Formal Methods Assessment Report

Agoric Swingset Kernel and Userspace, Phase 2: Adversarial Testing of Userspace Vat Interaction, and Modeling & Verification of Garbage Collection Protocol

09.11.2021

Initial revision: 21.10.2021

Authors: Daniel Tisdall, Andrey Kuprianov

Contents

Assessment Overview 3
The Project . . . 3
The Agoric SwingSet Platform . . . 3
Scope of this report . . . 4
Conducted work . . . 4
Timeline . . . 5
Conclusions . . . 5

Assessment Dashboard 6

Engagement Goals 7

Coverage 8
(MBT) Adversarial testing of inter-vat communication in userspace . . . 8
(MBT) Overview . . . 8
(MBT) Results . . . 9
(MBT) Method . . . 10
(GC) Model-checked TLA+ model of garbage collector protocol . . . 10
(GC) Overview . . . 10
(GC) Results . . . 11
(GC) Method . . . 12

Findings 13

IF-AGORIC2-01: Unprincipled use of OOP concepts has likely created technical debt 14
Involved artifacts . . . 14
Description . . . 14
Recommendation . . . 15

IF-AGORIC2-02: Make the code in gc-actions.js easier to understand 17
Involved artifacts . . . 17
Description . . . 17
Recommendation . . . 18

IF-AGORIC2-03: Do not overload isReachableFlag 20
Involved artifacts . . . 20
Description . . . 20
Recommendation . . . 20

IF-AGORIC2-04: retireImports dispatch does not have any effect on liveslots 21
Involved artifacts . . . 21
Description . . . 21
Recommendation . . . 21

IF-AGORIC2-05: Check the correctness of the retireImports syscall code with respect to timing 22
Involved artifacts . . . 22
Description . . . 22
Recommendation . . . 23

Assessment Overview

The Project

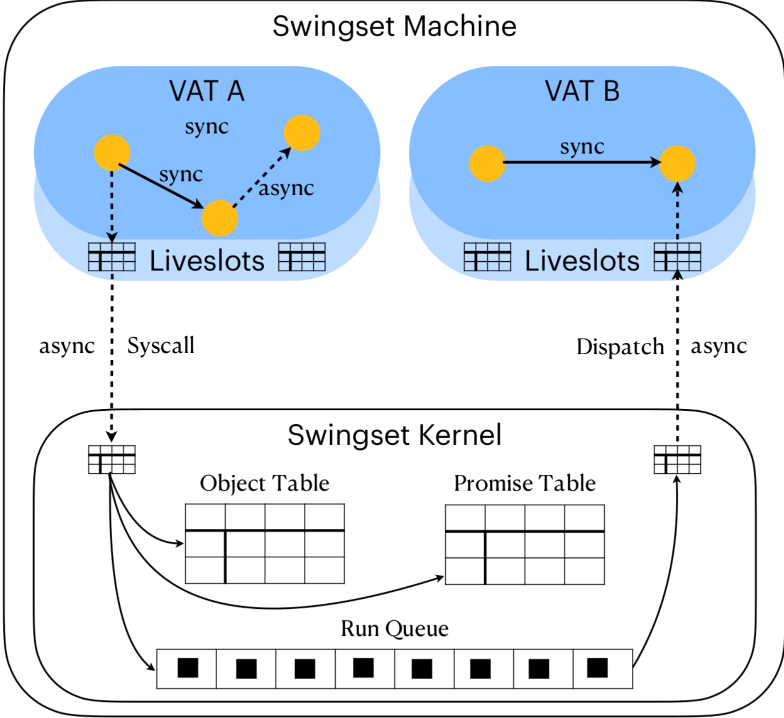
In April 2021, Agoric engaged Informal Systems to conduct a formal methods assessment of the documentation and the current state of the implementation of the Agoric SwingSet platform kernel and vat interactions.

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 along 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 that 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.

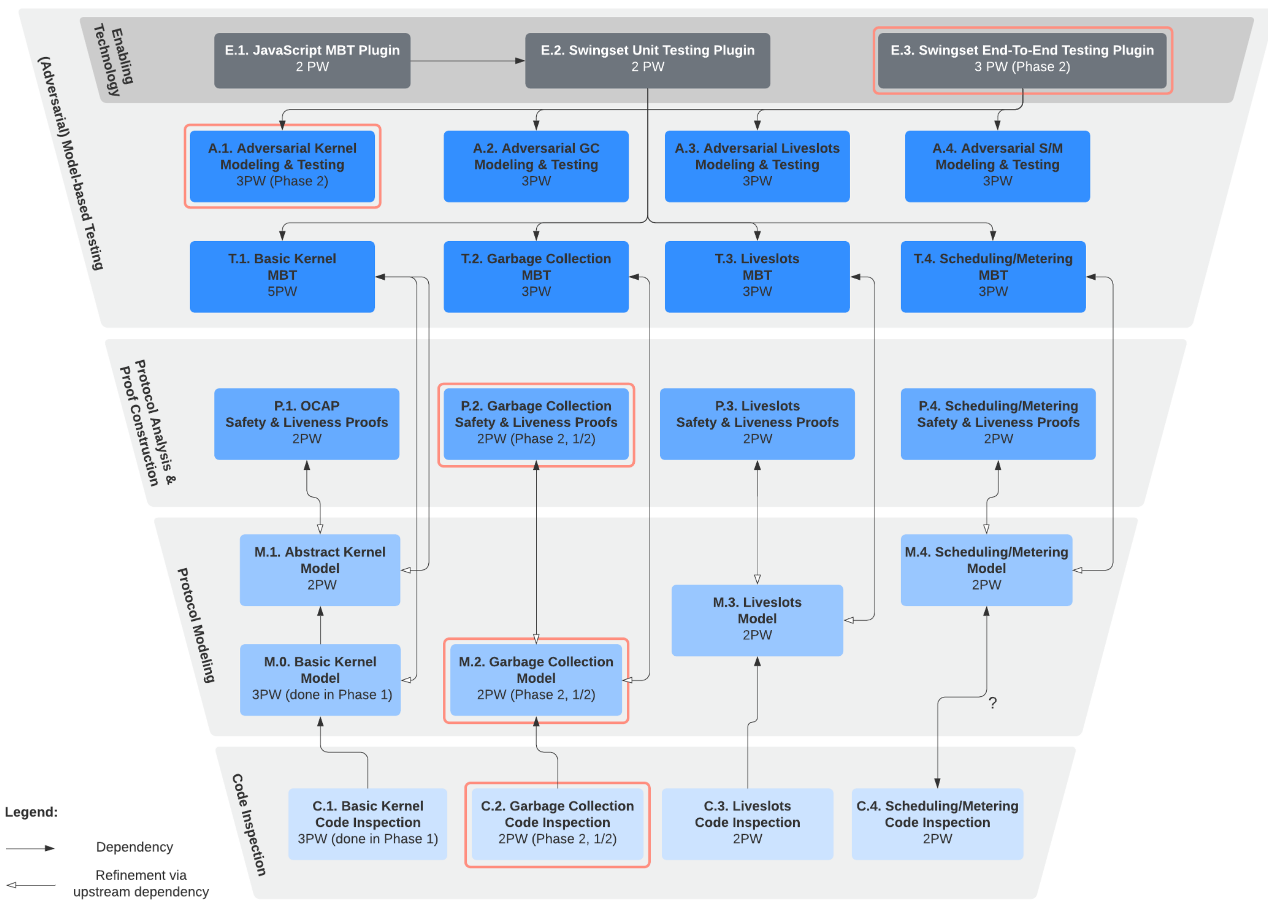
The figure below shows a diagram of the architecture of the SwingSet kernel and two interacting vats. Each vat is a unit of synchrony and synchronous communication only occurs inside a single vat. The liveslots layers mediate access to the outside world from userspace code. Every remote object access 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 2 of the Swingset analysis, with Phase 1 results covered in the previous report. The agreed-upon workplan for Phase 2 consisted of the following tasks, as outlined in the diagram:

1. Task E.3. Swingset End-To-End Testing Plugin
2. Task A.1. Adversarial Kernel Modeling & Testing
3. Task C.2. Garbage Collection Code Inspection
4. Task M.2. Garbage Collection Model
5. Task P.2. Garbage Collection Safety & Liveness Proofs



This report covers the above tasks, with the exception that the tasks related to garbage collection protocol have been restricted to the communication between kernel and vats, but not including the virtual object manager or comms vat. The tasks were conducted 22.07.2021 through 28.09.2021 by Informal Systems by the following personnel:

- Andrey Kuprianov: Principal Research Engineer at Informal Systems
- Daniel Tisdall: Research Engineer at Informal Systems

Conducted work

Starting 22nd July, Informal Systems conducted an assessment of the existing documentation and code. Our team started with reviewing the SwingSet package documentation to get an overview of the design principles of the system, and with a review of some critical components of Hardened JavaScript, which are critical for the security of the platform. Some of the effort included reviewing previously studied material, as we had a new team member join who was not familiar with the system from the first stage of the assessment.

After developing an understanding we created an adversarial testing model and test driver, to exercise the userspace

capabilities of inter-communicating vats. The 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 protocol, restricted to the parts that relate to the kernel, and vats, but not including the virtual object manager or comms vat. We created an abstract TLA+ model of the protocol and model checked it. We found no errors, and the model checking search completely exhausted the state space.

Informal Systems created issues in the agoric-sdk Github repository for a number of issues that we found over the course of the assessment. The issues can be found at the repository.

Timeline

- 22.07.2021: Start of assessment
- 26.07.2021: Bootstrap discussion: update from the Agoric side; setting the goals, priorities & timeline
- 01.09.2021: Intermediate meeting: preliminary demo of userspace adversarial testing & discussion on modeling the garbage collection protocol
- 23.09.2021: Final demo of userspace MBT & demo/discussion of GC modeling and verification
- 28.09.2021: End of assessment
- 21.10.2021: First draft of this report
- 02.11.2021: Correction suggestions received from Agoric
- 09.11.2021: Submission of the final version of this report

Conclusions

Overall we found the code to be organized, reasonably well documented, and faithful to the specification. Despite the general high quality of the implementation work in terms of correctness, we found several significant issues regarding code quality, code organization, and possible divergence from the specification and documentation. These are detailed in the relevant findings. The main source of issues seems to be the high complexity of the code in function bodies, and 'class' object definitions. This complexity can be considered a symptom of a greater problem, namely that the code does not use an Object Oriented Programming style, and is instead written in a procedural style. Indeed, much of the code does not exhibit clear separation of concerns, and has side effects which make it difficult to reason about. Code comprehension is difficult, and is likely to hinder developers from effectively working on the codebase as the project matures. The difficulty in understanding is not decreased by the use of JavaScript as a language, and the extensive use of creating objects dynamically via various factory functions, which make it hard to follow the thread of execution between method calls. Departing from the issues relating to code structure, we did *not* find any significant semantic errors in either the garbage collector protocol, or the inter-vat communication implementation. None of the pathways tested or modelled deviated from the expected behavior.

Assessment Dashboard

Target Summary

- **Name**: Agoric SwingSet inter-vat communication in userspace code and garbage collection protocol
- **Type**: Documentation and implementation
- **Platform**: JavaScript
- **Artifacts**:
  - Agoric SDK at commit 774cb6ad30

Engagement Summary

- **Dates**: 22.07.2021 – 28.09.2021
- **Method**: Manual review & formal protocol modelling & protocol verification & adversarial testing
- **Employees Engaged**: 2
- **Time Spent**: 9 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 Object-Capability model, and its implementation
2. The garbage collector protocol

It was determined that Informal Systems would do analysis of 1) and 2) by way of adversarial (model-based) testing, and protocol model checking. In particular, during this phase of the assessment, Informal Systems checked the object capability (OCAP) system by applying adversarial testing to check and generate interactions among vats communicating in the userspace. Additionally, Informal Systems created a TLA+ model of one part of the garbage collector protocol functionality, namely the garbage collection flows in the kernel and the kernels interactions with exporting and importing vats.

Coverage

(MBT) Adversarial testing of inter-vat communication in userspace

(MBT) Overview

Users write code which executes in vats, and vats are able to communicate by sending messages and objects to each other. Informal Systems applied adversarial (model-based) testing to test the code paths that implement the inter-vat communication and Object-Capability model.

Informal Systems created a TLA+ model of a system consisting of a number of vats which non-deterministically create one of three object types and also interact with created objects in several ways. An execution consists of several steps of object creation and interaction. The SwingSet code is tested by converting executions generated by the model checker into a runnable script, and running the script with vat driver code implemented in the system. In a model step vats either create a reference to themselves, or a promise and a resolver function for the created promise, or they perform actions with existing objects.

Vat Reference

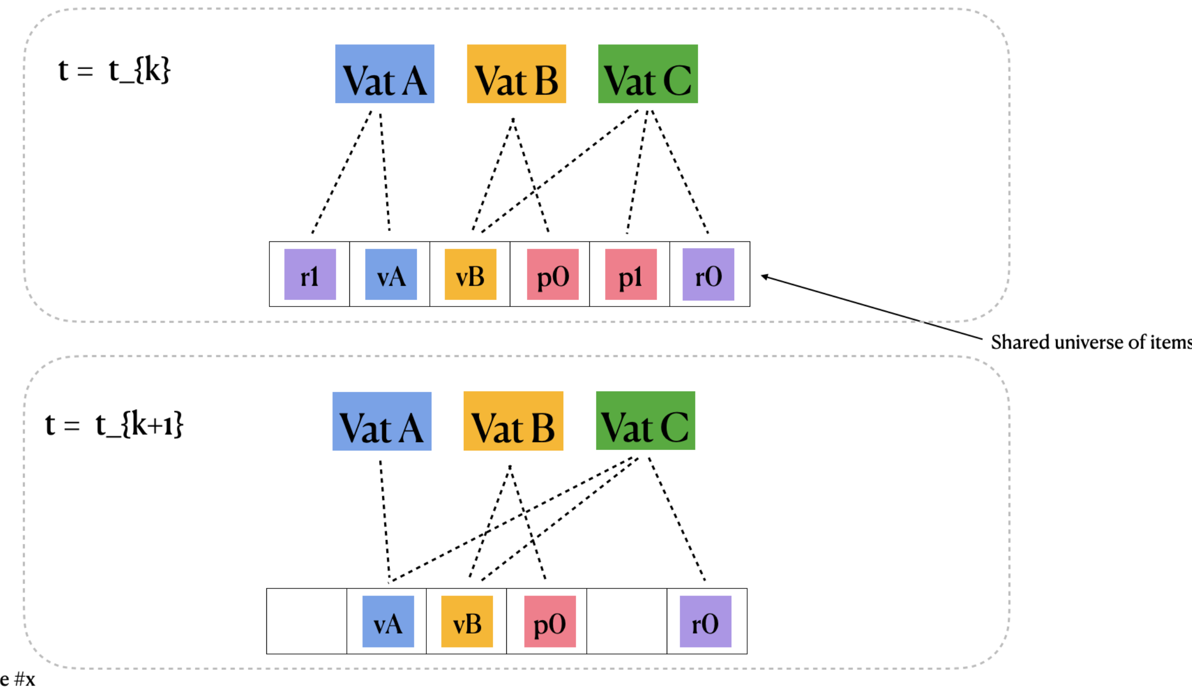
Vats can interact with a vat reference by calling a (remote) `.send` method on the reference, which takes an object as an argument. The `.send` method gives the subject vat the capability to use the argument object in future steps of the execution. Vats can also interact with a vat reference by passing it as an argument to the resolver of a promise. Any vat which had the capability to access the promise will subsequently have access to the resolved-to vat.

Promise and Resolver

Vats can interact with a promise by awaiting the result, or passing it as an argument in a `.send` method call on a vat reference. Vats can interact with a promise resolver function object by either passing it as an argument in a `.send` method call, or executing it (resolving the promise) with a vat reference or promise resolver object as argument.

Illustration

The figure below shows a representation of two subsequent model states. Each vat can access a subset of the items (objects) in the shared universe. Access is represented by dotted lines. Vat references are coloured, promises are pink and resolvers to promises are purple. Promises and their resolvers are also numbered correspondingly.



At time `t_{k}` vat A can access the resolver `r1` for promise `p1` even though it cannot access `p1`. It resolves the promise to `vA`, a reference to itself that it has access to. Vat C has access to `p1` so in the subsequent state at time `t_{k+1}` vat C has access to the item that `p1` resolved to (`vA`).

(MBT) Results

We did not uncover any error in the code by running the executions. We did also intentionally introduce errors (one at a time) into the SwingSet source code to ensure that our test driver was able to find them.

Executions run (all error free)

We generated and ran 1700 tests consisting of a sequence of model steps (max 9). The below table shows three batches of tests run, making 1700 in total. The *Num vats* column contains the number of vats modelled in the execution. The *Universe size* column contains the maximum number of objects (vat references, promises or resolvers) present in the system. The *Pattern* column contains a description of the model behavior that an execution on the corresponding batch matches.

Num executions	Num vats	Universe size	Pattern
1297	2	4	There is at least one *send* step.
303	2	4	Each vat does a step, and some step resolves a promise that another vat has access to.
100	3	5	Each vat does a step, and some step resolves a promise that another vat has access to.

(MBT) Method

There are three components to the adversarial (model-based) testing implemented.

1. The TLA+ model
   The model used for generating executions for testing.
2. Glue code used to convert TLA+ traces into scripts which can be followed by the test driver
   Python and Bash scripts are used to convert TLA+ executions into .json scripts which can be interpreted by the JavaScript code to run the system. The scripts take care of parsing the TLA+ executions, doing conversions, and running the scripted executions by calling the `swingset-runner` executable with a script filename as argument. There is also code to parse the logs generated by the test executions for any error messages.
3. JavaScript code used to run scripts against the SwingSet kernel
   The vat code consists of a bootstrap vat source file, and another vat source file, which is used by all the vats in the execution.

The TLA+ model models the universe of objects which can be interacted with using a single array to store the objects, however in the JavaScript code it is necessary for each vat to maintain a local Map that stores the objects it has access to. Therefore the conversion from a TLA+ execution to a runnable script must perform some logic which tracks the universe of objects and assigns object ids to each of them, which running vat code can use for lookups. The logic of this conversion can be found in the `states_to_driver_script.py` artifact.

The stdout stream of a running test execution is captured and the contents is scanned for keywords in the set {"err", "warning", "panic", "kernel"}. Additionally, it is checked that a sentinel string is present "Bootstrap Done.". The presence of a keyword or the absence of the sentinel string indicates a problem. The `summarize_results.py` artifact can be run to check each stdout stream for a problem.

The entry point for running a test execution against the system is found at `packages/swingset-runner-alt/bin/runner-alt`. The `swingset-runner-alt` package is a modified version of the `swingset-runner` package. It was not possible to run the system using the original `swingset-runner` package (see issue), so slight modifications were made to parts of the code that setup the system in order to work around the problem. None of the functional parts of the code were changed (for example the kernel or liveslots code).

In order to confirm the working of the adversarial testing driver we introduced errors into the SwingSet source code. The adversarial testing executions were able to detect the presence of all the introduced errors. Descriptions of each of the errors introduced can be found in `intentionally_introduced_errors.md`.

(GC) Model-checked TLA+ model of garbage collector protocol

(GC) Overview

The garbage collector protocol consists of three major psuedo-components which may be reasoned about in a modular manner. These are 1) the protocol between the kernel and liveslots, 2) the protocol between liveslots and the JavaScript engine garbage collector and 3) the comms vat protocol.

We created a TLA+ model of 1) in this phase of the assessment. We modelled the protocol for the lifetime of a single object exported by one vat and imported by one vat following the liveslots protocol rules, and one vat ignoring the liveslots protocol rules. In this manner we check the protocol in the presence of malicious behavior. The model models the execution flow of kernel and vat (liveslots) syscalls and dispatch calls, with the effects of each relevant syscall and dispatch being applied to the model state.

The syscalls modelled are

1. dropImport
2. retireImport
3. retireExport

and the dispatch calls modelled are

1. dropExport
2. retireImport

3. retireExport

The initial model state is the state in which an object has been exported by the exporting vat, and imported by both importing vats. The model execution follows a sequence of steps involved in freeing the object from each of the relevant kernel and liveslots data structures.

(GC) Results

The TLC model checker explored the entire state space consisting of 311 distinct states in under 1 second of running time. The checker *did not* find any violation of any of the invariant properties that we specified for the model. Moreover, the checker *did* find an example execution of the 'happy path', that is, an example of a correct protocol execution, resulting in the safe freeing of an object at the end of its lifetime.

Model vat states: Known and Unknown

In the model a vat is in a known state if it has not yet dropped or retired the object whose lifetime is being modelled. A vat state transitions from Known to Unknown if a dispatch message is sent to the vat, or if the vat spontaneously makes a drop or retire syscall. In the case that the state transitions because of a drop or retire syscall, the transition captures a sequence of events in the running system, namely the transitions from the REACHABLE state to the FINALIZED state and ultimately the UNKNOWN state.

Invariant 1: Importer cannot reach object while exporter cannot.

We checked for a state in which an exporter has freed its strong reference while an importing vat can still reach the object. This can occur if either the `exportedRemotable` variable in liveslots has been freed, or the model variable `exporter_state` is Unknown, while the `importer_state` variable is Known.

No state satisfied this predicate, so the invariant holds.

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

Invariant 2: The kernel object is never freed while a vat can reach the object.

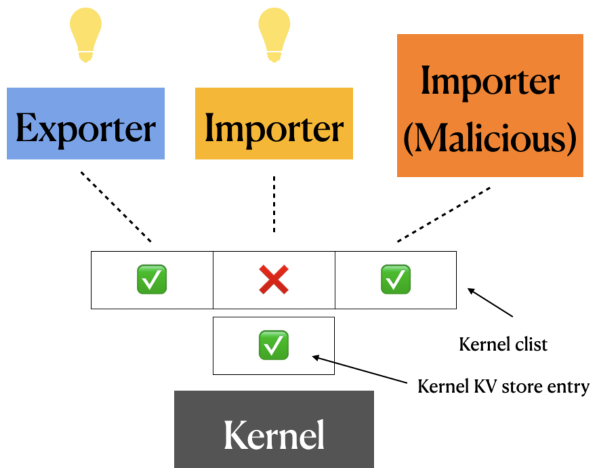
We checked for a state in which an exporting or importing vat can reach the object, while the kernel cannot. We also check for the case that the kernel reference count `reachableCnt` is not 0 while the object has been freed.

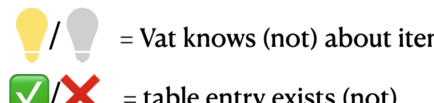
No state satisfied this predicate, so the invariant holds.

```
KernelObjUnsafeFree ==
    /\ ~kernel_obj
    /\ \/ importer_state = Known
       \/ exporter_state = Known
       \/ 0 < reachableCnt
```

Invariant 3: The kernel clist data structures are never missing translation data for a vat while the vat can reach the object.

We checked for a state in which the kernel clist data structure entry for a vat has been freed while the vat can access the object. Figure 4 shows a representation of the kernel data structures and vat states in the model. The state illustrated is one that satisfies the predicate because the importer state is known (the lightbulb is lit) while the kernel clist entry not does exist.





No state satisfied this predicate, so the invariant holds.

```
ClistUnsafeFree ==
    \/ /\ ~importer_clist
       /\ importer_state = Known
    \/ /\ ~exporter_clist
       /\ exporter_state = Known
```

Desired outcome: some sequence of actions results in the correct completion of an object lifetime

In addition to the checked invariants, we also checked that some execution exists that implies liveness of the garbage collector. We defined a predicate that is true in a state in which all kernel data structures and liveslots data structures have been freed.

It *was* possible to find an execution with a state that satisfied the predicate, implying that some execution exists in which an object lifetime completes (is eventually freed).

```
DesiredOutcome ==
    /\ importer_state = Unknown
    /\ exporter_state = Unknown
    /\ kernel_obj = FALSE
    /\ importer_clist = FALSE
    /\ exporter_clist = FALSE
    /\ venomous_clist = FALSE
```

(GC) Method

The logic encoded in the model is an interpretation derived from reading the relevant source files in the SwingSet package. Informal Systems read the relevant code and constructed a mental model of the protocol, as it occurs in the code. The notes for this mental model can be found in the `summary_notes_code.md` artifact. The TLA+ model was then constructed using the mental model, and its semantics closely match the notes.

Findings

ID	Title	Category	Severity	Issue
IF-AGORIC2-01	Unprincipled use of OOP concepts has likely created technical debt	Code structure	Informational	#4040
IF-AGORIC2-02	Make the code in gc-actions.js easier to understand	Code structure	Informational	#4041
IF-AGORIC2-03	Do not overload isReachableFlag	Code structure	Informational	#4042
IF-AGORIC2-04	retireImports dispatch does not have any effect on liveslots	Code structure	Informational	#4043
IF-AGORIC2-05	Check the correctness of the retireImports syscall code with respect to timing	Code structure	Informational	#4044

Finding Categories

- *Protocol*: A flaw or problem in an abstract protocol or algorithm.
- *Implementation*: A problem with the source code. For example a bug, a divergence from a specification, or a poor choice of data structure.
- *Code structure*: A problem that impacts the extent to which the project is maintainable and understandable by developers in the long term.
- *Documentation*: A lack of documentation, or insufficient clarity, accuracy, understandability or readability of existing documentation.

Finding Severities

- *Informational*: The issue does not pose an immediate risk (it is subjective in nature). Findings with informational severity are typically suggestions around best practices or readability.
- *Low*: The issue is objective in nature, but the security risk is relatively small or does not represent a security vulnerability.
- *Medium*: The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited.
- *High*: The issue is an exploitable security vulnerability.

In the effort to improve the usefulness of this report, we do not include here descriptions for issues in the *Documentation* category; all issues can be found in the Agoric-SDK repository.

IF-AGORIC2-01: Unprincipled use of OOP concepts has likely created technical debt

Category	Code structure
Severity	Informational
Issue	agoric-sdk#4040

Involved artifacts

- agoric-sdk

Description

We found that a majority of the code in the SwingSet package does not adhere to Object Oriented programming principles. We have found that much of the code is written in a procedural style. The result of the procedural style means that the code suffers from being difficult to understand, and surely difficult to safely modify and extend (especially if a new developer were to join the project).

While other findings (2-2, 2-3) hint at the problem, we give here a more detailed write up of the problem, examining *one* function. Consider the `deliverToTarget` method of `kernel.js`

```
async function deliverToTarget(target, msg) {
    insistMessage(msg);
    /** @type { PolicyInput } */
    let policyInput = ['none'];
    const { type } = parseKernelSlot(target);
    if (type === 'object') {
      const vatID = kernelKeeper.ownerOfKernelObject(target);
      if (vatID) {
        policyInput = await deliverToVat(vatID, target, msg);
      } else if (msg.result) {
        resolveToError(msg.result, VAT_TERMINATION_ERROR);
      }
    } else if (type === 'promise') {
      const kp = kernelKeeper.getKernelPromise(target);
      if (kp.state === 'redirected') {
        // await deliverToTarget(kp.redirectTarget, msg); // probably correct
        // TODO unimplemented
        throw new Error('not implemented yet');
      } else if (kp.state === 'fulfilled') {
        const presence = extractPresenceIfPresent(kp.data);
        if (presence) {
          policyInput = await deliverToTarget(presence, msg);
        } else if (msg.result) {
          const s = `data is not callable, has no method ${msg.method}`;
          // TODO: maybe replicate whatever happens with {}.foo() or 3.foo()
          // etc: "TypeError: {}.foo is not a function"
          resolveToError(msg.result, makeError(s));
        }
```

```
        // else { todo: maybe log error? }
      } else if (kp.state === 'rejected') {
        // TODO would it be simpler to redirect msg.kpid to kp?
        if (msg.result) {
          resolveToError(msg.result, kp.data);
        }
      } else if (kp.state === 'unresolved') {
        if (!kp.decider) {
          kernelKeeper.addMessageToPromiseQueue(target, msg);
        } else {
          insistVatID(kp.decider);
          // eslint-disable-next-line no-use-before-define
          const deciderVat = vatWarehouse.lookup(kp.decider);
          if (deciderVat) {
            if (deciderVat.enablePipelining) {
              policyInput = await deliverToVat(kp.decider, target, msg);
            } else {
              kernelKeeper.addMessageToPromiseQueue(target, msg);
            }
          } else if (msg.result) {
            resolveToError(msg.result, VAT_TERMINATION_ERROR);
          }
        }
      } else {
        assert.fail(X`unknown kernelPromise state '${kp.state}'`);
      }
    } else {
      assert.fail(X`unable to send() to slot.type ${type}`);
    }
    return harden(policyInput);
  }
```

It's clear that the function is highly complex, with many nested conditional statements. The burden placed on the developer working with the code is high. It should be noted that many of the functions called within it (recursively) are similarly complex. In fact, in order to understand the result of this function for given arguments, the reader will likely have to mentally (or physically) inline all the function calls contained within (recursively). Very few behaviors can be deduced from function names alone (due to either non-descriptive naming or the presence of many conditional statements).

The execution pathways in the kernel source code interleave logic for many different arguments types, and interleave updates to many different systems. For example consider the possible state space of argument types in `deliverToTarget`. Roughly speaking the space is a subset of {object, promise} X {redirected, fullfilled, . . . } X {import, export} X . . . , with writes/updates made to many different 'systems' including the refcount system, the logs, message queues of various objects, metering, transcript, ect. The complexity cannot be understated.

We propose that a codebase using principled object oriented design could implement the same functionality with a fraction of the complexity.

Recommendation

Put briefly, we recommend a gradual shift to using a more Object Oriented approach in the codebase, trying to refactor existing code where possible, and being strict in using OOP in new code.

For an example of how OOP principles could lead to a cleaner codebase, consider a design in which an Export class is used in the kernel to track any object that has been exported from one vat to another, and consider also its garbage collection lifetime and how it might be implemented. The kernel might process Syscalls similarly to how it does now, creating a set of objects implementing an Event API for each Syscall made. Some events might be

created which match a particular Export object. The events could be processable by the Export object, which could mutate itself, and reactively make method calls on a Garbage Collector Lifetime class instance that it has a member reference to. The Garbage Collector Lifetime instance object might mutate itself, and reactively emit events which can then be processed by a Kernel instance, which could reactively mutate itself and emit new Dispatch events, ect. The advantages of such a design are separation of concerns and encapsulation.

Additionally, strive where possible to separate updates to state, from the computation of what the update should be. Generate update objects, and process (consume) the update objects, to the greatest extent possible, all at one time or in siloed temporal intervals. The previous paragraph mentions mutation for simplicity, but alternative styles of programming exist, such as using persistent data structures.

The hints given here are just hints, and any OOP programming at the scale required for the SwingSet package would require a substantial investment in design time.

Concluding: a full OOP rework of the SwingSet package would be an expensive and idealistic undertaking. Appreciating this, we recommend that alternative ways of structuring the codebase be *at least considered* going into the future, as we doubt the sustainability of continuing with the current pattern of writing more and more complex procedural code.

IF-AGORIC2-02: Make the code in gc-actions.js easier to understand

Category	Code structure
Severity	Informational
Issue	agoric-sdk#4041

Involved artifacts

- agoric-sdk

Description

The function `processNextGCAction` in `gc-actions` demonstrates less than ideal code quality, which can be improved. The code demonstrates some instances of problems that occur in many places in the SwingSet codebase (see finding 2-1).

Combining writes with reads unnecessarily (side effects)

Consider the following code

```
 function filterActions(vatID, groupedActions) {
   const vatKeeper = kernelKeeper.provideVatKeeper(vatID);
   const krefs = [];
   for (const action of groupedActions) {
     const { type, kref } = parseAction(action);
     if (filterAction(vatKeeper, action, type, kref)) {
       krefs.push(kref);
     }
     allActionsSet.delete(action);
   }
   return krefs;
 }

// ...

 const vatIDs = Array.from(grouped.keys());
 vatIDs.sort();
 for (const vatID of vatIDs) {
   const forVat = grouped.get(vatID);
   // find the highest-priority type of work to do within this vat
   for (const type of typePriority) {
     if (forVat.has(type)) {
       const actions = forVat.get(type);
       const krefs = filterActions(vatID, actions);
       if (krefs.length) {
         // at last, we act
         krefs.sort();
         // remove the work we're about to do from the durable set
```

```
          kernelKeeper.setGCActions(allActionsSet);
          return harden({ type: `${type}s`, vatID, krefs });
        }
      }
    }
  }
```

In `filterActions` the line `allActionsSet.delete(action);` is a side effect which deletes entries from a variable in the outer scope. This is bad because because

1. all the actions in `groupedActions` are deleted, so the deletion step can easily be performed before or after the `const krefs = filterActions(vatID, actions);` line below.
2. In javascript `filter` functions usually create a new container containing only predicate passing values of the called on container.
3. Side effects are code-smell, they increase the cognitive load on developers, slowing development time and often leading to errors.

Confusing naming discipline

As mentioned in bullet 2) above, the filterActions function name does not adhere to JavaScript convention as it performs writes. Additionally, there is a similarly named function

```
  function filterAction(vatKeeper, action, type, kref) {
    // ...
  }
```

which has completely different behavior. The `filterAction` function is a predicate, whereas the `filterActions` function returns a container, and performs additional writes. It may be better to name `filterAction` something like `isValidAction`.

Recommendation

Rework the code in `gc-actions.js` so that there is a clear separation between read steps, logic steps, and write steps. Additionally, strive to use function names that are similar only if the semantics and return types are similar, especially if they are located close to each other.

The code could be structured similarly to the below snippet. The snippet is intended to be an example of a cheap reworking of the existing code. A larger scale redesign of the source code using OOP principles, as per finding 2-1, would improve over what is given here substantially.

```
export function processNextGCAction(kernelKeeper) {

    function isValidAction(vatKeeper, type, kref) {
        //...
    }

    function filterActions(vatID, groupedActions) {
      const vatKeeper = kernelKeeper.provideVatKeeper(vatID);
      const krefs = [];
      for (const action of groupedActions) {
        const { type, kref } = parseAction(action);
        if (isValidAction(vatKeeper, type, kref)) {
          krefs.push(kref);
        }
      }
      return krefs;
    }
```

```
  const allActionsSet = kernelKeeper.getGCActions();

  const grouped = new Map(); // grouped.get(vatID).get(type) = krefs to process
  for (const action of allActionsSet) {
    const { vatID, type } = parseAction(action);
    if (!grouped.has(vatID)) {
      grouped.set(vatID, new Map());
    }
    const forVat = grouped.get(vatID);
    if (!forVat.has(type)) {
      forVat.set(type, []);
    }
    forVat.get(type).push(action);
  }

  function actionToReturnAndActionsToDelete(){
      let actionsToDelete = new Set()
      const vatIDs = Array.from(grouped.keys());
      vatIDs.sort();
      for (const vatID of vatIDs) {
        const forVat = grouped.get(vatID);
        // find the highest-priority type of work to do within this vat
        for (const type of typePriority) {
          if (forVat.has(type)) {
            const actions = forVat.get(type);
            actionsToDelete.addAll(actions)
            const krefs = filterActions(vatID, actions);
            if (krefs.length) {
              // at last, we act
              krefs.sort();
              // remove the work we're about to do from the durable set
              return [harden({ type: `${type}s`, vatID, krefs }), actionsToDelete]
            }
          }
        }
      }
      return [undefined, actionsToDelete]
  }

  const [actionToReturn, actionsToDelete] = actionToReturnAndActionsToDelete()
  allActionsSet.removeAll(actionsToDelete)
  // remove negated items from the durable set
  kernelKeeper.setGCActions(allActionsSet);
  return actionToReturn;
}
```

IF-AGORIC2-03: Do not overload isReachableFlag

Category	Code structure
Severity	Informational
Issue	agoric-sdk#4042

Involved artifacts

- agoric-sdk

Description

The `isReachableFlag` variable is treated completely differently between exporting and importing vats. The difference is documented and the semantics make sense, but it is not immediately clear that the uses are different without having read the doc.

Recommendation

Use two separate variables. This is easier for the reader, and will likely lead to trickle down improvements in code quality. For instances, consider the following conditional

```
function setReachableFlag(kernelSlot, _tag) {
  // ...

  // increment 'reachable' part of refcount, but only for object imports
  if (!isReachable && type === 'object' && !allocatedByVat) {
    // eslint-disable-next-line prefer-const
    let { reachable, recognizable } = getObjectRefCount(kernelSlot);
    reachable += 1;
    // kdebug(`++ ${kernelSlot} ${tag} ${reachable},${recognizable}`);
    setObjectRefCount(kernelSlot, { reachable, recognizable });
  }
}
```

This kind of check can be removed, or made simpler and easier to understand by choosing to use different variables. This issue is a symptom of a deeper problem relating to underuse of OOP principles (see finding 2-1).

IF-AGORIC2-04: retireImports dispatch does not have any effect on liveslots

Category	Code structure
Severity	Informational
Issue	agoric-sdk#4043

Involved artifacts

- agoric-sdk

Description

The liveslots code that handles the dispatch:

```
function retireImports(vrefs) {
  assert(Array.isArray(vrefs));
  vrefs.map(vref => insistVatType('object', vref));
  vrefs.map(vref => assert(!parseVatSlot(vref).allocatedByVat));
  // console.log(`-- liveslots ignoring retireImports ${vrefs.join(',')}`);
}
```

The code does does implement any feature of the garbage collection protocol. The lack of action taken here indicates some slack in the garbage collector protocol.

Recommendation

Change this code to either take some action in the liveslots, or remove the `retireImports` dispatch call entirely and design a new mechanism by which vats can clear dead weakrefs that are no longer valuable.

IF-AGORIC2-05: Check the correctness of the retireImports syscall code with respect to timing

Category	Implementation
Severity	Informational
Issue	agoric-sdk#4044

Involved artifacts

- agoric-sdk

Description

The intended behavior relating to `syscall.dropImports` is not clear due to a mismatch between the docs and the code, and suspicious code.

Consider the following documentation in garbage-collection.md (line 343)

```
Importing vats perform `syscall.retireImport` when they can neither reach nor recognize an
↪  import. If the Presence was never used in virtualized data or as a weak key, this will
↪  happen at the end of the crank in which the Presence is collected. Otherwise, the vat may
↪  do `syscall.dropImport` now, and `syscall.retireImport` much later.
```

The documentation implies that a `syscall.retireImport` call may be made at a much later (different turn/crank) time than `syscall.dropImport`, but it is unclear that this prescription matches the code. Consider `liveslots.js`:

```
    for (const vref of deadSet) {
      const { virtual, allocatedByVat, type } = parseVatSlot(vref);
      assert(type === 'object', `unprepared to track ${type}`);
      if (virtual) {
        // Representative: send nothing, but perform refcount checking
        // eslint-disable-next-line no-use-before-define
        doMore = doMore || vom.possibleVirtualObjectDeath(vref);
      } else if (allocatedByVat) {
        // Remotable: send retireExport
        exportsToRetire.push(vref);
      } else {
        // Presence: send dropImport unless reachable by VOM
        // eslint-disable-next-line no-lonely-if, no-use-before-define
        if (!vom.isVrefReachable(vref)) {
          importsToDrop.push(vref);
          // eslint-disable-next-line no-use-before-define
          if (!vom.isVrefRecognizable(vref)) {
            importsToRetire.push(vref);
          }
        }
      }
    }
    deadSet.clear();
```

```
  if (importsToDrop.length) {
    importsToDrop.sort();
    syscall.dropImports(importsToDrop);
  }
  if (importsToRetire.length) {
    importsToRetire.sort();
    syscall.retireImports(importsToRetire);
  }
  if (exportsToRetire.length) {
    exportsToRetire.sort();
    syscall.retireExports(exportsToRetire);
  }
```

Notice that the `vref` is pushed to `importsToRetire` only if it is also pushed to `importsToDrop`. Notice also that the `deadSet` is cleared before any syscalls are made. This implies that the vref will never be processed again, unless it is added to the deadSet again, however vrefs are only added to the deadSet in the `finalizeDroppedImport` function

```
function finalizeDroppedImport(vref) {
  const wr = slotToVal.get(vref);
  // The finalizer for a given Presence might run in any state:
  // * COLLECTED: most common. Action: move to FINALIZED
  // * REACHABLE/UNREACHABLE: after re-introduction. Action: ignore
  // * FINALIZED: after re-introduction and subsequent finalizer invocation
  //   (second finalizer executed for the same vref). Action: be idempotent
  // * UNKNOWN: after re-introduction, multiple finalizer invocation,
  //   and post-crank cleanup does dropImports and deletes vref from
  //   deadSet. Action: ignore

  if (wr && !wr.deref()) {
    // we're in the COLLECTED state, or FINALIZED after a re-introduction
    deadSet.add(vref);
    slotToVal.delete(vref);
    // console.log(`-- adding ${vref} to deadSet`);
  }
}
```

and only when the state is Collected or Finalized.

Recommendation

Check the code for correctness and reflect the correct code in the documentation.