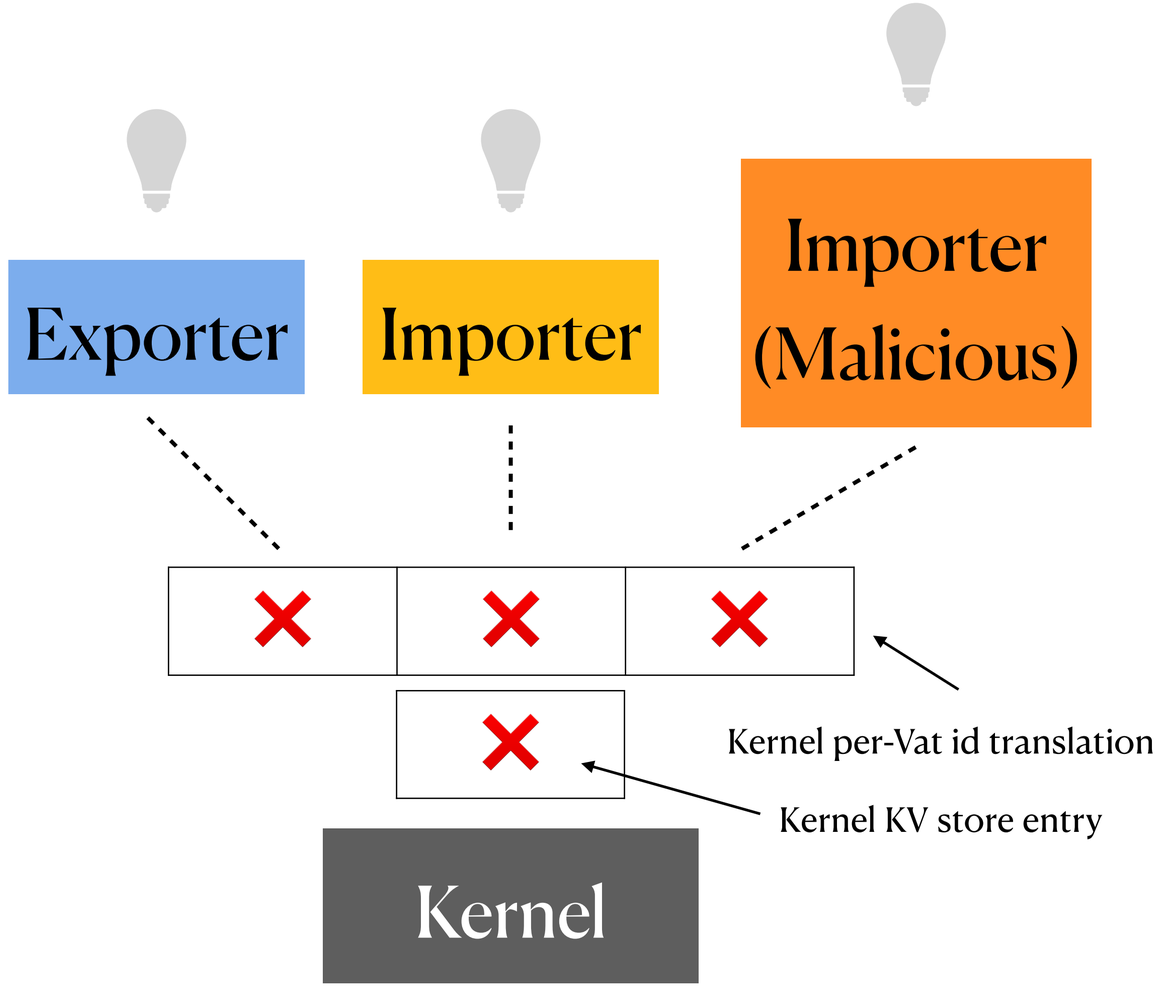
Maybe dispatch

Maybe syscall

Additional GC logic

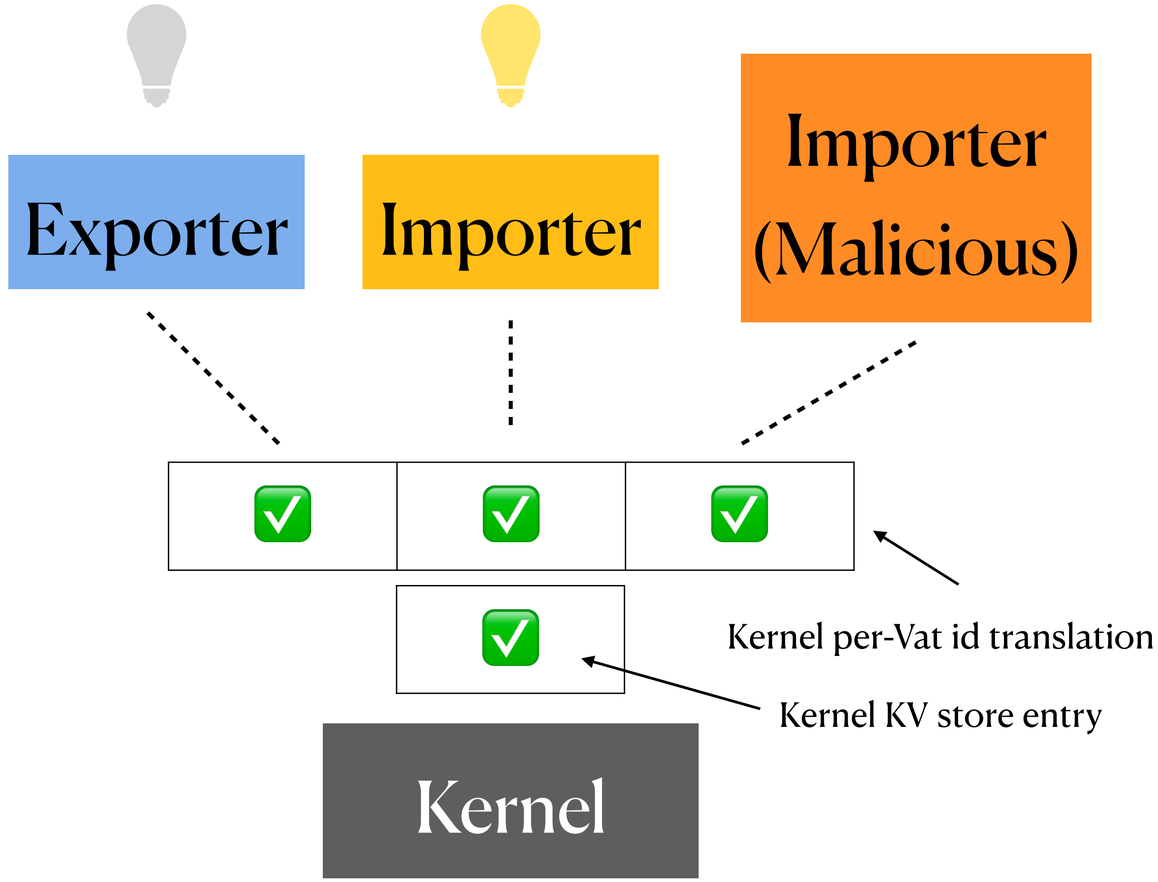
Invariants

```
EventuallyFreeEverything ==
    LET DesiredOutcome ==
        /\ importer_state = Unknown
        /\ exporter_state = Unknown
        /\ kernel_obj = FALSE
        /\ exporter_lookup = FALSE
        /\ importer_lookup = FALSE
        /\ malicious_importer_lookup = FALSE
    IN
    ~DesiredOutcome
```



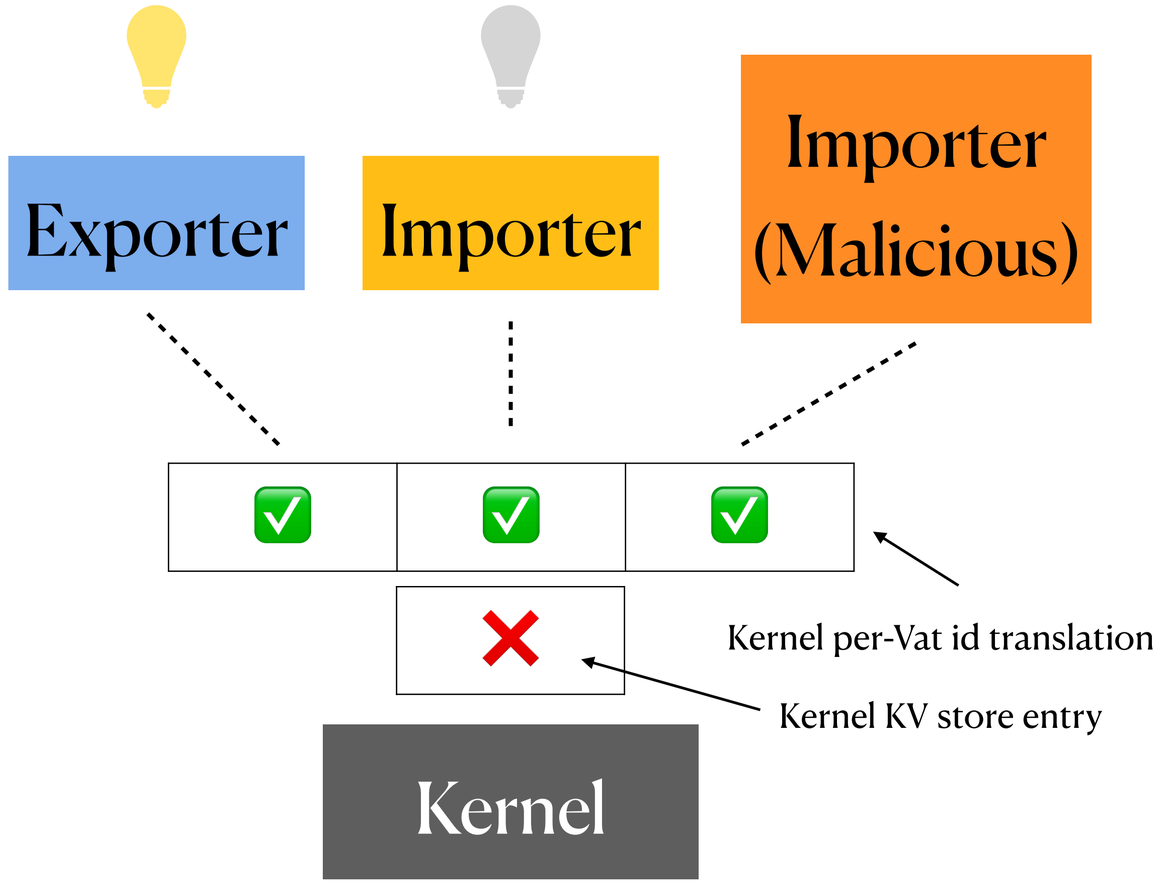
Invariants

```
ExporterForgetsEarly ==
    /\ \/ exporter_state = Unknown
       \/ ~exported_remotable
    /\ importer_state = Known

Inv == ~ExporterForgetsEarly
```



Invariants

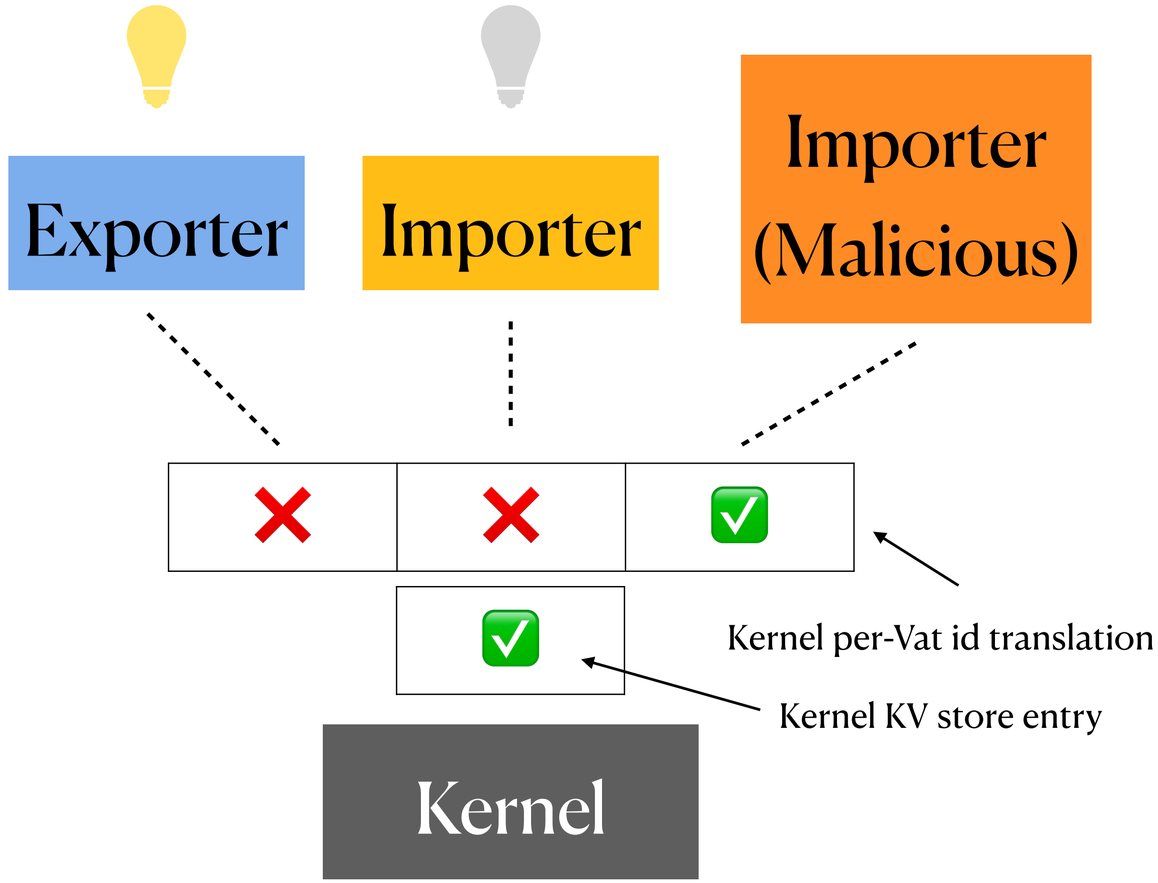
```
KernelUnsafeFree ==
    /\ ~kernel_obj
    /\ \/ importer_state = Known
       \/ exporter_state = Known
       \/ 0 < reachableRefCnt

Inv == ~KernelUnsafeFree
```



Kernel Garbage Collector

Invariants

```
LookupTableUnsafeFree ==
    \/ /\ ~importer_lookup
       /\ importer_state = Known
    \/ /\ ~exporter_lookup
       /\ exporter_state = Known

Inv == ~LookupTableUnsafeFree
```



Summary

- Lightweight model
- Easy checked with TLC in a second
- Models logic spread over ~4k lines of code in < 200 lines of TLA+!
- Gives high degree of confidence of protocol, in the face of difficult-to-reason-about process interleaving
- Final spec is a more precise description of the protocol than the documentation, and much easier to read

Status

Checking MC_kernel_gc.tla / MC_kernel_gc.cfg

Success : Fingerprint collision probability: 1.5E-14

Start: 14:57:29 (Sep 27), end: 14:57:29 (Sep 27)

States

Time	Diameter	Found	Distinct	Queue
00:00:00	13	1 183	311	0

Coverage

Module	Action	Total	Distinct
MC_kernel_gc	Init	1	1
MC_kernel_gc	Next	1 182	310

Model-based testing of inter-vat communication

Generating and running thousands of tests

Userspace code

VatA

...

```
await E(otherVat).foo({
  "baz": 42,
  "boz": "hello world",
  "biz": (x, y) => { bar(y, x) },
  "bez": new Promise((res, rej) => {
    //...
  })
})
```

...

Vat B

...

```
async foo(thing){

  wip(thing.baz)
  wop(thing.boz)
  await wap(thing.biz).flib()
  wep(thing.bez)

  // ... ect
  // almost arbitrary JS code

}
```

...

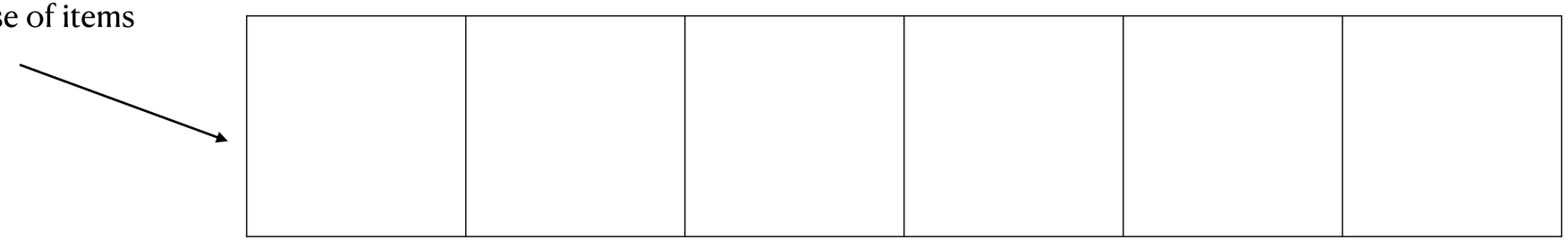
Decomposition of behaviours

Vat A

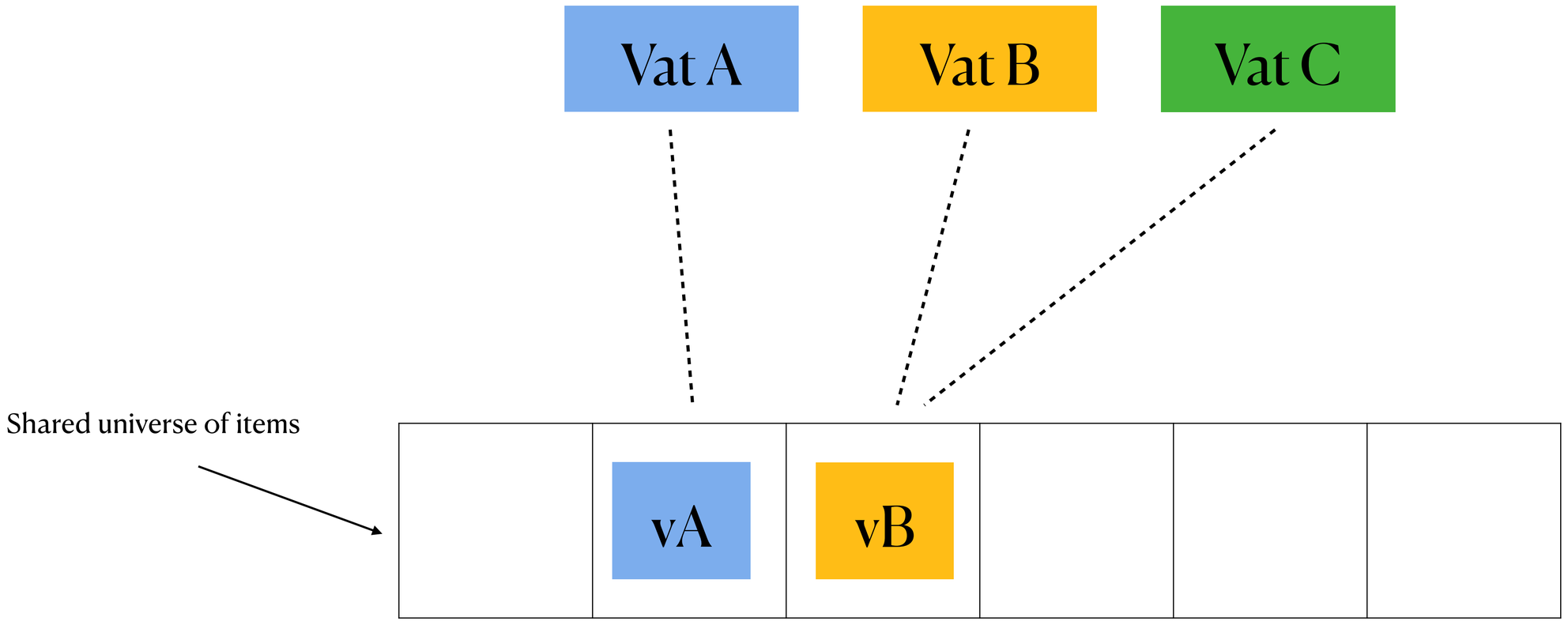
Vat B

Vat C

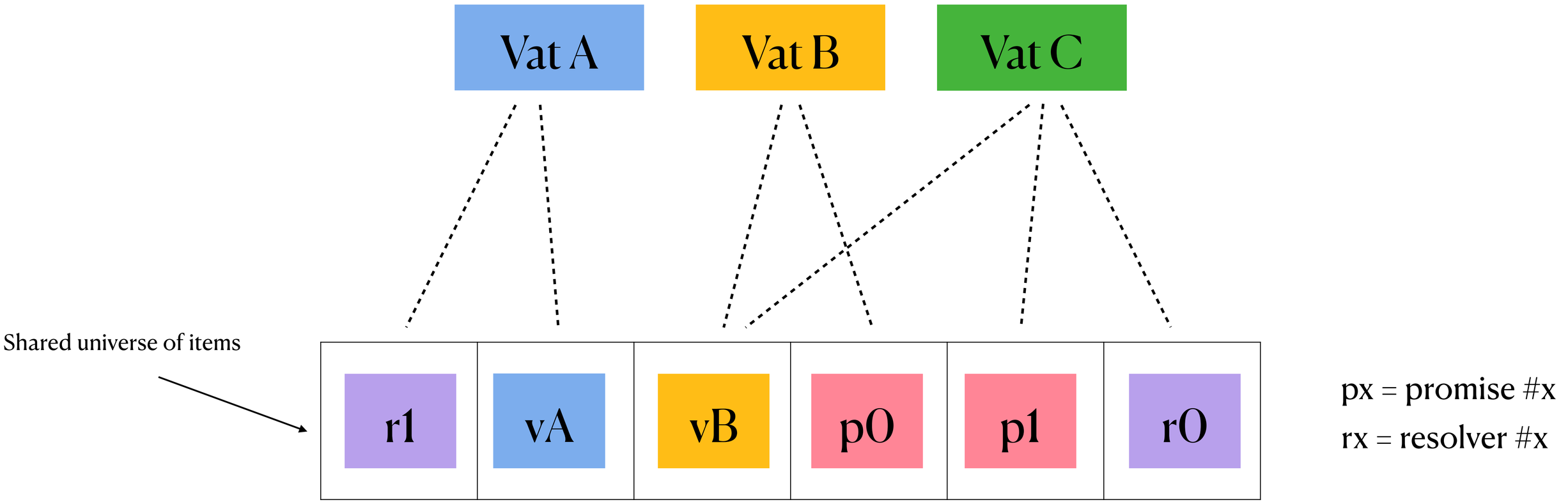
Shared universe of items



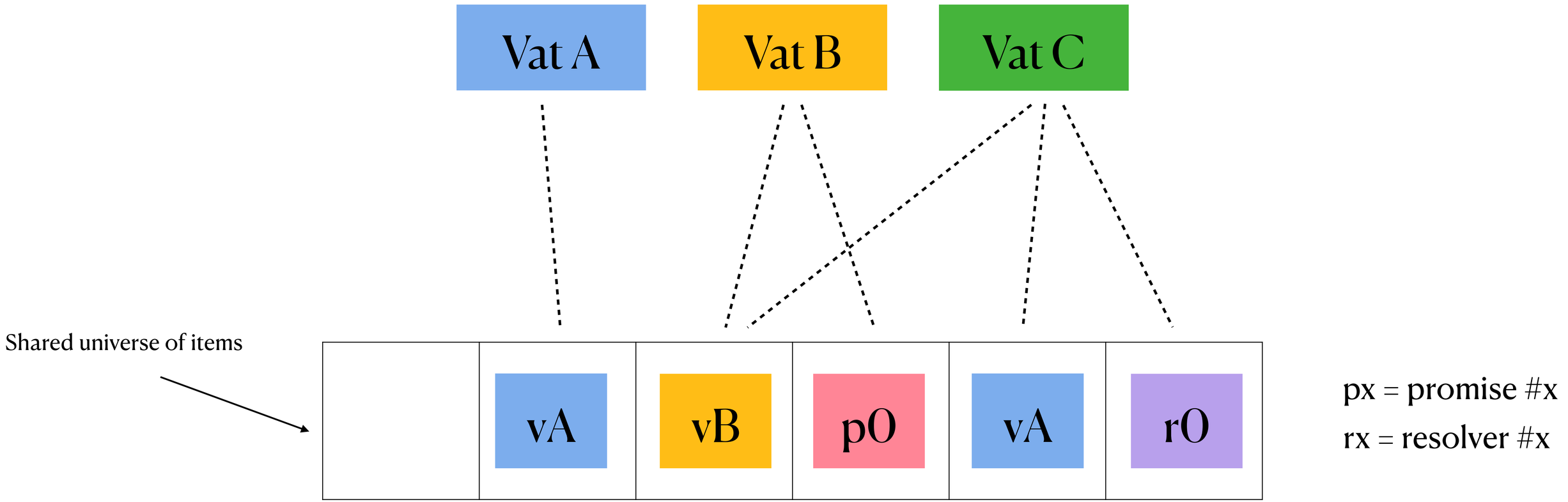
Decomposition of behaviours



Decomposition of behaviours



Decomposition of behaviours



Decomposition of behaviours

```
Next ==
    \/ SendItemStep
    \/ DropItemStep
    \/ StoreSelfReferenceStep
    \/ StorePromiseStep
    \/ ResolveStep
    \/ TransferControlStep
```

```
async sendItem(targetVatId, itemId) {
  await E(v).sendItemToVat(items.get(itemId), items.get(targetVatId))
},

async storeSelfRef(itemId) {
  items.set(itemId, selfRef)
},

async dropItem(itemId) {
  items.delete(itemId)
},

async storePromise(promiseId, resolverId) {
  const { promise, res } = createPromiseParts()
  let promise = new Promise((resolve, _) => {
    res = resolve;
  });
  items.set(promiseId, promise)
  items.set(resolverId, res)
},

async resolve(resolveItemId, resolverId) {
  await E(items.get(resolverId))( items.get(resolveItemId))
},
```

Generating executions with the Apalache model checker

- 3 components
  - Historical traces
  - Able to generate multiple executions for each invariant
  - ‘View’ projection allows control over counterexample differentiation

Generating executions with the Apalache model checker

```
\* @type: Seq(STATE) => Bool;
TraceResolvesPromiseSeenByOther(hist) ==
    LET
        \* @type: (BANK, Int) => Set(VAT);
        WatchersForPromiseId(bank_, promiseId) == bank_[PromiseIxOrNeg1(bank_, promiseId)].watchers
    IN
    LET
        Behavior ==
        /\ \E i, j \in DOMAIN hist:
           /\ (i + 1) \in DOMAIN hist
           /\ i < j
           /\ \E k \in DOMAIN hist[i].bank:
              /\ hist[i].bank[k].type = "resolver"
              /\ hist[i+1].bank[k].type = "blank"
              \* Ensure that another vat has a turn after the resolving vat
              /\ hist[i+1].curr # hist[j].curr
              /\ hist[j].step # "transferControl"
              \* Find a case where a vat resolves a promise for which it is not the only watcher
              /\ {hist[i].curr} # WatchersForPromiseId(hist[i].bank, hist[i].bank[k].promiseId)
    IN
    ~Behavior
```

Promise resolution

Acting vat is different in states [I+1] and [j]

jth step is not merely a transfer of control

Some other vat has access to the resolved promise

Generating executions with the Apalache model checker

```
ExploratoryView == IF step_cnt < 4 THEN "" ELSE step
```

“States A and B are different if the *step* variable is different and one of the states has a *step count* of 4 or more.”

Effort to reward ratio

- ~ 200 lines driver code
- ~ 300 lines glue code
- ~ 200 lines TLA+
- Generate a trace in seconds/minutes

- Generate thousands of tests for a given behaviour
- Behaviours can be as complex or as simple as wished
- A very abstract model can generate tests for a complex runtime environment
- Tests may be easily incorporated into a CI or regression suite

Thank you!

- agoric.com
- agoric-sdk/SwingSet/kernel
- informal.systems
- apalache.informal.systems
- github.com/informalsystems/modelator
- informal.systems/services/security-audits

Check it out & happy to connect

andrey / daniel @ informal.systems

Agoric

informal SYSTEMS