Formal Methods Assessment Report

Agoric Swingset Kernel, Comms and Userspace. Phase 3: Integration of Adversarial Testing of Userspace Vat Interaction. Modeling and Verification of Garbage Collection Protocols in Comms and the Kernel.

2.2.2022
Last revised 2.2.2022

Authors: Daniel Tisdall

Contents

Formal Methods Assessment Overview

The Project

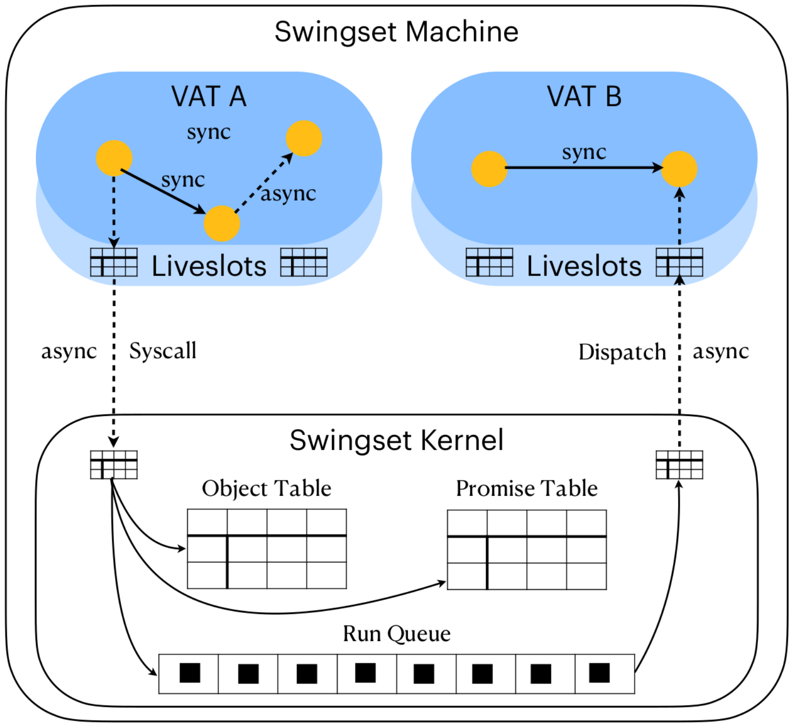
In April 2021, Agoric engaged Informal Systems to conduct an ongoing formal methods assessment of the documentation and the current state of the implementation of the Agoric SwingSet platform kernel and vat interactions. This document reports on the work done in phase 3 of the assessment.

The Agoric SwingSet Platform

The Agoric SwingSet is the basis for development with the Agoric SDK, a development kit for writing smart contracts in a secure, hardened version of JavaScript. Smart contracts built with the Agoric SDK have mediated asynchronous communication, and messages can only be sent to references according to the rules of the Object Capabilities model that the SwingSet implements.

The Object Capabilities (OCAP) model is a model for reasoning about communication. An Object-Capability is a transferable, unforgeable authorization to use a designated object. The SwingSet machine allows JavaScript code to communicate according to the model, and executes code in a userspace similar to the one offered by a Unix operating system. The SwingSet kernel component operates analogously to a Unix kernel, and vats correspond to Unix userspace processes. The kernel provides services for isolation, composition, and communication between vats: it enforces the OCAP properties.

The figure below shows a diagram of the architecture of the SwingSet kernel and two interacting vats. Each vat is a unit of synchrony and synchronous communication only occurs inside a single vat. The liveslots layers mediate access to the outside world from userspace code. Every remote object access is implemented via translation tables in the liveslots layers and the kernel, which operates by pulling work off from a run queue.



Scope of this report

This report covers phase 3 of the analysis. Phase 1 and phase 2 results are covered in previous reports. The agreed-upon work plan for phase 3 consisted of the following tasks:

1. Adversarial Userspace Testing Integration
2. Garbage Collection Code Inspection (Kernel)
3. Garbage Collection Code Inspection (Comms)
4. High Fidelity Garbage Collection Modeling (Kernel)
5. High Fidelity Garbage Collection Modeling (Comms)

The tasks were conducted 01.10.2021 through 23.12.2021 by Informal Systems by the following personnel:

- Daniel Tisdall: Research Engineer at Informal Systems

Conducted work

Starting 1st October, Informal Systems conducted a formal methods assessment of the existing documentation and the code. Our team began by reviewing the SwingSet package documentation to refamiliarise ourselves with aspects of the system not focused on in phase 2.

After developing an understanding, work began to integrate the adversarial tests and test driver into the Agoric SDK official code repository, to exercise the userspace capabilities of inter-communicating vats. This work includes a TLA+ model, as well as vat code written in Javascript, and some additional scripts written in Python and Bash for pre and post processing steps.

Additionally, we inspected documentation and code relevant to the working of the garbage collector protocols, restricted to the parts that relate to the kernel, vats and comms vat, but not including the virtual object manager or liveslots. We created a model of garbage collection protocol flows in the comms vats, and also created a high fidelity model of garbage collection protocol flows in the kernel. This model is more detailed than the kernel garbage collection model created and model checked in phase 2.

Timeline

- 01.10.2021: Start of formal methods assessment
- 15.12.2021: Technical discussion between Daniel and Agoric
- 23.12.2021: End of formal methods assessment
- 01.02.2022: First draft of this report

Conclusions

Overall we found the code to be organized, reasonably documented, and faithful to the specification. Despite the general high quality of the implementation work, we found several significant issues regarding code quality, code organization, and divergence from the specification. These points have been previously addressed in the report from phase 2 of the formal methods assessment. Departing from these issues, we did not find any semantic errors in the garbage collection protocols of either the kernel or comms vats, nor the inter-vat communication implementation. None of the pathways tested or modelled deviated from the expected behavior.

Dashboard

Target Summary

- **Name**: Agoric SwingSet inter-vat communication in userspace code and garbage collection protocols
- **Type**: Documentation and implementation
- **Platform**: Javascript
- **Artifacts**:
  - Agoric SDK at release 10.0.2

Engagement Summary

- **Dates**: 01.10.2021 – 23.12.2021
- **Method**: Manual review, formal protocol modeling, protocol verification and adversarial testing integration
- **Employees Engaged**: 1
- **Time Spent**: 6 Person Weeks

Engagement Goals

The scope of the assessment developed over time as a result of a series of meetings between the Agoric and Informal teams. The highest priority aspects of the system were determined to be

1. The garbage collector protocol as it applies to comms vats.
2. The garbage collector protocol implementation in the kernel.
3. Integration of test-suite using model generated tests (model-based testing).

It was determined that Informal Systems would analyse 1) and 2) via protocol model checking, and address 3) by integrating the model-based testing implementation of phase 2 into the official Agoric SDK code repository.

Coverage: Integration of Model-Based Testing tests and driver into Agoric SDK test

Overview of work done

We integrated the model-based testing approach from phase 2 into an automated test inside the Agoric SDK source repository. The test exercises parts of the kernel and liveslots code that are most important for correctly implementing the userspace API. We made small improvements to the model and code from phase 2 which do not change the semantics substantially but improve readability and improve the effectiveness of the approach. In particular, we have changed the model and test generation code to now benefit from the TLC model checker which is able to generate tests of much longer length. In phase 2 we exercised the system using executions consisting of at most 9 steps, but now executions consisting of 15 steps are tested.

The model-based tests stress the ability of the kernel and liveslots implementations to correctly implement behaviors visible in userspace. The tests are useful to quickly catch any regressions that may be introduced into the SwingSet codebase in future.

Please see the report of phase 2 for more details on the model and testing, as these components have not changed significantly. Please see the pull request for more details on the integration into the Agoric SDK codebase.

Artifact location

Pull request

Coverage: Modeling and Verification of Garbage Collection Protocol in Comms

Overview of work done

We created an abstract TLA+ model of the garbage collection protocol between comms vats over an object lifetime. The term Comms denotes the components of the SwingSet that together implement communication between different SwingSet machines. For instance, it is possible for a vat operating on one SwingSet machine to call a method on a reference provided to it by a vat on another SwingSet machine. A garbage collection solution is therefore needed to alert vats on remote machines when they are able to forget imported references.

The TLA+ model models logical flows that can occur in a universe consisting of four comms vats and one kernel who may be aware of the lifetime of a shared object. One vat is implicitly denoted the root exporter. The root exporter vat is the vat for which the kernel is their local kernel. The object in question has been exported by the kernel, and must be made available remotely. The model captures behaviors where the comms vats reference the object in messages to each other. The garbage collection messages are modelled explicitly while the `SendUse` operator in the model models a real system referencing the object in remote `Send` messages.

Protocol and diagram

The garbage collection protocol depends on vats correctly sending one of three message types to each other at the correct times, as prescribed by the algorithm.

- **dropExport**
  Used by object importers to alert upstream exporters that their export is no longer needed.
- **retireExport**
  Used by object importers to alert upstream exporters that their own local handle to the object is no longer needed.
- **retireImport**
  Used by object exporters to alert downstream importers that their own local handle to the object is no longer needed.

Due to network latency, it is possible for messages to 'cross over the wire'. For example, if `vat A` sends `message A` to `vat B` and then `vat B` sends `message B` to `vat A`, it is possible for `message B` to arrive at `vat A` before `message A` arrives at `vat B`. For this reason, sequence numbers accompany messages. This gives vats the option to ignore certain inbound messages if the associated sequence number is sufficiently out of date.

The diagram shows an artistic interpretation of a snapshot of model behavior. Four vats are connected by first-in-first-out communication channels, and each vat maintains data necessary to run the protocol (depicted with boxes `owner`, `clists`, `lastUsage`, `isReachable`, `sentCnt` and `receiveCnt`). The diagram is asymmetric because one vat is denoted the exporter (vat 0 in this instance) and their owner variable will be set to the *kernel* sentinel value. The variables meaning closely matches the meaning in the source code implementation:

- `owner`
  Tracks the source of the object imported from a remote machine. This will either be a vat identifier or the *kernel* sentinel value.
- `clists`
  Tracks the existence of translation table data needed to interpret messages from a remote machine that reference the object.
- `lastUsage`
  Tracks the sequence number of the last outbound application level message that referenced the object.
- `isReachable`
  Tracks whether a remote vat should be able to reach the referenced object.

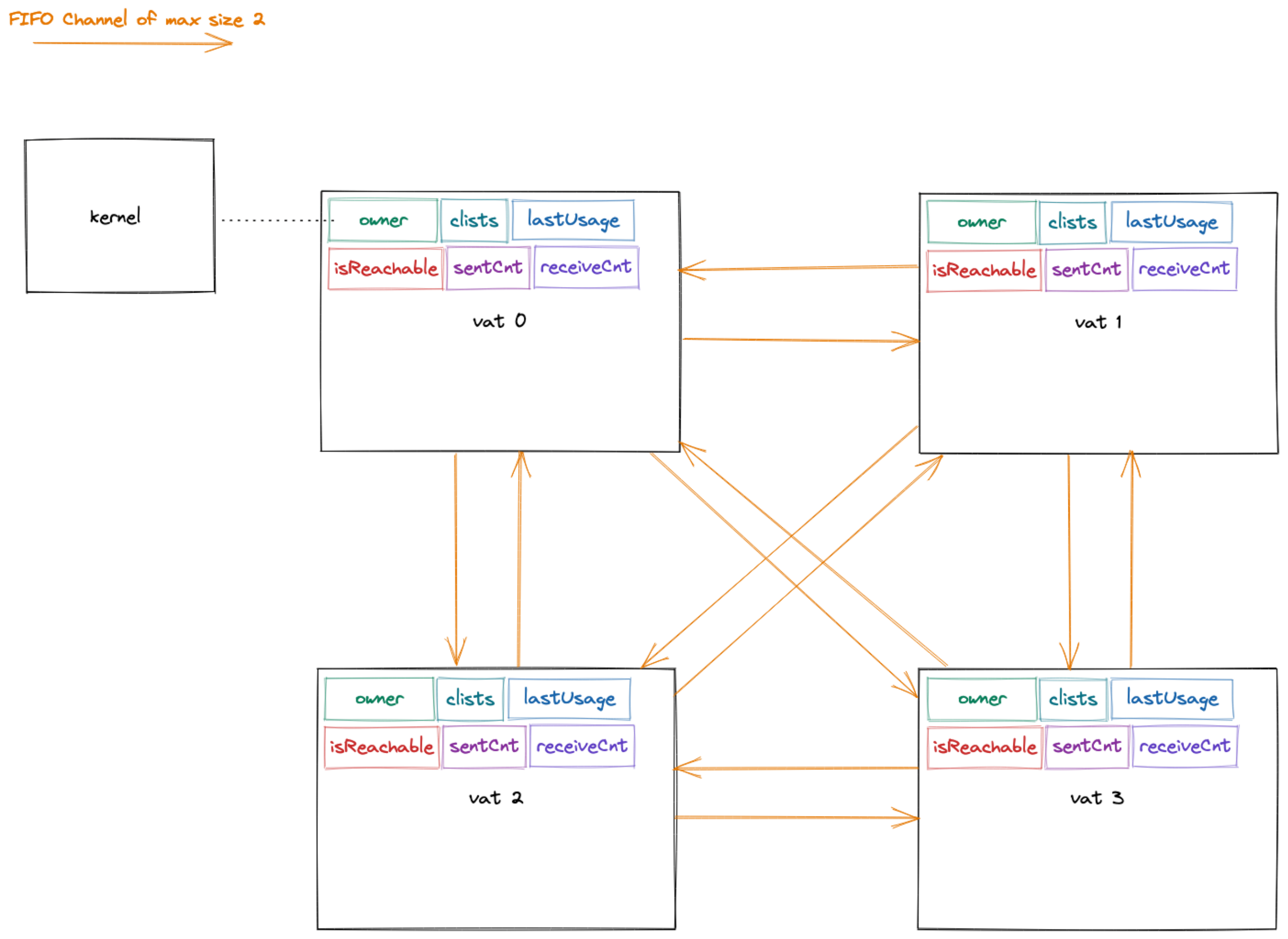


Figure 1: A diagram depicting a state of the comms model.

- `sentCnt`
  Tracks the number of messages sent on a given fifo queue.
- `receiveCnt`
  Tracks the number of messages received on a given fifo queue.

Parameters

The model captures interleavings of steps among the comms vats, and thus checks corner cases that may be present due to concurrency in the real world. The model is parameterized with three variables

1. `VATS == {"v0", "v1", "v2", "v3"}`
   The set of comms vats (4 by default).
2. `FIFO_LIMIT == 2`
   Each pair of vats communicates via two FIFO queues, one for each communication direction. In order to make model checking feasible the length of the queues is limited. Processes will not perform *send* steps when the relevant queue is full.
3. `SEND_USE_LIMIT == 2`
   We bound execution in order to prevent monotonically increasing sequence numbers in the protocol from creating an infinite state space. In this way we disallow infinite executions which are allowed in the real system by limiting the number of times each comms process may introduce the object to a third party process. Such infinite executions trivially preclude liveness (object is eventually garbage collected) and are not interesting to model check.

Results

We checked the model using the TLC model checker. In particular, we checked

1. An execution exists where the object lifetime cycle is eventually complete (the object is freed by all processes)

An execution where the object lifetime cycle eventually completes is a successful execution.

```
(*
    Check that an execution exists where the object lifetime is successfully completed. *)
GarbageCollectionTrace ==
    LET Behavior ==
        /\ trueState = [v \in VATS |-> TRULY_UNKNOWN]
        /\ isReachable = [p \in VatPairs |-> FALSE]
        /\ clistExists = [p \in VatPairs |-> FALSE]
    IN ~Behavior
```

TLC was able to generate an execution satisfying the correct behavior, requiring 17 steps. The model was parameterized with 4 vats, `FIFO_LIMIT` = 3, `SEND_USE_LIMIT` = 3.

2. Among all executions, there is never a state where a vat has imported the object and simultaneously no path exists (through clists) for that vat to refer to the object in a message to the kernel

If a vat has imported an object it must be able to refer to that object in a message to the vat whose owner is the *kernel*. The communication pathway between the importing vat and the *kernel* may actually be made of a number of hops between pairs of vats who simultaneously import the object from an upstream vat and export the object to a downstream vat. Each of these exporting and importing relationships must be accompanied with `clist` data, in order to translate messages correctly. The property that this data must always exist along the entire communication path is an instance of the commonly known property '*no use after free*'.

```
(*
    If some vat can reach the object then there should be some communication path
    for that vat to refer to the object in a message to the objects original owner (_kernel).
@type: () => Bool; *)
UsefulReferencePathExistsWhenItShould ==
    (*
    There is a path of clists leading back to the kernel?
    *)
    LET
    \* @type: (VAT) => Bool;
    ExistsPathThroughClistsToKernel(v) ==
        LET
        (*
        Compute transitive closure of the directed graph formed by edges.
        @type: Set(<<VAT,VAT>>) => Set(<<VAT,VAT>>); *)
        transitiveClosure(edges) ==
            IF edges = {} THEN {}
            ELSE LET
            RECURSIVE
            \* @type: (Set(<<VAT,VAT>>), Int) => Set(<<VAT,VAT>>);
            Combine(_, _)
            \* @type: (Set(<<VAT,VAT>>), Int) => Set(<<VAT,VAT>>);
            Combine(acc, k) ==
                IF k = 0
                THEN acc
                ELSE Combine(acc \cup { <<s[1],t[2]>> : s \in acc, t \in edges }, k - 1)
            IN Combine(edges, Cardinality(edges) - 1)
        IN
        \E edge \in transitiveClosure({p \in VatPairs : clistExists[p]}):
        /\ edge[1] = v
        /\ owner[edge[2]] = _kernel

    IN \A v \in VATS: trueState[v] = TRULY_REACHABLE =>
        \/ owner[v] = _kernel
        \/ ExistsPathThroughClistsToKernel(v)
```

TLC checked all executions of the system in 1 hour and 12 minutes, checking 82683592 distinct states in total. The model was parameterized with 4 vats, `FIFO_LIMIT` = 2, `SEND_USE_LIMIT` = 2. No violation of the invariant was found.

Summarizing: no execution was found that deviated from the expected behavior.

Target

- Agoric SDK 10.0.2

Artifacts

- Model directory

Findings

1. SwingSet comms: documentation in garbage-collection.md for comms unclear #4198

Recommended usage

Use the TLC model checker to check properties of the model.

```
java -XX:+UseParallelGC -Xmx12g -cp tla2tools.jar tlc2.TLC -workers auto comms.tla
```

Properties, invariants and other configurations can be set in `comms.cfg`.

Coverage: High Fidelity Modeling and Verification of Garbage Collection Protocol in Kernel

Overview of work done

We created an abstract but detailed TLA+ model of the garbage collection protocol inside the kernel that tracks the collection of objects in vats and the kernel and promises in the kernel. A large part of the work involved in creating the model was a detailed read of the documentation and source code in the SwingSet package. As part of this process we extracted important logic and wrote it in Java code, in order to improve our understanding, especially in areas of the code that were difficult to understand due to the lack of types or for other reasons.

The model models logical flows that can occur in a universe consisting of three vats and one kernel who may be aware of the lifetime of three objects and three promises. The model captures behaviors where the vats reference objects and promises in syscalls, and the kernel responds appropriately and processes its internal run queue. Each syscall that can be made by a vat into the kernel is modelled in detail: the effects on the kernel's internal data structures are captured.

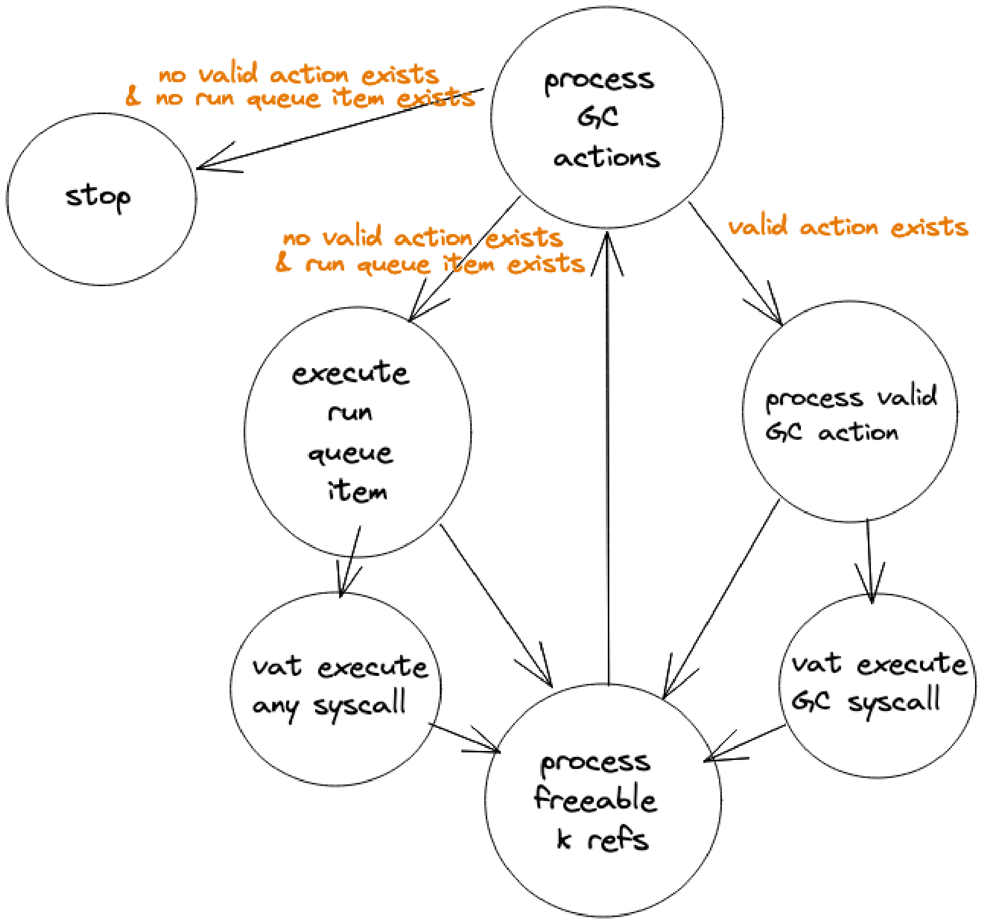
The model is structured by way of an abstract state machine that is the union of kernel and vat behavior. Each state in the abstract state machine allows a different action to transition to the next state. The set of all possible actions is as follows

- **processGcActions**
  Process the gcActions set, deleting invalid actions and collecting valid ones.
- **executeGcAction**
  Execute a valid garbage collection action.
- **processMaybeFree**
  Garbage collect hanging promises in the kernel and add relevant garbage collection tasks to the gcActions set for hanging objects.
- **executeRunQueueEntry**
  Process a task from the kernel run queue.
- **dropImportsSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **retireImportsSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **retireExportsSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **sendSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **subscribeSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **resolveSyscall**
  Mirrors relevant code, including slot translation and reference count changes.
- **tryRunLoopBodyButNothingToDo**
  This action can occur if the kernel has execution authority but there is no task on the run queue or valid garbage collection action.
- **nothing**
  This action can occur if there is no work on the kernel run queue and no vat makes a syscall.
- **skipSyscall**
  This action can occur if a vat has execution authority but does not make a syscall.

In any single state of the abstract state machine a subset of these actions will be enabled.

Diagram

The model contains two variables which control the high level state machine of the models operation: `executor` and `executorPermittedOperations`. The `executor` is always either the kernel, or a particular vat. The `executorPermittedOperations` is a string whose value controls the possible actions that can taken in each step of model execution.



This abstract state machine is captured entirely in the `Next` operator of the TLA+ model. Auxiliary operators capture the data structure updates.

Parameters

The model captures every execution in the kernel that can result from any combination of vat behaviors (following liveslots rules). This means that all combinations and orderings of legal syscalls are explored. The model contains parameters

1. `VAT_IDS == {"v0", "v1", "v2"}`
   The set of vats (3 by default).
2. `PROMISE_IDS == {"p0", "p1", "p2"}`
   The set of ids reserved for promises. In the model it was not necessary to differentiate between vat local identifiers and the kernel identifiers, as much of the vat-kernel translation logic has been abstracted.
3. `OBJECT_IDS == {"o0", "o1", "o2"}`
   The set of ids reserved for objects.
4. `SYSCALL_LIMIT == 3`
   The number of syscall that each vat can make. As we do not model metering we must bound the number of steps that a single vat can remain the `executor`. This is done by setting a limit on the number of syscalls made by the vat.

Status

The model has so far been developed solely by Informal Systems, and Informal Systems has successfully sanity checked the model using a small parameter configuration. It was discovered that some properties are violated when using larger parameter combinations. Due to the high fidelity of the model and the complexity of the system and code being modelled it is necessary to closely collaborate with Agoric to further develop the model. This is needed to ensure that the model faithfully captures the intent of the Agoric team, and that the model does contain inaccuracies not present in the code.

Target

- Agoric SDK 10.0.2

Artifacts

- Model directory
- Java and JavaScript code directory

Recommended usage

Use the TLC model checker to check properties of the model.

```
java -XX:+UseParallelGC -Xmx12g -cp tla2tools.jar tlc2.TLC -workers auto kernel_enhanced.tla
```

Properties, invariants and other configurations can be set in `kernel_enhanced.cfg`.