Technical Report: Semantics and further Use-Cases and Evaluation of the C-Saw language

Henry Zhu[1], Junyong Zhao[2], Nik Sultana[3]
[1] *University of Illinois Urbana-Champaign*
[2] *University of Arizona*
[3] *Illinois Institute of Technology*

9th March 2023

Abstract

This report provides supplementary technical details to the conference paper that introduced C-Saw [6], a language for expressing software architecture patterns. This report provides additional examples of using C-Saw, supplementary evaluation details, and it defines the formal semantics of the language.

Contents

1 Further description of the C-Saw DSL **2**

2 More architecture examples in C-Saw **6**
2.1 Parallel sharding 6
2.2 Caching 7
2.3 Fail-over 7
2.4 Watched fail-over 8

3 Semantics **19**
3.1 Event structures 20
3.2 Labels 21
3.2.1 Graphical notation 21
3.3 Supporting definitions 22
3.4 Program semantics 24
3.5 DSL statement semantics 25
3.6 Example 27
3.7 Topology 28

4 Further evaluation **30**

1 Further description of the C-Saw DSL

This section builds on Section 6 of the C-Saw paper [6].

Start and stop. The 'start' and 'stop' primitives control whether an instance is running or not. Once started, an instance cannot be started again until it is stopped, otherwise the primitive would fail. Similarly, a stopped instance cannot be stopped. At least one instance is started in 'main', which can use and propagate parameters when starting instances. When an instance is started, its junctions are started concurrently in an arbitrary order.

Instance and junction references. Junction names are always fully-qualified. The '::' operator is used to form junction names. The special names 'me::junction' and 'me::instance' refer to the containing junction and the instance of an expression, respectively.

Distributed Key-Value (KV) table. Each junction has a KV table that can be synchronized between junctions. Junctions can push, but cannot pull: that is, they may *write* to both their and other junctions' tables, but may only *read* their local table. C-Saw adapts the tuple-space idea [1] but restricts readability to junctions.

Junction state. An executing junction can receive remote updates to its table through write, assert and retract. These updates are not made to the table until the junction is next scheduled for execution. 'write' can only be used on data that has been generated by 'save'—i.e., so-called *named data*. We can 'restore' any values except for read-only ones, such as parameters (described further below).

A junction can discard parallel KV updates through the 'keep' primitive. This primitive is idempotent and can be applied to propositions and data. Local updates to the table, performed using save, assert and retract, are visible immediately to the junction and overwrite pending updates from other junctions. There can be a race condition when updating and reading these values unless the logic is carefully structured. To help with this structuring and to selectively permit external updates while the junction is running, the 'wait' primitive blocks execution until a formula is satisfied, and allows the junction's table to reflect changes to propositions in that formula and a set of data keys. If the formula is immediately true, then the statement returns immediately. The 'otherwise' primitive can be used to impose a time limit on the blocking statement.

Names. The following entities are named: propositions, data, instances and junctions, and variables—these can consist of parameters and for-bound symbols, and may range over sets and set elements; the latter can be propositions, data, and instances and junctions. Names can be indexed, as described next.

Parameters, data types, indexing. Definitions can accept parameters of different types of data. Propositions, named data, sets, and host-language data are all legal parameters. Examples can be seen in §5. A definition must be given the right number of parameters in the right order for the program to be well-formed. main can take an arbitrary number of parameters. These are usually distributed among the instances that it starts.

In this paper, parameter variables are indicated as $\overline{p}$ to distinguish them from other types of names, such as for-bound symbols $\widetilde{p}$. Both definition parameters and 'for' variables are constant variables: that is, they can be read but not assigned to.

Sets have a fixed size at compile time and can contain any kind of data but *not* other sets. For example, sets can contain references to instances—an example is given in §2.3.

Sets may be provided literally, as seen at ❶ in Fig. 5, or declared using the **set** syntax and provided as a compile-time parameter, or derived from another set. A set may be derived from another set in two ways:

1. As a mapping, as done for the set Backend at ❶ in Fig.6,
2. Using the **subset** declaration syntax to allow external code, through $\lfloor \ldots \rceil$ syntax, to populate a set as a subset of a previously-defined set.

All sets and subsets are necessarily finite, and it is always possible to iterate over them.

Sets can be *indexed* using other data except for sets. Indices can be formed in two ways:

1. Using for-bound symbol, such as in InitBackend$[\widetilde{b}]$ and Backend$[\widetilde{tgt}]$ in Fig. 6.
2. Using the **idx** declaration syntax. This allows external code in the host language—through $\lfloor \ldots \rceil$ syntax—to provide a choice function over a given set or subset.

Indices and sets, including subsets, can be passed as definition parameters. This can be seen for sets with the $\overline{backends}$ parameter to $\tau_f :: b$ in Fig. 6. An example of indices being passed by parametsr is shown in §2.3.

Neither indices nor sets should be serialized or transmitted between junctions, because they might not have valid interpretations at the receiving end.

A contract with the host language requires that the externally-definable subsets and indices must have valid values relative to the sets to which they are defined.

Functions and brackets. Functions are templates that are expanded at compile time. They are similar to named equivalents of the $\langle E \rangle$ syntax that gathers a composition of expressions in a common scope. This is not a scope for definitions, but one for *fate* [3]: that is, if part of the expression fails then the whole expression fails unless there is some suitable handling logic. $\langle\!| E |\!\rangle$ brackets have

an added behavior: upon failure, a roll-back of state (the KV table) is carried out, restoring it to the point before the brackets were entered. The $\lfloor \ldots \rceil$ syntax is not allowed in $\langle\!|E|\!\rangle$ since roll-back is undefined for it.

More on branching. 'skip' is a no-op, and 'return' leaves a fate scope. Both operations can only succeed. Since functions are expanded templates, 'return' in a function will leave the junction, not just return from the function to the junction. 'case' is a key control-flow syntax used in this language. Each arm of a case-expression terminates in one of a fixed number of ways. 'break' leaves the case expression, 'next' retries the case, but can only match *after* the arm that succeeded, and 'reconsider' was described in §6 of the C-Saw paper [6]. There are additional validity constraints on case constructs: they cannot be empty or only contain an 'otherwise' branch, nor can 'next' be used immediately before 'otherwise'.

Recursion. Recursion is restricted in this language. It can take place through template-based recursion on expressions, formulas or declarations—these are described further below. Bounded recursion can also occur through 'reconsider' which retries a case-expression, or 'retry' which retries a junction.

Template-based Recursion: Expressions/Formulas. The sugaring 'for $\widetilde{n} \in \vec{N}^m$ *op* $I[\widetilde{n}]$', where $I[n]$ is either E or F and possibly has n free, expands into

$$I[N_1]\ op\ \ldots\ op\ I[N_m]$$

where $op \in \{\vee, \wedge, ;, +, \|, \mathsf{otherwise}[t]\}$.

There are no other constraints on recursion. For example, operator application may be nested—the example

$$\mathsf{for}\ \widetilde{p} \in \{E_1, E_2, E_3\}\ \ \mathsf{otherwise}[t]\ E[\widetilde{p}]$$

becomes:

$$E[E_1]\ \mathsf{otherwise}[t]\ \langle E[E_2]\ \mathsf{otherwise}[t]\ E[E_2]\rangle$$

(Note that operators associate to the right.)
Another example showing the loop's unwinding:

$$\mathsf{for}\ \widetilde{p} \in \{E_1, E_2, E_3\}\ ;\ E[\widetilde{p}]$$

becomes:

$$E[E_1];\ \langle E[E_2];\ E[E_2]\rangle$$

Using 'break' we can exit the loop early.
When 'for' iterates over a singleton set, the loop evaluates only to one instantiation:

$$\mathsf{for}\ \widetilde{n} \in \{E_1\}\ \ op\ E[n]\ =\ E[E_1]$$

When the set is empty:

$$\begin{array}{l}\mathsf{for}\ \widetilde{p} \in \{\}\ \vee\ E[\widetilde{p}]\ =\ \mathsf{false} \\ \mathsf{for}\ \widetilde{p} \in \{\}\ \wedge\ E[\widetilde{p}]\ =\ \neg\mathsf{false}\end{array}$$

And for other operators,

$$\mathsf{for}\ \widetilde{p} \in \{\}\ \ op\ E[\widetilde{p}]\ =\ \mathsf{skip}$$

Template-based Recursion: Declarations. We use 'for' to initialize a set of propositions using **init**, as seen in Fig. 6. In the same example we can see 'for' being used in a 'case' expression. In both cases, the code is inlined at compile time. With the 'case' expression, we can mix different types of recursion, for example:

$$\mathsf{for}\ \widetilde{x} \in \{\ldots\}\ \ (\mathsf{for}\ \widetilde{y} \in \{\ldots\}\ \wedge\ (\mathrm{Foo}[\widetilde{x}] \vee \mathrm{Bar}[\widetilde{y}])) \Rightarrow$$

```
  # 'y' is free here, but 'x' is bound.
```

Communication to self. Junctions cannot send data to themselves—applying 'write' to themselves would be redundant. Junctions may assert or reject propositions, but these are not "communicated" to the junction—the change is made locally. That is, assert [] Prop may be executed in a junction j (assuming that Prop has been properly declared there), but assert [j] Prop may not.

Initialization. Junction definitions use **init** syntax to declare and initialize proposition (**prop**) and data (**data**) variables. The latter are always initialized with the special **undef**. This is not a valid value—trying to write or restore it results in an error. A data variable is given its first valid value using save. **undef** is also used to initialize **subset** and **idx**. **set** must be specified at load time.

Junction safety conditions. **verify** is used to state properties that should hold in different parts of the system, upon those parts being reached in the control flow. We rely on ternary logic—**verify** will return an error if it needs to evaluate $f@P$ and f is not running.

2 More architecture examples in C-Saw

Section 5 of the C-Saw paper [6] provides several example of how the language can be used to capture architectural patterns that support the implementation of important features. Examples of such features were given in Fig. 1 of the paper.

This section builds on the paper to provide additional examples of how to use the DSL to implement important features.

2.1 Parallel sharding

```
InstanceTypes = {τ_Front, τ_Back}
Instances = {Fnt : τ_Front, Bck_1 : τ_Back, ..., Bck_N : τ_Back}
def τ_Front :: (t̄) ◀
  | init prop ¬Work
  | init data n
  | set Backs # Assigned to {Bck_1, ..., Bck_N} ❶
  | for t̃gt ∈ Backs  init prop ¬ActiveBackend[t̃gt] ❷
  | subset tgt of Backs ❸
  | init prop ¬HaveAtLeastOne
  ⌊Choose();⌉{tgt}; save(..., n);
  retract [] HaveAtLeastOne;
  for b̃ ∈ tgt + ❹
    if ActiveBackend[b̃] then
      ⟪ write(n, b̃); assert [b̃] Work; wait [] ¬Work; ❺
        assert [] HaveAtLeastOne; ❻
      ⟫ otherwise[t̄] retract [] ActiveBackend[b̃];
  # Complain if not one backend is viable.
  if ¬HaveAtLeastOne complain();
```

Figure 1: Snippet of N-ary sharding to a *set* of back-ends. The syntax is explained in §2.1.

The code in Fig. 5 of the paper is limited to using a single back-end at a time. This can be improved to use all the back-ends in parallel. One way of doing this involves making Work into a set indexed by tgt, and changing the penultimate line of Fig. 5 to the following:

$$\langle \text{wait } [] \neg\text{Work[tgt]; write}(n, \text{ tgt}); \text{ assert [tgt] Work[tgt]} \rangle$$

Extending this idea further, Fig. 1 shows how the sharding logic can be extended to *sets* of back-end targets. It restructures the architecture to achieve higher availability. Similar architectures could optimize for throughput and latency through load-balancing. In Fig. 1 we see ❶ **set** syntax used to declare a set defined at compile-time, ❷ a derived set called ActiveBackend to track which

back-ends are usable, ❸ **subset** syntax used to declare a runtime-defined subset of an existing set (note that a different kind of "tgt" is being used here than that in ❷), ❹ iteration through a set in parallel (i.e., using the '+' operator), ❺ the same core line from the paper, ❻ use of a proposition to determine if no viable back-ends exist, to alert the operator that the computation cannot terminate successfully.

2.2 Caching

Recall that use-case ⑤ from in Fig. 1 of the C-Saw paper described how caching can be used to avoid repeating expensive or time-consuming operations. This section describes an implementation of an inline cache that memoizes function calls. Not all functions are amenable to memoization—functions need to be pure. For amenable functions, the cache reduces the response time for clients, and reduces the pressure on the resources needed to compute a function. If the architecture separates the part of the system where the function is computed from the rest of the system, then the cache also reduces pressure on the communication resources between the two parts of the system.

The features of the cache, such as its sizes and eviction strategy, are orthogonal to the architecture, and are therefore outside of the DSL's scope. They are expressed and implemented in the host language or provided by linked libraries.

The implementation described in this section interfaces with external functions (in the host language) that classify the request's type. This classification determines whether the cache should be consulted. For cacheable operations, the implementation performs a cache look-up, calls the requested function, and caches the result.

Fig. 2 shows the cache's implementation. Note that τ_{Fun} is closely based on τ_{Auditing} in Fig. 4 from the C-Saw paper. $\lfloor F \rceil$ implements the function to be computed (and whose results can be memoized).

This code uses two data objects: n and m. The state held in junctions' KV-tables and the state held by the host language interact in the following ways:

- n is affected by the context at entry into the junction, and it serializes components that are needed in the remainder of the computation.
- $\lfloor$CheckCacheable$\rceil$ affects Cacheable, which is made explicit by the syntax: $\lfloor$CheckCacheable$\rceil${Cacheable }.
- $\lfloor$LookupCache$\rceil$ affects Cached.
- $\lfloor F \rceil$ affects m, which is used in generating the response.

2.3 Fail-over

A fail-over architecture can be implemented in various ways that provide different trade-offs between availability and overhead. Different implementations

can also differ subtly in their tolerance of different kinds of faults that might arise—such as short losses of synchronization between parts of the system.

This section describes a full implementation in C-Saw that supports fault-tolerance and multiple fail-over stages.

In this architecture, we typify the application logic into a single-instance *front-end* and at least two instances of *back-end*. Back-ends provide redundancy: as long as one back-end works then the system can continue to function. This fail-over design handles a subsystem restarting or reappearing after a transient network outage. The entire system is parametrized by timeouts to discover faults early.

The architecture's logic is not tightly coupled to application logic, and in our prototype the same logic is applied to both Redis and Suricata. Fig. 3 sketches the two instance types and their junctions. The front-end's junctions face clients ($\tau_f :: c$) or back-ends ($\tau_f :: b$). Code for the latter is provided in Fig. 6 which shows how that junction behaves during the Starting phase when contact is made between back-ends and the front-end, and the subsequent phase where client requests are handled. The logic of the architecture is summarized in Fig. 4 for the back-end and Fig. 5 for the front-end.

The implementation described in this section provides an implicit fail-over between warm replicas of back-end instances. While adequate for the design goal, it can be made **(i)** less conservative, and lower latency, by not requiring all the back-ends to respond before returning a response to the client—a single back-end responding would be sufficient; **(ii)** use less network overhead by only having a *single* back-end return a pre-response; **(iii)** scale better than the current linear scaling overhead when additional back-ends are added by structuring sets of back-ends to make the cost logarithmic. To show another point in the design space, an alternative design featuring a watchdog instance is given in §2.4.

2.4 Watched fail-over

One of the take-aways of this research was how the same architectural concept can be implemented in different ways using C-Saw, leading to different architectural features. This section presents an alternative architecture to the fail-over feature described in §2.3 of the C-Saw paper.

The architecture in this example supports two back-ends, o and s, where o is preferred to s, and s is used when o is unavailable. This design also features a watchdog that arbitrates back-end liveness. The front-end focuses on engaging with only one of the two back-ends—unlike the other design which involved engaging with all backends.

The system starts by picking a back-end on which to focus. It then traverses states depending on faults that can arise. The system can continue to function unless both back-ends become unresponsive, or unless the single synchronized back-end becomes unresponsive. The high-level state diagram for the design front-end is shown in Fig. 10. That diagram reuses the notation introduced in Fig. 4 of the C-Saw paper, showing the transitions between states of the

system. Transitions are denoted by arrows indicating whether the transition is made externally (via scheduling) or internally by the system (through one or more changes in instances or their configuration).

The states are composed of the states of instances: the white circle denotes a front-end, and the two blue circles denote back-ends. Within the circles we find an indication of *their* internal state: 0 means that they are initialized but not synchronized, and m and n are two distinct synchronization points. The black edge between the front-end and one of the back-ends denotes the *focus* of the front-end, i.e., which of the two back-ends is currently picked as being the leader.

In Fig. 10 we see a back-end being chosen for focus upon succesful startup, and the system then transitioning between states depending on whether one or both back-ends become unavailable. The system continues functioning through the orange states, and attempts to recover back into a green state. Should both back-ends become unavailable, the system enters a red state and must be restarted.

```
InstanceTypes = {τ_Cache, τ_Fun}
Instances = {Cache : τ_Cache, Fun : τ_Fun}
def main(t̄) ◀
  start Cache(t̄) + start Fun(t̄)
def complain() ◀ ...
def τ_Cache :: (t̄) ◀
  | init prop ¬Work          | init prop ¬Cacheable
  | init prop ¬Cached        | init prop ¬NewValue
  | init data n              | init data m
  ⌊CheckCacheable⌉{Cacheable}; ❶
  case {
    Cacheable ⇒ ❷
      ⌊LookupCache⌉{Cached}; ❸
      next❹
    ¬Cacheable ∨ (Cacheable ∧ ¬Cached) ⇒ ❺
      save(..., n);
      ⟨write(n, Fun);
       assert [Fun] Work;
       wait [m] ¬Work; restore(m, ...);
       assert [] NewValue;
       ⟩ otherwise[t̄] complain();
      next
    Cacheable ∧ NewValue ⇒ ❻
      ⌊UpdateCache⌉; break
  }
def τ_Fun :: (t̄) ◀
  | init prop ¬Work          | init prop ¬Retried
  | init data n              | init data m
  | guard Work
  restore(n, ...);
  ⌊F⌉;
  retract [] Retried;
  case {
    Work ⇒
      ⟨save(..., m); write(m, Cache);
       retract [Cache] Work⟩ otherwise[t̄]
        if ¬Retried then assert [] Retried;
        else complain();
      reconsider
    otherwise ⇒ skip
  }
```

Figure 2: Adding an application-specific caching layer. This examples builds on Fig. 4 from the C-Saw paper [6], whose τ_{Auditing} we largely reuse here as τ_{Fun}. The main differences from previous examples involve the interfacing with externally-defined functions. The key steps in this junction are: ❶ determine whether a request's response could be cached; ❷ have the DSL code react to changes made by external code—e.g., Cacheable is set by ⌊CheckCacheable⌉; ❸ the "case" statement is redone but will not reconsider this branch; ❹ performs the lookup using ⌊LookupCache⌉; ❺ call the function if the result cannot be cached, or if the cache misses; ❻ update the cache if the result is cacheable.

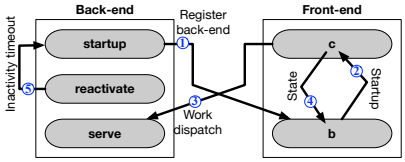


Figure 3: The fail-over architecture described in §2.3 relies on two instance types: back-end and front-end, shown as τ_b and τ_f in our code. They have three and two junctions respectively. This diagram shows important interactions between junctions: ① the junction $\tau_b :: \mathit{startup}$ registers the back-end instance with $\tau_f :: b$, which in turn makes $\tau_f :: c$ aware of which back-ends are available for failing-over; ② once at least a single back-end is available, $\tau_f :: b$ signals $\tau_f :: c$ to start handling requests from clients; ③ client requests are dispatched to back-ends; ④ $\tau_f :: b$ oversees the canonical state of the system, to initialize additional back-ends that come online; ⑤ after a period of inactivity by a backend, possibly brought about by a network failure, the back-end attempts to register itself anew with the front-end.

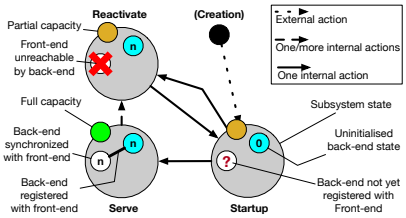


Figure 4: States of a back-end instance. After the instance is created, the $\tau_b :: \mathit{startup}$ junction is scheduled as described in Fig. 3. This either times out, resulting in $\tau_b :: \mathit{reactivate}$ being scheduled, or the instance is subsequently used to serve client requests through schedules of $\tau_b :: \mathit{serve}$. This diagram also explains visual cues used in later diagrams.

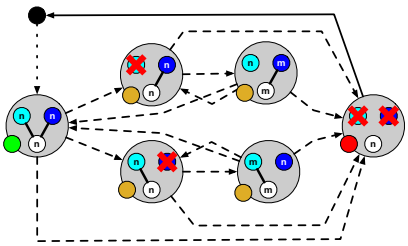


Figure 5: States of the backend-facing junction in the front-end instance. The system is in full capacity (left-most state) if both back-ends are online and synchronized. It fails (right-most state) if both back-ends are unavailable—making fail-over impossible. If at least one back-end is available then the system can operate but there is no fail-over capacity. If a back-end fails then it can recover when its state is resynchronized during the registration step with $\tau_f :: b$.

```
def τ_f :: b(backends, t̄) ◀
  | init data state
  | init prop Starting          | init prop ¬Active
  | init prop ¬Activating       | init prop ¬Retried
  | for tgt̃ ∈ backends  init prop ¬Backend[tgt̃] ❶
  if Starting then
    for b̃ ∈ backends +
      ⟨wait [] InitBackend[b̃] otherwise[t̄] skip⟩;
    retract [] HaveAtLeastOne;
    for b̃ ∈ backends ;
      if InitBackend[b̃] then
        ⟪ Initialize(b̃);
          # Next line relies on idempotence.
          assert [] HaveAtLeastOne;
        ⟫ otherwise[t̄] skip;
    if ¬HaveAtLeastOne then complain;
    retract [] Retried;
    case {
      Starting ⇒
        # Progress f::c beyond Starting.
        retract [f :: c] Starting otherwise[t̄]
          if ¬Retried then
            assert [] Retried;
          else complain();
        reconsider
      otherwise ⇒ skip
    }
  else
    case {
      Call ⇒
        ⟨ verify ¬Active;
          write(state, f :: c);
          assert [f :: c] Active;
          wait [state] ¬Active;
        ⟩ otherwise[t̄] complain();
        retract [] Call;
        break
      for b̃ ∈ backends  ¬Call ∧ InitBackend[b̃] ⇒
        Initialize(b̃) otherwise[t̄] skip;
        retract [] InitBackend[b̃];
        break
      otherwise ⇒ skip
    }
```

Figure 6: Code for the backend-facing junction in the front-end instance sketched in Fig. 5. The syntax at line ❶ shows the formation of a set from another set: Backend is a set of propositions that is indexed by a backend identifier.

```
InstanceTypes = {τ_f, τ_b}
Instances = {f : τ_f, b_1 : τ_b, b_2 : τ_b}
def main(t̄) ◀
  start b_1 startup(t̄) serve(t̄) reactivate(⌊3 ∗ t̄⌋)+
  start b_2 startup(t̄) serve(t̄) reactivate(⌊3 ∗ t̄⌋)+
  start f b({b_1 :: serve, b_2 :: serve} ❶, t̄)
    c({b_1 :: serve, b_2 :: serve}, t̄)
def complain ◀ ⌊...⌉; return
def Initialize(tgt) ◀ ❷
  verify ¬Activating ∧ ¬Active;
  write(state, tgt);
  assert [tgt] Activating; ❸
  wait [] ¬Activating;
  assert [tgt] Active;
  # If we fail on this, the backend won't be used
  # by f::c, and the backend will reattempt
  # reactivation later after a period of inactivity
  # expires.
  # 'f::c' below can be made into a parameter.
  assert [f :: c] Backend[tgt]; ❹
  retract [] Active;
```

Figure 7: Part of the architecture description for the fail-over architecture described in §2.3 of the C-Saw paper. *Initialize* is a function called to initialize a newly-registered backend $\overline{tgt}$. Location ❶ shows an example of passing set parameters in the DSL, and ❷ shows the declaration of the $\overline{tgt}$ parameter the is used as a destination junction in ❸, and as an index in ❹. These language features are described further in §6 of the C-Saw paper.

```
def τ_f :: c(backends, t̄) ◀
  | init prop Starting                    | init prop ¬Active
  | init prop ¬Req                        | init prop ¬Call
  | init prop ¬HaveAtLeastOne
  | init data state                       | init data req
  | init data preresp
  | for tgt̃ ∈ backends   init prop ¬Backend[tgt̃]
  | for tgt̃ ∈ backends   init prop ¬Running[tgt̃]
  # Req is asserted externally
  # to process client request.
  | guard ¬Starting ∧ Req
  retract [] Req;
  verify ¬Call;
  assert [f :: b] Call;
  wait [state] Active;
  restore(state, ...);
  retract [] Call;
  ⌊H_1⌉;
  save(..., req);

  retract [] HaveAtLeastOne;
  for b̃ ∈ backends +
    if Backend[b̃] then
      ⟪ verify S(b̃) ⟶ b̃@Active ∧ ¬b̃@Running[b̃];
        write(b̃, req);
        assert [b̃] Running[b̃];
        wait [preresp] ¬Running[b̃];
        assert [] HaveAtLeastOne;
      ⟫ otherwise[t̄] retract [] Backend[b̃];

  if ¬HaveAtLeastOne complain();
  verify HaveAtLeastOne;

  restore(preresp, ...);
  save(..., state);
  write(f :: b, state);
  ⌊H_3⌉;
  retract [f :: b] Active;
```

Figure 8: Code for the client-facing front-end junction in the fail-over architecture described in §2.3 of the C-Saw paper [6]. The code for the backend-facing front-end junction is shown in the paper.

```
def τ_b :: serve(t̄) ◀
  | init prop ¬Active                | init prop ¬Activating
  | init prop ¬RecentlyActive        | init data preresp
  | init data state                  | init data req
  | init prop ¬Running[me::junction]
  | guard Activating ∨ (Active ∧ Running[me::junction])
  case {
    Activating ⇒
      restore(state, ...);
      # If the remote retraction fails,
      # then b::reactivate will eventually
      # retry the startup.
      retract [f :: b] Activating otherwise[t̄]
        retract [] Activating;
      break
    otherwise ⇒
      assert [me::instance::reactivate] RecentlyActive
      restore(req, ...);
      ⌊H_2⌉;
      save(..., preresp);
      ⟨ write(f :: c, preresp);
        retract [f :: c] Running[me::junction];
      ⟩ otherwise[t̄] retract [] Active
  }

def τ_b :: startup(t̄) ◀
  | init prop ¬InitBackend[me::instance::serve]
  | guard ¬me::instance::serve@Active
  assert [f :: b] InitBackend[me::instance::serve]
    otherwise[t̄] skip

def τ_b :: reactivate(t̄) ◀
  | init prop ¬RecentlyActive
  | init prop ¬Active
  retract [] RecentlyActive;
  wait [] RecentlyActive otherwise[t̄]
    ⟨retract [me::instance::serve] Active;
      retract [me::instance::serve] Activating⟩;
```

Figure 9: Code for the back-end in the fail-over architecture sketched in §2.3 of the C-Saw paper [6].

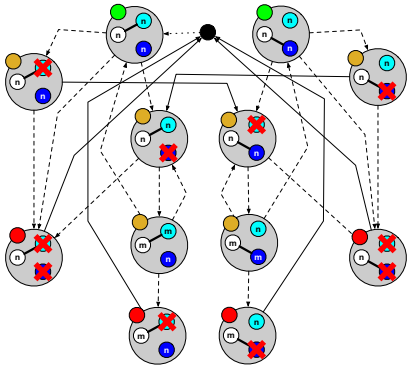


Figure 10: States of the front-end of the "watched" fail-over system described in §2.4.

```
InstanceTypes = {τ_f, τ_w, τ_o, τ_s}
Instances = {f : τ_f, w : τ_w, o : τ_o, s : τ_s}
def main(t̄) ◀
  (start w c_o() c_s() c_unrecov() + start o(t̄) + start s(t̄)) ; start f(t̄)
def complain ◀ ...
def RunBackend(n, t̄, tgt) ◀
  ⟪write(n, tgt); assert [tgt] Run[tgt]⟫
    otherwise[t̄] complain();
def τ_f :: (t̄) ◀
  | init prop ¬Reply
  | for tgt ∈ {o, s}   init prop ¬Run[tgt]
  | init prop ¬failover      | init prop ¬nofailover
  | init data n              | init data m
  # Junction won't be scheduled until ¬Reply.
  | guard ¬Reply
  ⌊H_1⌉; save(..., n);
  verify ¬Run[o] ∧ ¬Run[s] ∧ ¬Reply
  verify ¬ (failover ∧ nofailover)
  case {
    failover ∧ ¬nofailover ⇒
      RunBackend(n, t̄, s);
      break
    ¬failover ∧ nofailover ⇒
      RunBackend(n, t̄, o);
      break
    otherwise ⇒
      RunBackend(n, t̄, o) + RunBackend(n, t̄, s)
        otherwise[t̄] complain();
        # Here could implement more robust handling,
        # to retry RunBackend () for example.
  };
  # Don't wait too long for completion, prioritize
  # throughput.
  wait [m] Reply otherwise[t̄] return;
  # If Reply hasn't been reset in line above then this
  # junction won't be scheduled again because of guard.
  retract [] Reply;
  restore(m, ...);
  ⌊H_3⌉;
def Watch(tgt, prop) ◀
  | for tgt ∈ {o, s}   init prop ¬Run[tgt]
  | init prop ¬prop
  ⟪assert [tgt] prop; assert [f] prop⟫ otherwise complain()
def τ_w :: c_s() ◀
  | guard ¬S(o) ∧ S(s) ∧ S(f)
  Watch(s, failover)
def τ_w :: c_o() ◀
  | guard ¬S(s) ∧ S(o) ∧ S(f)
  Watch(o, nofailover)
def τ_w :: c_unrecov() ◀
  | guard ¬S(s) ∧ ¬S(o) ∨ ¬S(f)
  complain()
```


Figure 11: First half of the code for §2.4. Note the proposition name being passed as the second parameter to the function *Watch*; it must be resolvable at compile-time since functions behave as templates in this language.

```
def reply(t̄, other) ◀
  verify ¬Reply@f
  # Condition below isn't too strong since
  # either 's' or 'o' may Reply,
  # so we ensure that the other backend isn't
  # currently in Reply mode.
  verify ¬Reply@other
  ⟨save(..., m);
   write(m, f);
   assert [f] Reply;
  ⟩ otherwise⌈t̄⌉ complain();
def τ_s :: (t̄) ◀
  | for tgt̃ ∈ {s}   init prop ¬Run[tgt̃]
  | init prop ¬Reply
  | init data n      | init data m
  | guard Run[s]
  verify ¬Reply
  restore(..., n);
  ⌊H_2⌉;
  retract [f] Run[s];
    otherwise⌈t̄⌉ complain();
  case {
    failover ⇒
      reply(t̄, o);
      retract [] Reply;
      break;
    otherwise ⇒ ⌊skip⌉
  };
def τ_o :: (t̄) ◀
  | for tgt̃ ∈ {o}   init prop ¬Run[tgt̃]
  | init prop ¬Reply
  | init data n      | init data m
  | guard Run[o]
  verify ¬Reply
  restore(..., n);
  ⌊H_2⌉;
  retract [f] Run[o];
    otherwise⌈t̄⌉ complain();
  reply(t̄, s);
  retract [] Reply
```

Figure 12: Second half of the code for §2.4.

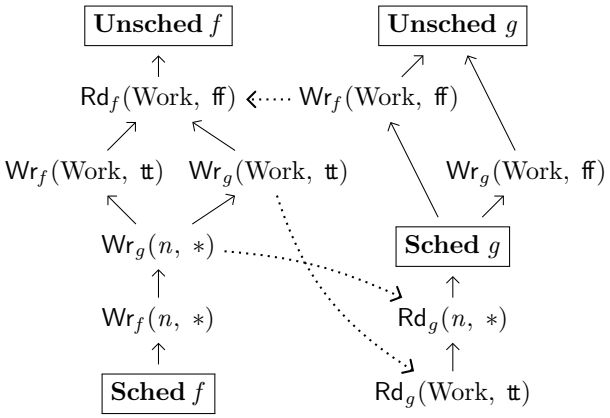


Figure 13: Part of the event structure for Fig. 3 from the C-Saw paper [6]. All arrows are enablement arrows, but arrows are dotted to emphasize cross-junction enablement. Scheduling events are shown boxed for emphasis.

3 Semantics

This section uses event structures [5] to give formal semantics to the C-Saw DSL. Intuitively, event structures describes enablement and conflict between events. This approach for describing semantics has been used to characterize concurrency of distributed and weakly-consistent systems [2], and it seemed like a suitable approach to use for C-Saw.

Fig. 14 shows a subset of C-Saw's semantics. Event structures are triples consisting of a set of events, and the enablement and conflict relations. In the subset above, an event is represented by a *label* describing that event such as "$\mathsf{Wr}_J(v,\ *)$"—which updates the value of data item v in the memory of junction J. In this subset of rules, the top rules only introduce new labels, and the bottom rule describes the parallel composition of the semantics. This form of composition simply unifies two structures; other forms of composition, such as ';', are more complex. Section 3.5 contains the rest of the rules.

We take advantage of the graphical notation of event structures to give examples of system behavior. Fig. 13 represents the system from Fig. 3 in the C-Saw paper [6]. These semantics reduce DSL behavior to a small set of general events, such as scheduling and unscheduling of a junction (**Sched** f and **Unsched** f), writes of data ($\mathsf{Wr}_f(n,\ *)$) and propositions ($\mathsf{Wr}_f(\text{Work},\ \mathtt{tt})$), and reads ($\mathsf{Rd}_f(\text{Work},\ \mathtt{ff})$). Symbols $\mathtt{tt}$ and $\mathtt{ff}$ represent "true" and "false" in the semantics. In this example, event $\mathsf{Wr}_f(\text{Work},\ \mathtt{tt})$ occurs when proposition Work is set to true in the memory of junction f; and $\mathsf{Rd}_f(\text{Work},\ \mathtt{ff})$ occurs when Work is read as false in the memory of junction f. This example does not involve conflict between events, which can arise when code branches. Section 3.6 contains larger examples based on another example of a DSL expression.

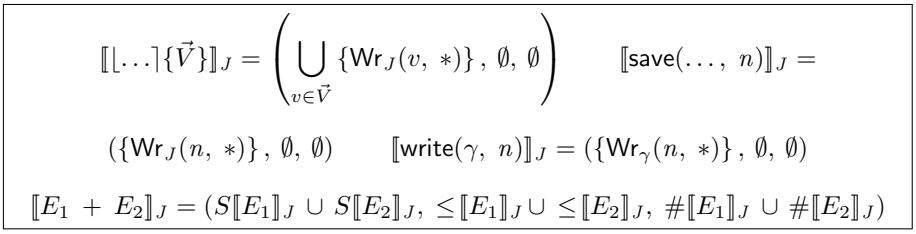


Figure 14: Semantic rules for part of the language syntax—the syntax is shown in Table 1 of the C-Saw paper.

"Local priority" rule. Junctions execute concurrently and may send messages to each other in parallel. Messages are used to perform updates to junctions' KV-tables. While a junction is running, updates are queued to take effect after the junction finishes executing, and before it is scheduled to execute again. If multiple updates to the same state occur then they are handled in the order that they are received—races are avoided by the design of the sychronization logic expressed in the DSL. A junction can only directly update another junction's state if the latter is executing wait on that state—for both propositions and data objects. If state updates arrive at a running junction, and that junction updates that same state, then the pending update will be ignored. That is, local updates have priority.

3.1 Event structures

This section starts by outlining the basic definitions of event structures [5] to make the description more self-contained. The cited literature provides the details and discussion related to the basic definitions of event structures.

An *event* is a triple $(id, label, outward)$ consisting of a unique identifier drawn from an inexhaustable set, a label, and a Boolean value labeled "outward". The labels used in C-Saw's semantics are defined in §3.2. "Outward" is used to track whether an event can enable events through composition, for instance events related to exception-handling. All events start out with "outward" being true, and it will be manipulated by some statements.

An *event structure* is a triple $(S, \leq, \#)$ consisting of: a set of events S, an enablement relation $\leq$ and a conflict relation $\#$. The $\leq$ relation is reflexive and transitive. The $\#$ relation is irreflective and symmetric. We previously encountered this triple in Fig. 14, and it will be used in §3.5 to give the remainder of the language semantics.

To qualify as an event structure the following properties must hold:
conflict inheritence:

$$\forall e_1, e_2, e_3 \in S.\ s_1 \# s_2\ \wedge\ s_2 \leq s_3\ \longrightarrow\ s_1 \# s_3$$

and *finite causes*:

$$\forall e \in S.\ |[e]| \in \mathbb{N}$$

where

$$[e] \;=\; \{e \in S \mid e' \leq e\}$$

Two events e_1, e_2 are *concurrent* if they are incomparable by enablement and are not conflicting:

$$e_1 \not\leq e_2 \;\wedge\; e_2 \not\leq e_1 \;\wedge\; \forall e_1' \in [e_1], e_2' \in [e_2].\; \neg(e_1' \# e_2')$$

3.2 Labels

Labels represent the activity taking place during an event. Examples of labels were previously given in §3, and in this section we describe the remaining labels that are used in C-Saw's semantics.

The full set of labels is:

$$\begin{aligned} L \in \quad & \{\mathsf{Rd}_J(K,\, V),\; \mathsf{Wr}_J(K,\, V),\; \mathsf{Start}_J(\gamma),\; \mathsf{Stop}_J(\gamma), \\ & \quad \textbf{Sched}\; J,\; \textbf{Unsched}\; J,\; \mathsf{Synch}_J(\vec{K}),\; \mathsf{Wait}_J(\vec{K}, K)\;\} \end{aligned}$$

Further to the labels described in §3, $\mathsf{Synch}_J(\vec{K})$ represents a synchronization barrier across concurrent event chains. This is an intermediate event that is inserted by the semantics during some operations to preserve intuition, and an example will be seen soon. $\mathsf{Wait}_J(\vec{K}, K)$ is a placeholder label that is decomposed into a pattern of network events at a later stage to simplify the semantics, as will be described in §3.5.

The examples in the C-Saw paper abstract some behavior, such as the *complain* () function:

$$\textbf{def}\; \underline{\textit{complain}}() \;\blacktriangleleft\; \ldots$$

in Fig. 4 of the C-Saw paper [6]. We represent this abstracted behavior using ad hoc labels such as the "complain" label in §3.6.

3.2.1 Graphical notation

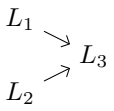
Event structures can be represented graphically as shown in Fig. 13. This section describes the notation more accurately. A larger example will be given in §3.6.

The graphical notation captures event structures' formalization of enablement and conflict between events. In this notation, events are represented using their labels.

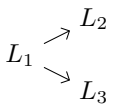
The notation relies on two key definitions. The first is *immediate causality*, represented by an arrow between two events. This captures a minimal form of enablement: " $L_1 \longrightarrow L_2$ " iff, taking e_i to correspond with L_i: $e_1 \lesssim e_2$ and $\neg\exists e'.\; e_1 \lesssim e' \;\wedge\; e' \lesssim e_2$.

The second is *minimal conflict*, represented by a zizag between two events. This captures a minimal form of conflict: " $L_1 \rightsquigarrow L_2$ " iff, taking e_i to correspond with L_i: $e_1 \# e_2$ and $\forall e, e'.\; e \leq e_1 \wedge e' \leq e_2 \wedge e \# e' \longrightarrow e = e_1 \wedge e' = e_2$. (Note that the arrow used here denotes material implication, and is a different arrow than that used for immediate causality.)

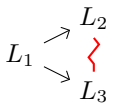
The graphical notation can convey an intuition of the behavior of a system that is described by an event structure. The notation $L_1 \rightarrow L_2$ means that L_1's event is *necessary* for L_2 to occur. Furthermore, fan-in events are *conjunctive*; that is, L_3 below can only occur if both L_1 and L_2 occur:



Fan-out events create parallel chains of event execution:



And such parallel chains can be mutually exclusive if they are conflicting, as shown below:



3.3 Supporting definitions

This section provides some definitions used when giving semantics to the C-Saw DSL.

The isolate function mutates an event to set its *outward* flag to false. This is used in the semantics to capture event interactions for exception-handling, as will be seen by the semantics of $\langle\!| \cdot |\!\rangle$ and otherwise.

$$\text{isolate}\,((id, label, outward)) = (id, label, \mathsf{ff})$$

This function will also be lifted to work on sets of events.

The DSL semantics will be expressed using $[\![\cdot]\!]^{\eta}_{J}$, where J is the junction in which the semantics are being evaluated and η is a finite function that maps to DSL statements. It is initialized as follows:

$$\{\mathsf{sub} \mapsto \mathsf{skip},\ \mathsf{return} \mapsto \mathsf{skip},\ \mathsf{break} \mapsto \mathsf{skip},\\ \mathsf{reconsider} \mapsto \mathsf{skip},\ \mathsf{next} \mapsto \mathsf{skip}\}$$

The parameter η is used to give semantics to statements that affect control flow. sub tracks which statement will be evaluated next in sequence, and the other values will depend on sub to some extent—this will be made clear by the semantics. Parameter η will be changed while recursively evaluating the semantics of DSL statements, but J will remain fixed. We will use the notation $\eta\{\mathsf{return} \mapsto \eta(\mathsf{sub})\}$ to denote the update of η such that return is changed to map to $\eta(\mathsf{sub})$. When redundant, J and η will be omitted from the notation.

The next two definitions gather the rightmost and leftmost periphery of an event structure, and are used when composing event structures together:

$$\overset{\Rightarrow}{[\![E]\!]} = \begin{cases} S & \text{if } \text{“}\leq\text{”} = \emptyset \\ \{e \in S \mid \nexists e'.\ e \leq e'\} & \text{otherwise} \end{cases}$$

$$\overset{\Leftarrow}{[\![E]\!]} = \begin{cases} S & \text{if } \text{“}\leq\text{”} = \emptyset \\ \{e \in S \mid \nexists e'.\ e' \leq e\} & \text{otherwise} \end{cases}$$

To make fresh copies of event structures we use a map $\natural(\text{idx}, [\![E]\!])$ where idx is an arbitrary object used for indexing. This map creates a copy of events, updating their identifier to make them unique, and preserves their enablement and conflict relations. This map is used to describe the semantics of composition operators that lead to distinct but similar future behavior of a system, such as the E_1 $\mathsf{otherwise}$ E_2 operator that maps arbitrary failure during E_1 to execute E_2. For $e \in S[\![E]\!]$, we define $\natural_{\text{idx}} e$ to be the unique bijection to $S\natural(\text{idx}, [\![E]\!])$. The symbol idx is dropped when it is obvious from the context or if it is trivial (unique).

The function $\mathcal{N}$ is used to decompose case statements and give semantics to the next terminator by progressively reducing the cases that can apply.

$$\mathcal{N}\left[\begin{array}{l} \mathsf{case}\ \{ \\ \quad F_1 \Rightarrow E_1;\ T_1 \\ \quad F_2 \Rightarrow E_2;\ T_2 \\ \quad \vdots \quad\quad \vdots \\ \quad \mathsf{otherwise} \Rightarrow E_n \\ \} \end{array}\right] \mapsto \left[\begin{array}{l} \mathsf{case}\ \{ \\ \quad F_2 \Rightarrow E_2;\ T_2 \\ \quad \vdots \quad\quad \vdots \\ \quad \mathsf{otherwise} \Rightarrow E_n \\ \} \end{array}\right] \text{ if } n > 2$$

This function is undefined if the case expression contains only one case—in which case next cannot be used—or is malformed.

Another supporting definition involves a scheme to decompose a formula F into primitive events that relate to each proposition involved in F. For this we first convert F into its disjunctive normal form [4] (DNF):

$$\bigvee \bigwedge \{P, \neg Q, \ldots\}$$

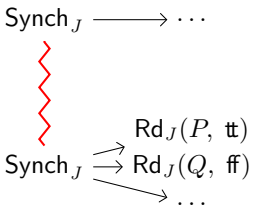
Next, that is converted into sets of sets of literals (propositions or their negations):

$$\{\ldots, \{P, \neg Q, \ldots\}, \ldots\}$$

Finally, these are mapped these into read-event labels:

$$\{\ldots, \{\mathsf{Rd}_J(P,\ \mathtt{tt}), \mathsf{Rd}_J(Q,\ \mathtt{ff}), \ldots\}, \ldots\}$$

Each element set represents a combination of reads than can guard subsequent logic. Each element set is structured into parallel events that are collectively prefixed by a Synch, and such that each element set is a strict alternative:



3.4 Program semantics

Mapping programs into event structures involves the following steps:

1. Functions are inlined. They have no distinct semantic meaning since they are templates (see §6 of the paper [6]).
2. Expressions only consist of formulas, ranged over by the metavariable F in Table 1 of the paper, and are in converted to DNF as described above.
3. Statements, including junction definitions, are mapped using the definitions in §3.5.
4. A post-processing step described in §3.5 expands placeholder events into atomic events.
5. A start-up portion, described next, is added to complete the program-level semantics.

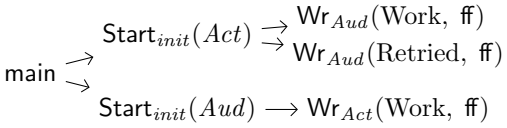
Start-up. The start-up portion of a program initializes and starts instances from a distinguished start-up instance. In involves two special names:

- The externally-occuring main event enables the subsequent events as defined by the semantics of the program's main statement:

$$\textbf{def}\ \mathsf{main} \blacktriangleleft \ldots$$

- The distinguished *init* junction represents the instance responsible for start-up.

The start-up behavior of the example in Fig. 4 from the paper is shown below. The rest of its semantics is visualized in §3.6.



3.5 DSL statement semantics

This section provides a general, infinitary version of the semantics for a DSL. That is, events have finite support as required by the definition of event structures, but branches may have infinite depth because subsumed subtrees are not filtered—a proposition that is set to false might later be used to define behavior when the proposition's value is true. This expands the semantics with redundant behavior that can be eliminated—either during a later deflationary pass or by construction. Formalizing a more accurate semantics is left as future work. The language's implementation only requires a weaker version of this semantics where unnecessary program behavior is curtailed.

Fig. 15 shows the semantic definitions for most statements. Two statements are handled separately because their behavior requires more explanation.

The first is the case expression. Let E be:

$$\begin{array}{l}\mathsf{case}\ \{\\ \quad F_1\ \Rightarrow\ E_1;\ T_1\\ \quad F_2\ \Rightarrow\ E_2;\ T_2\\ \quad \vdots \qquad \vdots\\ \quad \mathsf{otherwise}\ \Rightarrow\ E_n\\ \}\end{array}$$

In order to define $[\![E]\!]^\eta$ we make some intermediate definitions, starting with adaptations of η:

$$\begin{array}{lcl}\eta' & = & \eta\{\mathsf{break} \mapsto \eta(\mathsf{sub}),\ \mathsf{reconsider} \mapsto E\}\\ \eta'_i & = & \eta'\{\mathsf{next} \mapsto E'_i\} \quad \text{where } i < n\\ \eta'_n & = & \eta'\{\mathsf{next} \mapsto \mathbf{undef}\}\end{array}$$

where E'_i (where $i < n$) is:

$$\begin{array}{l}\mathsf{case}\ \{\\ \quad F_{i+1}\ \Rightarrow\ E_{i+1};\ T_{i+1}\\ \quad \vdots \qquad \vdots\\ \quad \mathsf{otherwise}\ \Rightarrow\ E_n\\ \}\end{array}$$

The remaining intermediate definition is:

$$\mathsf{case}(i) = \begin{cases} [\![E_i;\ T_i]\!]^{\eta'_i} \quad \mathsf{case}(i+1) & \\ \quad\uparrow \qquad\qquad\uparrow & \\ [\![F_i]\!]^{\eta'_i} \sim\sim [\![\neg F_i]\!]^{\eta'_i} & \text{if } i < n \\ & \\ & \\ [\![E_n]\!]^{\eta'_n} & \text{if } i = n \end{cases}$$

$$[\![\mathsf{assert}\ [\gamma]\ P]\!]_J = (\{\mathsf{Wr}_J(P,\ \mathsf{tt}),\ \mathsf{Wr}_\gamma(P,\ \mathsf{tt})\}\,,\ \emptyset,\ \emptyset) \qquad [\![\mathsf{retract}\ [\gamma]\ P]\!]_J = (\{\mathsf{Wr}_J(P,\ \mathsf{ff}),\ \mathsf{Wr}_\gamma(P,\ \mathsf{ff})\}\,,\ \emptyset,\ \emptyset)$$

$$[\![\mathsf{skip}]\!]_J = [\![\mathsf{restore}(n,\ \ldots)]\!]_J = (\emptyset,\ \emptyset,\ \emptyset) \qquad [\![\langle E\rangle]\!]^\eta_J = [\![E]\!]^{\eta\{\mathsf{return}\mapsto\eta(\mathsf{sub})\}}_J \qquad [\![\mathsf{wait}\ [\vec{n}]\ F]\!]_J = (\{\mathsf{Wait}_J(\vec{n}, F)\}\,,\ \emptyset,\ \emptyset)$$

$$[\![\mathsf{return}]\!]_J = [\![\eta(\mathsf{return})]\!]_J \qquad [\![\mathsf{start}\ \iota]\!]_J = (\{\mathsf{Start}_J(\iota)\}\,,\ \emptyset,\ \emptyset) \qquad [\![\mathsf{stop}\ \iota]\!]_J = (\{\mathsf{Stop}_J(\iota)\}\,,\ \emptyset,\ \emptyset)$$

$$[\![E_1;\ E_2]\!]^\eta_J = \left(S[\![E_1]\!]^{\eta\{\mathsf{sub}\mapsto E_2\}}_J \cup S[\![E_2]\!]^\eta_J,\quad \leq [\![E_1]\!]_J \cup \leq [\![E_2]\!]_J \cup \bigcup_{e_1 \in S\vec{[\![E_1]\!]}} \left\{ (e_1, e_2) \mid e_2 \in S\overleftarrow{[\![E_2]\!]} \right\},\quad \#[\![E_1]\!]_J \cup \#[\![E_2]\!]_J \right)$$

$$\begin{aligned}
[\![E_1 \parallel E_2]\!]_J = \ & \Big(S[\![E_1]\!]_J \cup S[\![E_2]\!]_J \cup \textstyle\bigcup_{e\in S[\![E_1]\!]} \{\natural e\} \cup \bigcup_{e\in S[\![E_2]\!]} \{\natural e\}\,, \\
& \leq [\![E_1]\!]_J \cup \leq [\![E_2]\!]_J \cup \\
& \quad \left\{ (e, \natural e' \mid e \in S\vec{[\![E_1]\!]},\ e' \in S[\![E_2]\!] \right\} \cup \left\{ (e, \natural e' \mid e \in S\vec{[\![E_2]\!]},\ e' \in S[\![E_1]\!] \right\} \cup \\
& \quad \left\{ (e, \natural e \mid e \in S[\![E_1]\!] \setminus S\vec{[\![E_1]\!]} \right\} \cup \left\{ (e, \natural e \mid e \in S[\![E_2]\!] \setminus S\vec{[\![E_2]\!]} \right\}, \\
& \#[\![E_1]\!]_J \cup \#[\![E_2]\!]_J \cup \{(e_2, \natural e_1 \mid e_2 \in S[\![E_1]\!],\ e_1 \lneq e_2\} \cup \{(e_2, \natural e_1 \mid e_2 \in S[\![E_2]\!],\ e_1 \lneq e_2\})
\end{aligned}$$

$$\begin{aligned}
[\![E_1\ \mathsf{otherwise}\ E_2]\!]_J = \ & \Big(\mathrm{isolate}\,[S[\![E_1]\!]_J] \cup \textstyle\bigcup_{e\in S[\![E_1]\!]} \{S\natural(e, [\![E_2]\!])\}, \\
& \leq [\![E_1]\!]_J \cup \textstyle\bigcup_{e\in S[\![E_1]\!]} \{\leq\natural(e, [\![E_2]\!])\} \cup \bigcup_{e\in S[\![E_1]\!]} \{(e', e'') \mid e'' \in \overleftarrow{\natural}\ (e, [\![E_2]\!]),\ e' \lneq e\}, \\
& \#[\![E_1]\!]_J \cup \textstyle\bigcup_{e\in S[\![E_1]\!]} \{\#\natural(e, [\![E_2]\!])\} \cup \bigcup_{e\in S[\![E_1]\!]} \{(e, e') \mid e' \in \overleftarrow{\natural}\ (e, [\![E_2]\!])\} \Big)
\end{aligned}$$

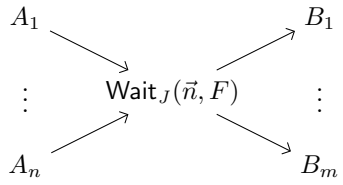
$$[\![\mathsf{reconsider}]\!]_J = [\![\eta(\mathsf{reconsider})]\!]_J \qquad [\![\mathsf{retry}]\!]_J = [\![J]\!]_J \qquad [\![\mathsf{next}]\!]_J = [\![\eta(\mathsf{next})]\!]_J \qquad [\![\mathsf{break}]\!]_J = [\![\eta(\mathsf{break})]\!]_J$$

$$\begin{aligned}
[\![\langle\!|E|\!\rangle]\!]^\eta = \ & \Big(\mathrm{isolate}\left[S[\![E]\!]^{\eta'} \right] \cup \{e'\}\,,\ \leq [\![E]\!] \cup \textstyle\bigcup_{e\in S\overleftarrow{[\![E]\!]}} \{(e', e)\},\ \#[\![E]\!] \Big) \\
& \text{where } \eta' = \eta\{\mathsf{return} \mapsto \eta(\mathsf{sub})\} \text{ and } e' = \mathsf{Synch}_J.
\end{aligned}$$

Figure 15: Semantics of DSL statements, continuing from Fig. 14.

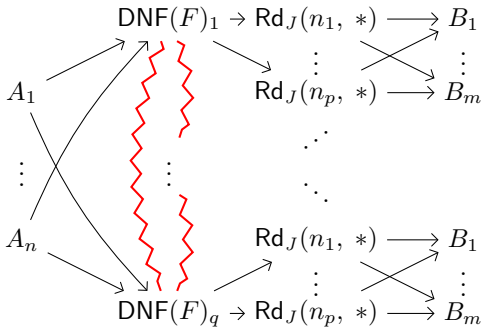
Finally, $[\![E]\!]^\eta = \mathsf{case}(0)$,

The second is the wait statement. It is initially mapped to a "$\mathsf{Wait}_J(\vec{n}, F)$" event which can generally interconnect with other events as shown below:



We then expand $\mathsf{Wait}_J(\vec{n}, F)$ into a set of two kinds of events. First, events that include the DNF-expansion of F, shown here as a q-ary set of disjuncts: $\mathsf{DNF}(F)_1, \ldots, \mathsf{DNF}(F)_q$. Second, the reads of data state $\vec{n}$: $\mathsf{Rd}_J(n_1, *), \ldots, \mathsf{Rd}_J(n_p, *)$

These sets of events are then interconnected as shown below. This is designed to stage the evaluation of the wait statement: first determine that F is satisfied, then read $\vec{n}$.



3.6 Example

This section uses the graphical notation described in §3.2.1 to illustrate the event structure for the example described in §5.1 of the paper [6]. The start-up behavior of this example was shown in §3.4.

There are two instances in this example. The behavior of *Act* is shown next, and that of *Aud* is shown in Fig. 16.

The instances interact implicitly by updating propositions in each other's KV-tables. *Act* engages *Aud* at the occurrence of event $\mathsf{Wr}_{\{Act,Aud\}}(\text{Work}, \mathsf{tt})$, and is engaged back when $\mathsf{Rd}_{Act}(\text{Work}, \mathsf{ff})$ occurs. The complexity of *Aud* 's behavior in Fig. 16 arises from the combination of τ_{Auditing}'s retry logic and its failure-handling.

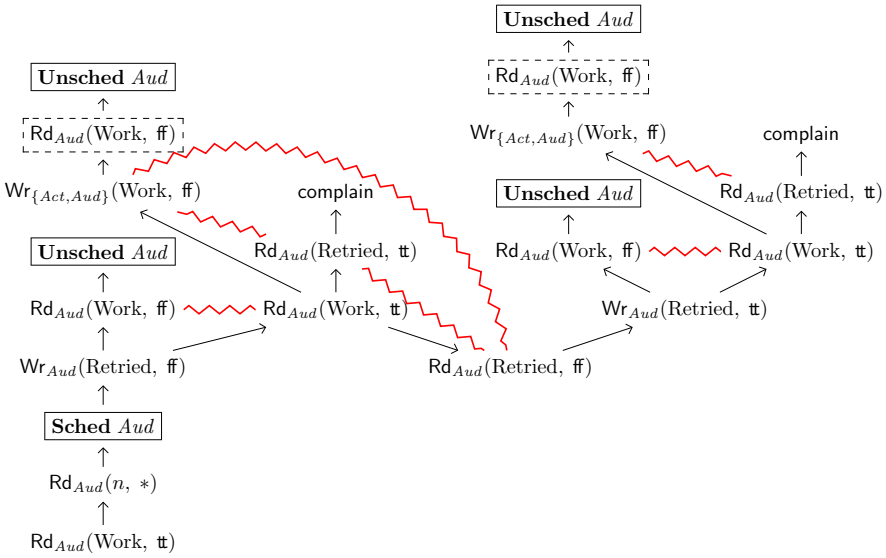
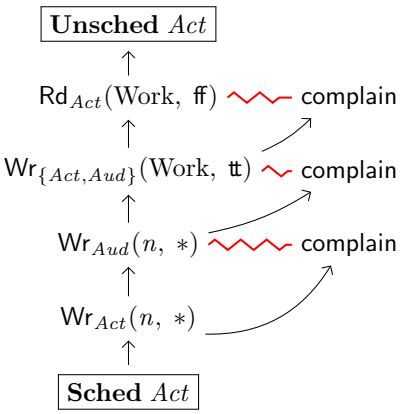


Figure 16: Behavior of the *Aud* instance from §5.1 of the C-Saw paper.



3.7 Topology

The topology of a C-Saw-architected system, showing the communication paths between components, is derived from the definition of Topo:

$$\text{Topo} = \bigcup_{\iota \in \mathsf{Instances}} \bigcup_{\gamma \in \text{Junctions}(\iota)} \{(\gamma, \gamma') \mid \gamma' \in \text{Topo}_\gamma(E_\gamma)\}$$

Topo produces a directed graph whose nodes are junctions and whose edges indicate communication from one junction to another. Its definition depends on the following definitions:

- Instances (see §4 from the C-Saw paper)
- Junctions(ι), which maps an instance to its set of junctions (by analysis of C-Saw expressions),
- E_γ, which is the DSL statement of junction γ.
- $\text{Topo}_\gamma(E)$, which recursively computes the set of communication targets for junction γ by analyzing the syntax of the junction's DSL expression. For example, the statements "assert $[\gamma']$ P" "retract $[\gamma']$ P" and "write(γ', n)" would return the set $\{\gamma'\}$; "$\langle E' \rangle$" evaluates to $\text{Topo}_\gamma(E')$; and "E_1; E_2" evaluates to $\text{Topo}_\gamma(E_1) \cup \text{Topo}_\gamma(E_2)$.

4 Further evaluation

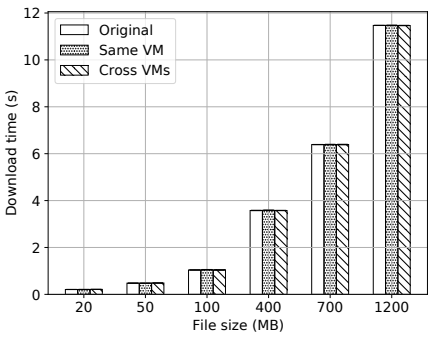
This section contains additional graphs that complement those in the evaluation given in §7 of the C-Saw paper [6].

Fig. 17a shows the performance of modified cURL when executed over large files, and complements Fig. 13a in the C-Saw paper which focused on small files. The performance difference for large files is less intelligible.

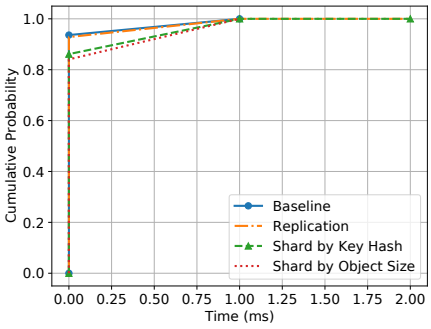
Fig. 17b shows the performance overhead of various reconfigurations of Redis under a SET workload. It is the complement of the paper's Fig. 13c.

Fig. 17c shows the behavior of Redis reconfigured for object-size sharding when subjected to a workload featuring a corresponding distribution to that used for key-based sharding in Fig. 11b.

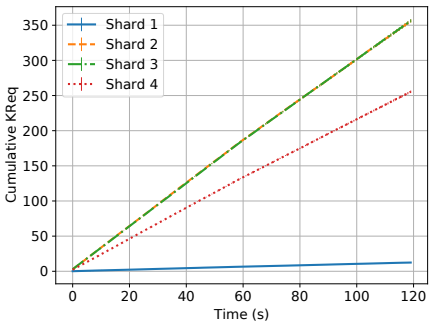
Fig. 17d extends the evaluation in §8.3 of the C-Saw paper. It shows the normalized overhead when Suricata's architecture was modified to support sharding, compared to the original version of Suricata. In this version we used a batch size of 2048 to mitigate overhead.



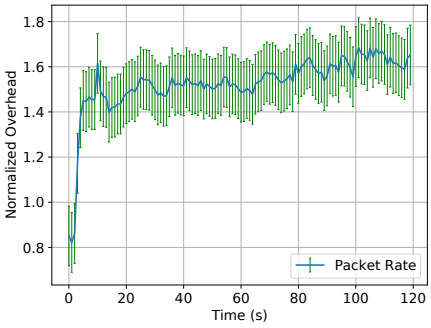
(a) Performance of modified cURL.



(b) Performance overhead of modified Redis (SET).



(c) Redis sharding based on object size.



(d) Suricata sharding overhead.

Figure 17: Additional graphs from experiments.

References

[1] Nicholas Carriero and David Gelernter. Linda in Context. *Commun. ACM*, 32(4):444–458, April 1989.

[2] Simon Castellan. Weak memory models using event structures. In Julien Signoles, editor, *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Saint-Malo, France, January 2016.

[3] D. Clark. The Design Philosophy of the DARPA Internet Protocols. *SIGCOMM Comput. Commun. Rev.*, 18(4):106–114, aug 1988.

[4] David Gries and Fred B Schneider. *A logical approach to discrete math.* Springer Science & Business Media, 2013.

[5] G Winskel. Event Structures. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, page 325–392, Berlin, Heidelberg, 1987. Springer-Verlag.

[6] Henry Zhu, Junyong Zhao, and Nik Sultana. A Domain-Specific Language for Reconfigurable, Distributed Software Architecture. In *15th Workshop on Advances in Parallel and Distributed Computational Models (to appear)*. IEEE, 2023.