International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 14, Number 1, pages 40–80, January 2024

A Domain-Specific Language for Reconfigurable, Distributed Software

Henry Zhu
University of Illinois Urbana-Champaign, Department of Computer Science,
Urbana, Illinois, United States

Junyong Zhao
University of Arizona, Department of Computer Science,
Tucson, Arizona, United States

Nik Sultana
Illinois Institute of Technology, Department of Computer Science,
Chicago, Illinois, United States

Received: July 25, 2023
Revised: October 18, 2023
Accepted: November 24, 2023
Communicated by Masahiro Shibata

Abstract

A program's architecture—how it organizes the invocation of application-specific logic—influences important program characteristics including its scalability and security. Architecture details are usually expressed in the same programming language as the rest of a program, and can be difficult to distinguish from non-architecture code. Once defined, a program's architecture is difficult and risky to change because it couples tightly with application logic over time.

We introduce C-Saw: an approach to express a software's architecture using a new embedded domain-specific language (EDSL) designed for that purpose. It *decouples* application-specific logic from architecture, making it easier to identify architectural details of software. C-Saw leverages three ideas: (i) introducing a new, formally-specified EDSL to separate an application's architecture description from its programming language; (ii) reducing architecture implementation to the definition and management of distributed key-value tables, and (iii) introducing an expressive state-management abstraction for distributed applications.

We describe a prototype implementation of C-Saw for C programs and use its implementation to build end-to-end examples of expressing and changing the architecture of widely-used, third-party software. We evaluate this on Redis, cURL, and Suricata and find that C-Saw provides expressiveness and reusability, requires fewer lines of code when compared to directly using C to express architectural patterns, and imposes low performance overhead on typical workloads.

Keywords: Key-Value Tables, Process Algebra, Coordination Language, Domain-Specific Language

1 Introduction

Software's architecture describes its fundamental information-processing structure [39] and varies in its complexity. Examples of architecture include: a sequence of processing steps, a pipeline of concurrent stages, an event-handling system, a fan-out to worker instances, and a mix of these patterns [27].

The choice of architecture influences important software characteristics such as security [30] and performance [48]. For example, architecture affects how software can scale to meet demand by harnessing additional resources to distribute the load from computation, communication and storage demands.

Since both *architecture* and the *application-specific* logic are usually described using the same programming language, there is no language-level distinction between them, and no obstacle to them being tightly coupled over time. As a result, it can be difficult to alter one without affecting the other [32]. The blurring of architecture and logic complicates the implementation of important features that depend on architecture-level changes. Fig. 1 shows examples of such features which include caching and load-balancing.

As a result of architecture's poor visibility in source code and its coupling with non-architecture code, architecture-level changes are *high-friction*: they take effort, risk introducing bugs, and create a maintenance burden if the software diverges from an up-stream, canonical open-source version. One could avoid architecture-level change by designing an overly-general architecture to begin with, but this raises practitioners' red flags because it risks "premature optimization" [33], "creeping elegance" [22], and introducing a "bad smell" from needless complexity due to "speculative generality" [26]. Even then, general interfaces might not forestall the need for eventual revision since the software's requirements can evolve.

To avoid these problems, we need a low-friction method to express software's architecture. It needs to support a range of architecture patterns, be linguistically distinguished from application logic, and induce low overhead. New and existing software could then be adapted more easily to respond to new and changing needs that require architecture-level changes.

In this paper, we introduce C-Saw ("see-saw"): an approach to express a software's architecture using a new embedded domain-specific language (DSL) designed for that purpose. C-Saw relies on distributed key-value tables to track both architecture-related state and application-logic state. These tables are managed by DSL expressions. The DSL is inlined into the application source-code and it is designed to work with existing software and languages—we prototyped this for the C language and developed usage examples involving widely-used, third-party applications.

The DSL can express a set of *architectures* that serve commonly-occurring *needs* such as those serviced by the examples in Fig. 1. These needs include: **(i)** *availability* through fail-over or replication; **(ii)** *manageability* through live migration or scale-out; **(iii)** *performance* through caching for latency, load-balancing for throughput, or object-size sharding for lower scheduling overhead; **(iv)** *lower resource cost* through scale-in or fusion of instances; **(v)** *security* through remote auditing.

The key idea in C-Saw involves decoupling an application's general architecture description from its application-specific logic. C-Saw shrinks the scope of understanding and changing a software's architecture, thus lessening the effort and risk. The DSL has restricted expressiveness to limit unwanted behaviors. It provides a concise syntax and formal semantics to channel intuition into short and accurate architecture specifications.

We found that even intuitively-simple needs can have subtle and complex specifications, and this underscored the importance of using a specialized language for describing architectures. For example, we explored several formulations of fail-over logic that differed in redundancy, resourcing, and complexity. Another benefit of using the DSL is that architecture specifications are more *reusable* since they are decoupled from application-specific logic. We evaluate reuse of logic expressed using the DSL in our prototype.

C-Saw is inspired by coordination languages [9], Architectural Description Languages (ADLs) [8], and process algebrae [37], but it has important differences: (i) C-Saw is designed for use in existing software and tool-chains rather than necessitate a rewrite into a new language, and (ii) Unlike process algebrae, C-Saw was designed for use in deployment contexts that do not typically allow an all-to-all communication model and higher-order channels between distributed components, in the interest of improved security and lower overheads. Also, more emphasis is placed on the interface to the host language which is used to describe application logic.

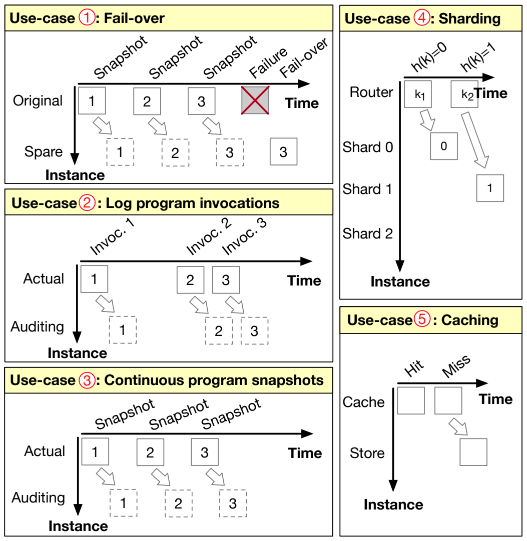


Figure 1: Behavioral sketches of architectural configurations that provide important application features. These diagrams give a simplified glimpse of architecture-related interactions over time. The x-axis shows events occurring over time, such as state-snapshots, application invocations, failures, or queries. The y-axis shows state synchronization occurring across specialized *instances* derived from the original software. Instances are components of the original application that capture distinct non-architectural features. Sketch ① describes fail-over, ② and ③ describe remote, integrity-preserving auditing of a local process (the first is *one-time* and the second is *continuously* running), ④ describes sharding, and ⑤ describes caching. These examples are detailed further in §2.

C-Saw's prototype uses libcompart [44] as a distributed runtime that coordinates logic across the program's architecture. It provides configuration, communication, and fault-handling support. Unlike microservices [13,35,47] and distributed OSs [38], C-Saw is focused on a language-level scope. Broader OS-level scoping is left for future research.

C-Saw's current design necessitates modification of the software's source code to derive subprograms that the DSL-expressed architecture will then interface with. Currently, this modification is done manually, and we evaluate its intrusiveness for the third-party systems to which we applied the C-Saw prototype. Automating this analysis and transformation is a separate research project and is left as future work.

This paper makes the following novel **contributions**: (1) A design (§3) that integrates C-Saw with existing C code-bases. (2) A formally-specified domain-specific language (§6) to separate an application's architecture description from its programming language. (3) A diverse suite of DSL-encoded architectures (§5) covering all the examples shown in Fig. 1, all of which were implemented on third-party software (Redis, cURL, and Suricata). (4) An expressive state-management abstraction (§9) that supports the DSL's C prototype to provide distributed synchronization. (5) A prototype implementation of C-Saw's DSL and state-management abstraction for C, and this prototype's evaluation (§10) for expressiveness, reusability, effort and performance on typical workloads.

The C-Saw prototype and evaluation suite are made freely available [5].

2 Software Architecture Use-Cases

Features such as those in **(i)-(v)** from the previous section are typically cross-cutting, application-wide features that rely on architecture-level support. This section defines software architecture to make this paper self-contained. It also gives examples of architectural modifications of widely-used, open-source systems that are used to evaluate C-Saw later in the paper: cURL, Redis, and Suricata.

We define *architecture* to mean the code that defines the *delivery system* of *work* between other code that implements application logic. Architecture is generally dynamic: it does not only describe a structure of application-logic invocations, but describes how that structure responds to external or internal changes—such as changes in demand or in resource availability. Perry and Wolf [39] provide an excellent further discussion of software architecture with examples.

Changing a software's architecture can affect its *capacity to do work*, or *differentiate kinds of work from each other*. In Fig. 1, example ① increases work capacity across a failure mode, and example ② adds new work: capturing select state for remote auditing.

Redis is a widely-used [21] NoSQL database [16] that is implemented as a single-threaded server [4]. We envisage three scenarios where architectural changes could benefit Redis, corresponding to examples ④, ⑤ and ① in Fig. 1. **(i) Scaling:** Redis does not automatically scale with the number of CPU cores. It is scaled by manually starting more Redis instances or by using external harnesses such as Redis Cluster [19] but this relies on all-to-all TCP connections. It also relies on Redis Cluster or external tools [31] for sharding [18]. By internalizing this feature we can save overhead and resources. **(ii) Availability:** Redis employs a leader-follower system for replication [20]. The leader streams changes to the follower process. An architecture-level approach to providing this feature involves on-demand checkpointing of Redis—the architecture would serialize state from across an instance—and resuming Redis from a checkpoint. This approach also weakens the requirement for the leader and follower to be synchronously active. **(iii) Performance:** Redis instances are typically heavy users of memory, and by having a cache for frequently-accessed objects we can evaluate configurations that outperform a instance [36].

cURL [46] is a library and client for transferring data using a variety protocols. It is widely-used [45] and we envisage scenarios where its architecture is changed to support remote auditing—such as examples ② and ③ in Fig. 1. For example, this could be used on employer-provided machines or storage partitions as part of a security or compliance-checking policy for a company that allows Bring-Your-Own-Device (BYOD) [10]. Or for device owners to better track their smart consumer electronics [29]. We subdivide this scenario into two use-cases: The **first** captures program state at a key point of an invocation, such as at the start of the program. The **second** captures program state continuously, trading-off a higher runtime overhead to acquire more information. State is logged remotely to protect its integrity. Both of these configurations rely on a state-snapshotting feature, similar to that described for Redis above.

Suricata [25] is one of the three foremost systems used for network security monitoring [2]. It implements a graph-based abstraction for packet handling, reminiscent of Click [34]. Packet analysis and threat detection tasks are interconnected in this graph, which is executed on a multi-threaded abstraction of the underlying hardware. We envisage two scenarios for changing Suricata's architecture, corresponding to ① and ④ in Fig. 1. **(i) Availability+Diagnostics:** At present, improving Suricata's availability requires external infrastructure to create a live Suricata replica that takes over in case of a crash. But whatever caused one Suricata instance to fail might cause its replica to fail too unless the cause was non-deterministic or resource-related. We can create a modified form of availability that involves continuously checkpointing Suricata state and resuming from the checkpoint in case of a crash. If the replica fails too, then we can use the checkpoint to reproduce the fault and understand it. To this end, we reuse the architectural pattern described earlier for fail-over in Redis, and interface it with Suricata's task graph. **(ii) Flow-level resourcing for performance:** We reuse the sharding logic from the earlier change to Redis' architecture, and use this to reserve resources for specific network flows identified as a 5-tuple [52]. This architectural configuration adds a policy layer on top of Suricata's allocation of cores to reserve some cores to process traffic of interest.

3 C-Saw

This section explains the workflow for adapting software to use C-Saw. The main steps in this workflow are shown in Fig. 2. The abstract description of the workflow in this section introduces the concepts of *junctions*, *instances*, and *instance types*. The sections that follow will provide concrete examples of applying C-Saw.

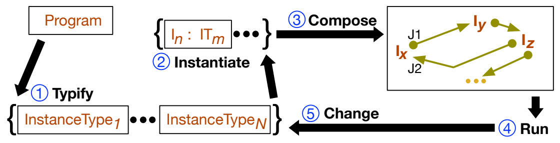


Figure 2: Workflow for C-Saw.

Adapting software to use C-Saw Step ① in Fig. 2 involves **typifying** a program: this involves dividing it into different parts that can then be composed into a form described using the DSL. We call these parts *instance types* (or simply **types** for short). Each part implements a subset of the program's behavior that is related to a specific feature. For example, a *back-end* instance type includes the parts of a program that implement back-end behavior.

Types are then ② **instantiated** and named using the DSL to form *instances*. For example, a back-end instance *type* could be instantiated multiple times to form separate back-end *instances* for load-balancing. For another example, an application can implement fail-over between distributed replicas that instantiate the same type—this is expanded into a detailed use-case description later.

A type is instantiated, configured and connected using the DSL to form different architectures. The process of forming types depends on the feature-size granularity that is being sought. Our approach to apply C-Saw to third-party programs involved forming DSL expressions based on the coarse typifying of the programs to capture features that were described in §2. Our evaluation shows that even a coarse typification can accommodate (1) different adaptations to a program's architecture once C-Saw is used, and that (2) the same architectural description can be *reused* in different applications.

Types have one or more *junctions*: points in the control flow in which the instance type evaluates a DSL expression. Junctions are used to structure the coordination between instances.

Fig. 3 shows an example of how the architecture is made explicit for an abstract sequential program "$H_1; H_2$" and includes the C-Saw concepts that were described so far. Instances can only communicate with each other through their type's junctions. For example, in Fig. 3 instances f and g (representing front-end and back-end instances respectively) can only send messages to a back-end instance through the junctions—and specifically by using the 'write', 'assert', and 'retract' statements in that example. This will be further illustrated using later examples such as Fig. 4 and Fig. 5.

Junctions and DSL expressions are **embedded** in the software's programming language. Introducing junctions into a program's source code involves inserting calls to C-Saw's API binding in the program's source code. The choice of where to introduce junctions, and how many to introduce, depends on the generality of the architecture sought: introducing more junctions creates more opportunities for architecture specification using DSL expressions.

Each junction has its unique name, a DSL expression, and a key-value (KV) table that stores state. The junction's behavior can be conditional on the table's contents. Instances communicate by making changes to each others' KV table when evaluating DSL expressions in junctions. Junctions can update their tables and those of other junctions through the evaluation of DSL expressions. All instances of a type share the same junction behavior but have a their own copy of the KV table. Using an analogy from Object-Oriented Programming, instance types are like classes and instances are like objects, but C-Saw does not support an inheritance hierarchy.

Instance types may have an arbitrary number of junctions, and each junction may communicate with an arbitrary number of other instances. Examples in the next sections will help make this clearer. Step ③ in Fig. 2 shows I_x having two junctions, J1 and J2, through which it exchanges KV updates with I_y and I_z respectively.

As will be shown in later examples such as Fig. 4 and Fig. 5, in addition to the architecture's structure and the communication between instances, DSL expressions also describe architecture-related logic that implements synchronization of KV-entries between instances, time-outs, retries, and fan-out and fan-in behavior. The DSL's constrained expressiveness makes it easier and safer to change its code than the general programming code in which it is embedded. The DSL supports limited recursion and it is not Turing complete.

Architecting software using C-Saw Software architecture is realized using C-Saw through the ③ **composition** of instances using the C-Saw DSL, by defining the behavior of junctions.

Later, a typification can be ⑤ **changed** to support different junction behavior or granularity. For the C-Saw prototype, we evaluate the changing of DSL expressions in an application modified to use C-Saw without changing its typification, and the reuse of DSL expressions across applications.

Running software composed using C-Saw In addition to the modifications to the program's source code to use C-Saw, the program's compilation is changed to link with a runtime system that interconnects C-Saw instances. The C-Saw prototype implementation uses libcompart [44]: a lightweight, portable runtime that provides channel abstractions for communication between instances. Its channels wrap OS-provided IPC, including TCP sockets and pipes. Each instance executes on the runtime. The runtime controls how instances interact with each other and undertakes message-passing between them, under the control of DSL expressions. Messages may contain serialized application data or control messages, as described in the next section. Serialization support for C data is described in §9.

④ **Running** a program whose architecture is described using C-Saw involves starting a special instance that computes the "main" function. In turn, this can start other instances that form the program's architecture, as we saw in Fig. 3.

```
InstanceTypes = $\{\tau_f,\ \tau_g\}$
Instances = $\{f : \tau_f,\ g : \tau_g\}$
def main() ◀ start $f(g)$ + start $g(f)$
def $\tau_f$ :: $junction(\overline{g})$ ◀ ❶
  | init prop $\neg$Work
  | init data $n$
  $\lfloor H_1 \rceil$; save(..., $n$);
  write($n$, $\overline{g}$);
  assert $[\overline{g}]$ Work; ❷
  wait [] $\neg$Work; ❸
def $\tau_g$ :: $junction(\overline{f})$ ◀
  | init prop $\neg$Work
  | init data $n$
  | guard Work ❹
  restore($n$, ...);
  $\lfloor H_2 \rceil$;
  retract $[\overline{f}]$ Work; ❺
```

Figure 3: Example in C-Saw DSL that typifies the program '$H_1; H_2$' into τ_f (instantiated as f) and τ_g (instantiated as g), and showing the special function 'main'. Each '**def**' in this example is a *junction* whose body is embedded in the host programming language. Taken together, these definitions describe the architecture of the program. A detailed explanation of this example's syntax is given in §4.

4 Architecture Descriptions in C-Saw

The DSL is designed to describe a broad set of architectures that meet various needs, including those in Fig. 1. This section outlines the language, which will be described in more detail in §6 after we present illustrative use-cases.

The example in Fig. 3 will be used to introduce the DSL. The DSL describes a set of running *instances* arranged into a topology that forms the software's architecture. Instances are derived from programs as described in §3 and are declared in the set Instances. They are given a single type from InstanceTypes, as shown in Fig. 3. Both instances and types are given names by the software architect.

Instances communicate with each other through their *junctions*. Junctions are used to structure concurrent execution on distributed state, and the DSL serves to make their coordination and synchronization explicit. We will encounter examples with multiple junctions, but in Fig. 3, both instances have a single junction, each called '*junction*'. A junction's execution is scheduled by the instance's application logic—e.g., a junction might be called to service a client request. Scheduling assumptions are made explicit using junction *guards*: on line ❶ of Fig. 3, junction '$\tau_f :: junction$' can be scheduled at any time, but line ❹ shows that '$\tau_g :: junction$' may only be scheduled when proposition Work is true.

The Work proposition is used to coordinate between the pair of junctions for rate-limiting, in this case by ensuring that only one instance is executing at a time. Instance f (the only instance of τ_f) asserts the proposition to instance g at ❷—this line updates the KV table of f and g, and is a form of communication between instances. Instance f awaits the retraction of Work at ❸—it awaits another instance to update its KV table. Retraction is done by g at ❺. In §5 we will see how these language features are used to define more complex features like fault-tolerance and redundancy.

Junctions are examples of *definitions*. Definitions start with zero or more *declarations*. These are prefixed by the pipe symbol ("| "). For example, "| **init prop** $\neg$Work" declares the proposition Work and initializes it to false. "| **init data** n" declares a variable n. Variables and propositions form the junction's *state*, and their values are stored in the junction's KV table.

Definitions can reference code from the host language in which the DSL is embedded. We use H to range over host-language statements. The notation $\lfloor H \rceil \{\vec{V}\}$ encloses code in the host programming language and specifies writable state $\vec{V}$. This notation separates application logic from the architecture expression. Only junction state $\vec{V}$ may be *written to* by the host language statement H. Arbitrary junction state may be *read* by H. Fig. 3 abstracts host-language code as $\lfloor H_1 \rceil$ and $\lfloor H_2 \rceil$. Dropping the $\{\ldots\}$ means that neither H_1 nor H_2 may alter their containing junction's KV table.

5 Examples of C-Saw from Use-Cases

This section provides in-depth examples of using the DSL to implement architectural patterns from Fig. 1.

5.1 Remote snapshots

Fig. 4 shows an implementation of example ② from Fig. 1: *one-time* remote snapshots. *Act* and *Aud* are both single-junction instances. *Act* forms part of the application, and *Aud* forms part of the remote logging system, reflecting the distributed architecture of the overall system. The logic mostly consists of a simple extension of Fig. 3.

This architecture can be reused for *continuous* remote snapshots ③ if we repeatedly invoke *Act* and *Aud* during a single execution of the overall system. This would repeatedly capture the *same* variables for logging; if we need to capture different variables during a program's lifetime then we would need additional instances or junctions. A multi-instance architecture example is given next.

5.2 Sharding

We can implement sharding—example ④ in Fig. 1—through an architecture that routes queries to different back-ends according to an N-way partitioned query-space. This architecture could be repurposed to load-balance *computations* across an application rather than load-balance *storage* as is being done here.

Fig. 5 shows an implementation of sharding in the DSL. It features two new behaviors when compared to previous examples: **(i)** the DSL is interacting with the host language to obtain values such as hashes that cannot be computed in the DSL because of its restricted expressiveness, and **(ii)** abstracting over the number of backends, which is configuration parameter external to the DSL.

```
InstanceTypes = {$\tau_{\text{Actual}}$, $\tau_{\text{Auditing}}$}
Instances = {$Act$ : $\tau_{\text{Actual}}$, $Aud$ : $\tau_{\text{Auditing}}$}
def main($\bar{t}$ ❶) ◀ start $Act(\bar{t})$ + start $Aud(\bar{t})$
def $\underline{complain}$() ◀ ...
def $\tau_{\text{Actual}} :: (\bar{t})$ ◀
    | init prop ¬Work
    | init data $n$
    ⌊$H_1$⌉; save(..., $n$);
    ⟨write($n$, $Aud$);
     assert [$Aud$] Work;
     wait [] ¬Work;
    ⟩ ❷ otherwise[$\bar{t}$] $\underline{complain}$();
def $\tau_{\text{Auditing}} :: (\bar{t})$ ◀
    | init prop ¬Work
    | init prop ¬Retried
    | init data $n$
    | guard Work
    restore($n$, ...);
    ⌊$H_2$⌉;
    retract [] Retried; ❹
    case {❸
      Work ⇒
        retract [$Act$] Work otherwise[$\bar{t}$]
          if ¬Retried then assert [] Retried;
          else $\underline{complain}$();
        reconsider ❺
      otherwise ⇒ skip
    }
```

Figure 4: Remote snapshot example. This extends the example from Fig. 3 with failure-awareness and tolerance. The main differences from Fig. 3 are: ❶ accepting a timeout parameter, ❷ using this time-out for failure-awareness and for ❸ retry-based failure-tolerance. The first time this junction is scheduled, line ❹ is redundant since Retried is initialized to false, but line ❹ ensures that Retried is reset each time that position is reached, before the logic that follows it.

Function `Choose()` is defined in the host language and computes which backend to target, by allowing the tgt value to be written back to the DSL. We use the **idx** declaration syntax to allow the DSL to use externally-provided indices. The number N of back-ends is a compile-time configuration parameter. It affects the definition of Instances and line ❶ in Fig. 5.

Note that ⌊`Choose();`⌉{tgt} is sufficiently abstract to implement different types of sharding. The simplest sharding is key-based. Using C-Saw we implemented a Redis extension that provides a more complex, *feature-based* sharding based on object size to improve memory locality. We shard by performing a look-up on a custom table that maps keys to object sizes, and we quantize sizes into disjoint ranges: 0-4KB, 4KB-64KB, and >64KB.

6 Overview of the C-Saw DSL

After the illustrative examples presented in §5, this section describes the syntax and semantics of the C-Saw DSL.

The DSL syntax is summarized in Table 1. To help explain the syntax, we will refer back to earlier examples.

$$
\begin{array}{rcll}
T & ::= & \mathsf{break} & \mid\ \mathsf{next} \\
 & \mid & \mathsf{reconsider} & \\
F & ::= & P & \mid\ \mathsf{false} \\
 & \mid & \neg F & \mid\ F_1 \wedge F_2 \\
 & \mid & F_1 \vee F_2 & \mid\ F_1 \longrightarrow F_2 \\
G & ::= & F & \mid\ \gamma @ F \\
V & ::= & P \ \mid\ S \ \mid\ I & \\
E & ::= & \lfloor H \rceil \{\vec{V}\} & \mid\ \langle E' \rangle \\
 & \mid & \langle\!| E' |\!\rangle & \mid\ \mathsf{return} \\
 & \mid & \mathsf{write}(\gamma,\ n) & \mid\ \mathsf{wait}\ [\vec{n}]\ F \\
 & \mid & \mathsf{save}(n,\ \vec{x}) & \mid\ \mathsf{restore}(\vec{x},\ n) \\
 & \mid & E_1;\ E_2 & \mid\ E_1\ +\ E_2 \\
 & \mid & \|_n\ \vec{E'} & \mid\ E_1\ \mathsf{otherwise}[t]\ E_2 \\
 & \mid & \mathsf{stop}\ \iota & \mid\ \mathsf{start}\ \iota\ \gamma_1(\vec{p}) \ldots \\
 & \mid & \mathsf{assert}\ [\gamma]\ P & \mid\ \mathsf{retract}\ [\gamma]\ P \\
 & \mid & \underline{f}(\vec{p}) & \mid\ \mathbf{verify}\ G \\
 & \mid & \mathsf{skip} & \mid\ \mathsf{retry} \\
 & \mid & \mathsf{case}\ \{ & \\
 & & \quad F\ \Rightarrow\ E';\ T & \\
 & & \quad \vdots \qquad \vdots & \\
 & & \quad \mathsf{otherwise}\ \Rightarrow\ E'' & \\
 & & \} &
\end{array}
$$

Table 1: C-Saw DSL Syntax, described in §6. Symbols: E are expressions; T are terminators used in case branches; F are propositional formulas and G are formulas that can be interpreted relative to a specific junction; P are user-defined propositions like Work in Fig. 3; and V ranges over different kinds of symbols: indices I, sets S, and propositions P.

```
InstanceTypes = $\{\tau_{\text{Front}},\ \tau_{\text{Back}}\}$
Instances = $\{Fnt : \tau_{\text{Front}},\ Bck_1 : \tau_{\text{Back}}, \ldots,\ Bck_N : \tau_{\text{Back}}\}$
def $\tau_{\text{Front}} :: (\overline{t})$ ◀
    | init prop $\neg$Work
    | init data $n$
    | idx tgt of $\{Bck_1, \ldots,\ Bck_N\}$ ❶
    ⌊Choose(); ❷⌉{tgt}; save(..., $n$);
    ⟨write($n$, tgt ❸); assert [tgt] Work; wait [] $\neg$Work⟩
      otherwise[$\overline{t}$] $\underline{complain}$();
```

Figure 5: N-ary sharding over $\{Bck_1, \ldots,\ Bck_N\}$. This example builds on Fig. 4. The definition of τ_{Back} is omitted because it closely follows τ_{Auditing}. The **idx** syntax used at line ❶ to declare an index that can be updated by the host language. This update may occur on line ❷. An index over a set can also be used as a cursor as seen on line ❸, which resolves to Bck_{tgt}.

Notation We use $\vec{\cdot}$ to denote a collection of terms, and we use metavariables ranging over parameters p, instances ι, junctions γ, DSL-defined functions $\underline{f}$, and named data n. Named data is stored in a Key-Value (KV) table that is local to each junction; the name is the key and the data is the value. The DSL is designed to clarify the logic governing the synchronization of these KV tables.

Host↔Junction sharing Junctions share data between their KV table and the host language by using the 'save' and 'restore' primitives to move host data to and from the KV table. Data can be pushed to other junctions' tables using 'write' (see Fig. 3).

Junction↔Junction synchronization and communication Junctions synchronize parts of their KV tables by using the 'write' primitive to push records to another junction, or using 'assert' or 'retract' for propositional symbols. This is done for Work at line ❸ in Fig. 3. The 'wait' primitive blocks until a formula is true. It allows for specific records in the KV table to be updated by another instance.

Composition This includes sequential (;) and parallel (+) composition. The special composition 'otherwise' is used for timed failure-handling. It is heavily used in examples (§5).

Control flow All DSL code is terminating except for calls to host-language code. As described below, the DSL supports loops but they are unrolled at compile time. It also supports limited branching: 'reconsider' (line ❺ in Fig. 4) branches to the containing 'case' expression if a different match is made (i.e., if the propositions checked before the current guard have changed, or the current guard no longer applies) otherwise the expression fails. 'retry' branches back to the beginning of a junction and can only be invoked a fixed number of times within a single scheduling of a junction.

Blocks Code blocks differ in what happens if a failure is encountered in the block. When using a transaction block $\langle\!|\vec{E}|\!\rangle$, a failure results in a clean rollback of the KV table. $\langle\vec{E}\rangle$ does not rollback if E is not executed successfully—whatever changes have been made to the table up to that point will persist. This matters because we get different behavior depending on where we put 'otherwise'.

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 [14] 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 §7.3.

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.10,
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. 10.
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. 10. An example of indices being passed by parametsr is shown in §7.3.

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* [17]: 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. 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 '$\mathsf{for}\ \widetilde{n} \in \vec{N}^m\ \ op\ I[\widetilde{n}]$', 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:

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

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. 10. 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.

7 Additional Architecture Examples

Section 5 provides several examples 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.

7.1 Parallel sharding

```
$\mathsf{InstanceTypes} = \{\tau_{\text{Front}},\ \tau_{\text{Back}}\}$
$\mathsf{Instances} = \{Fnt : \tau_{\text{Front}},\ Bck_1 : \tau_{\text{Back}}, \ldots,\ Bck_N : \tau_{\text{Back}}\}$
def $\tau_{\text{Front}} :: (\tilde{t})$ ◀
  | init prop $\neg$Work
  | init data $n$
  | set Backs # Assigned to $\{Bck_1, \ldots,\ Bck_N\}$ ❶
  | for $\widetilde{tgt} \in$ Backs   init prop $\neg$ActiveBackend$[\widetilde{tgt}]$ ❷
  | subset tgt of Backs ❸
  | init prop $\neg$HaveAtLeastOne
  $\lfloor$Choose();$\rceil$$\{$tgt$\}$; save($\ldots$, $n$);
  retract [] HaveAtLeastOne;
  for $\tilde{b} \in$ tgt  + ❹
    if ActiveBackend$[\tilde{b}]$ then
      $\langle\!|$ write($n$, $\tilde{b}$); assert $[\tilde{b}]$ Work; wait [] $\neg$Work; ❺
         assert [] HaveAtLeastOne; ❻
      $|\!\rangle$ otherwise$[\tilde{t}]$ retract [] ActiveBackend$[\tilde{b}]$;
  # Complain if not one backend is viable.
  if $\neg$HaveAtLeastOne $\underline{complain}$();
```

Figure 6: Snippet of N-ary sharding to a *set* of back-ends. The syntax is explained in §7.1.

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:

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

Extending this idea further, Fig. 6 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. 6 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.

7.2 Caching

Recall that use-case ⑤ from in Fig. 1 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. 7 shows the cache's implementation. Note that τ_{Fun} is closely based on τ_{Auditing} in Fig. 4 $\lfloor F \rceil$ 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.

7.3 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. 8 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. 10 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. 11 for the back-end and Fig. 9 for the front-end. The implementation of the description, client and backend are shown in Fig. 12, Fig. 10 and Fig. 14 respectively.

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 §7.4.

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

Figure 7: Adding an application-specific caching layer. This examples builds on Fig. 4, 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 $\lfloor$CheckCacheable$\rceil$; ❸ the "case" statement is redone but will not reconsider this branch; ❹ performs the lookup using $\lfloor$LookupCache$\rceil$; ❺ call the function if the result cannot be cached, or if the cache misses; ❻ update the cache if the result is cacheable.

7.4 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 §7.3.

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

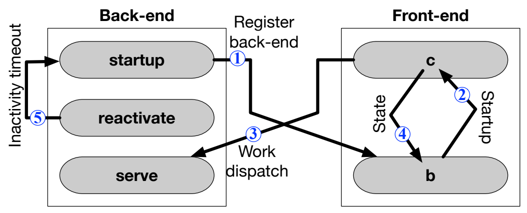


Figure 8: The fail-over architecture described in §7.3 relies on two instance types: back-end and front-end, shown as τ_b and τ_f in our code. They have three and two junctions respectively. This diagram shows important interactions between junctions: ① the junction $\tau_b :: startup$ registers the 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.

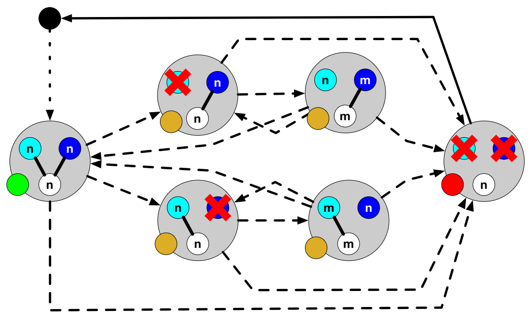


Figure 9: States of the backend-facing junction in the front-end instance. The system is in full capacity (left-most state) if both back-ends are online and synchronized. It fails (right-most state) if both back-ends are unavailable—making fail-over impossible. If at least one back-end is available then 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 $\tau_f :: b$.

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. 15. That diagram reuses the notation introduced in Fig. 11, 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. 15 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.

```
$\textbf{def}\ \tau_f :: b(\overline{backends}, \overline{t})$ ◀
  | $\textbf{init data}\ state$
  | $\textbf{init prop}$ Starting            | $\textbf{init prop}\ \neg$Active
  | $\textbf{init prop}\ \neg$Activating     | $\textbf{init prop}\ \neg$Retried
  | for $\widetilde{tgt} \in \overline{backends}$   $\textbf{init prop}\ \neg$Backend$[\widetilde{tgt}]$❶
  if Starting then
    for $\widetilde{b} \in \overline{backends}$ +
      ⟨wait [] InitBackend$[\widetilde{b}]$ otherwise$[\overline{t}]$ skip⟩;
    retract [] HaveAtLeastOne;
    for $\widetilde{b} \in \overline{backends}$ ;
      if InitBackend$[\widetilde{b}]$ then
        ⟪ $\underline{Initialize}(\widetilde{b})$;
          # Next line relies on idempotence.
          assert [] HaveAtLeastOne;
        ⟫ otherwise$[\overline{t}]$ skip;
    if $\neg$HaveAtLeastOne then $\underline{complain}$;
    retract [] Retried;
    case {
      Starting $\Rightarrow$
        # Progress f::c beyond Starting.
        retract $[f :: c]$ Starting otherwise$[\overline{t}]$
          if $\neg$Retried then
            assert [] Retried;
          else $\underline{complain}()$;
        reconsider
      otherwise $\Rightarrow$ skip
    }
  else
    case {
      Call $\Rightarrow$
        ⟨ $\textbf{verify}\ \neg$Active;
          write$(state,\ f :: c)$;
          assert $[f :: c]$ Active;
          wait $[state]$ $\neg$Active;
        ⟩ otherwise$[\overline{t}]$ $\underline{complain}()$;
        retract [] Call;
        break
      for $\widetilde{b} \in \overline{backends}$  $\neg$Call $\wedge$ InitBackend$[\widetilde{b}]$ $\Rightarrow$
        $\underline{Initialize}(\widetilde{b})$ otherwise$[\overline{t}]$ skip;
        retract [] InitBackend$[\widetilde{b}]$;
        break
      otherwise $\Rightarrow$ skip
    }
```

Figure 10: Code for the backend-facing junction in the front-end instance sketched in Fig. 9. 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.

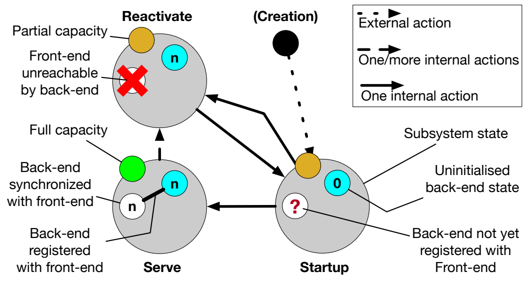

Reactivate
(Creation)
External action
One/more internal actions
One internal action
Partial capacity
Front-end unreachable by back-end
n
Full capacity
Back-end synchronized with front-end
n
n
Back-end registered with front-end
Serve
Subsystem state
Uninitialised back-end state
0
?
Back-end not yet registered with Front-end
Startup

Figure 11: States of a back-end instance. After the instance is created, the $\tau_b :: startup$ junction is scheduled as described in Fig. 8. This either times out, resulting in $\tau_b :: reactivate$ being scheduled, or the instance is subsequently used to serve client requests through schedules of $\tau_b :: serve$. This diagram also explains visual cues used in later diagrams.

```
InstanceTypes = $\{\tau_f,\ \tau_b\}$
Instances = $\{f : \tau_f,\ b_1 : \tau_b,\ b_2 : \tau_b\}$
def main($\overline{t}$) ◀
  start $b_1$ $startup(\overline{t})$ $serve(\overline{t})$ $reactivate(\lfloor 3 * \overline{t} \rceil)+$
  start $b_2$ $startup(\overline{t})$ $serve(\overline{t})$ $reactivate(\lfloor 3 * \overline{t} \rceil)+$
  start $f$ $b(\{b_1 :: serve, b_2 :: serve\}$ ❶, $\overline{t})$
     $c(\{b_1 :: serve, b_2 :: serve\}, \overline{t})$
def $\underline{complain}$ ◀ $\lfloor \ldots \rceil$; return
def $\underline{Initialize}(\overline{tgt})$ ◀ ❷
  verify $\neg$Activating $\wedge$ $\neg$Active;
  write($state$, $\overline{tgt}$);
  assert $[\overline{tgt}]$ Activating; ❸
  wait [] $\neg$Activating;
  assert $[\overline{tgt}]$ 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$[\overline{tgt}]$; ❹
  retract [] Active;
```

Figure 12: Part of the architecture description for the fail-over architecture described in §7.3. $\underline{Initialize}$ 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.

```
**def** $\tau_f :: c(\overline{backends}, \overline{t})$ ◄
  | **init prop** Starting                          | **init prop** $\neg$Active
  | **init prop** $\neg$Req                         | **init prop** $\neg$Call
  | **init prop** $\neg$HaveAtLeastOne
  | **init data** *state*                           | **init data** *req*
  | **init data** *preresp*
  | for $\widetilde{tgt} \in \overline{backends}$  **init prop** $\neg$Backend$[\widetilde{tgt}]$
  | for $\widetilde{tgt} \in \overline{backends}$  **init prop** $\neg$Running$[\widetilde{tgt}]$
  # Req is asserted externally
  # to process client request.
  | **guard** $\neg$Starting $\wedge$ Req
  retract [] Req;
  **verify** $\neg$Call;
  assert $[f :: b]$ Call;
  wait $[state]$ Active;
  restore$(state, \ldots)$;
  retract [] Call;
  $\lfloor H_1 \rceil$;
  save$(\ldots, req)$;

  retract [] HaveAtLeastOne;
  for $\widetilde{b} \in \overline{backends}$ +
    if Backend$[\widetilde{b}]$ then
      ⦃ **verify** $\mathcal{S}(\widetilde{b}) \longrightarrow \widetilde{b}$@Active $\wedge \neg\widetilde{b}$@Running$[\widetilde{b}]$;
        write$(\widetilde{b}, req)$;
        assert $[\widetilde{b}]$ Running$[\widetilde{b}]$;
        wait $[preresp]$ $\neg$Running$[\widetilde{b}]$;
        assert [] HaveAtLeastOne;
      ⦄ otherwise$[\overline{t}]$ retract [] Backend$[\widetilde{b}]$;

  if $\neg$HaveAtLeastOne $\underline{complain}()$;
  **verify** HaveAtLeastOne;

  restore$(preresp, \ldots)$;
  save$(\ldots, state)$;
  write$(f :: b, state)$;
  $\lfloor H_3 \rceil$;
  retract $[f :: b]$ Active;
```

Figure 13: Code for the client-facing front-end junction in the fail-over architecture described in §7.3. The code for the backend-facing front-end junction is shown in the paper.

```
def $\tau_b :: serve(\overline{t})$ ◀
  | init prop $\neg$Active                    | init prop $\neg$Activating
  | init prop $\neg$RecentlyActive            | init data $preresp$
  | init data $state$                         | init data $req$
  | init prop $\neg$Running[me :: junction]
  | guard Activating $\vee$ (Active $\wedge$ Running[me :: junction])
  case {
    Activating $\Rightarrow$
      restore($state$, ...);
      # If the remote retraction fails,
      # then b::reactivate will eventually
      # retry the startup.
      retract [$f :: b$] Activating otherwise[$\overline{t}$]
        retract [] Activating;
      break
    otherwise $\Rightarrow$
      assert [me :: instance :: $reactivate$] RecentlyActive
      restore($req$, ...);
      $\lfloor H_2 \rceil$;
      save(..., $preresp$);
      ⟨ write($f :: c$, $preresp$);
        retract [$f :: c$] Running[me :: junction];
      ⟩ otherwise[$\overline{t}$] retract [] Active
  }

def $\tau_b :: startup(\overline{t})$ ◀
  | init prop $\neg$InitBackend[me :: instance :: $serve$]
  | guard $\neg$me :: instance :: $serve$@Active
  assert [$f :: b$] InitBackend[me :: instance :: $serve$]
    otherwise[$\overline{t}$] skip

def $\tau_b :: reactivate(\overline{t})$ ◀
  | init prop $\neg$RecentlyActive
  | init prop $\neg$Active
  retract [] RecentlyActive;
  wait [] RecentlyActive otherwise[$\overline{t}$]
    ⟨retract [me :: instance :: $serve$] Active;
      retract [me :: instance :: $serve$] Activating⟩;
```

Figure 14: Code for the back-end in the fail-over architecture sketched in §7.3

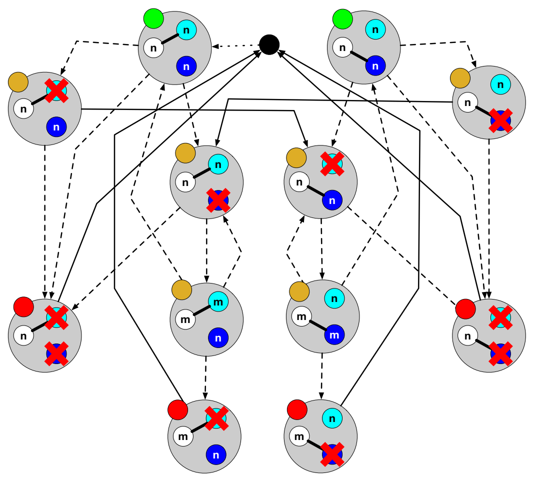


Figure 15: States of the front-end of the “watched” fail-over system described in §7.4.

```
InstanceTypes = $\{\tau_f,\ \tau_w,\ \tau_o,\ \tau_s\}$
Instances = $\{f : \tau_f,\ w : \tau_w,\ o : \tau_o,\ s : \tau_s\}$
def main($\overline{t}$) ◀
  (start $w\ c_o()\ c_s()\ c_{\mathrm{unrecov}}()$ + start $o(\overline{t})$ + start $s(\overline{t})$) ; start $f(\overline{t})$
def $\underline{complain}$ ◀ ...
def $\underline{RunBackend}(n, \overline{t}, \overline{tgt})$ ◀
  ⟨|write($n$, $\overline{tgt}$); assert [$\overline{tgt}$] Run[$\overline{tgt}$]|⟩
    otherwise[$\overline{t}$] $\underline{complain}()$;
def $\tau_f :: (\overline{t})$ ◀
  | init prop ¬Reply
  | for $\widetilde{tgt} \in \{o, s\}$   init prop ¬Run[$\widetilde{tgt}$]
  | init prop ¬failover        | init prop ¬nofailover
  | init data $n$                | init data $m$
  # Junction won't be scheduled until ¬Reply.
  | guard ¬Reply
  $\lfloor H_1 \rceil$; save(..., $n$);
  verify ¬Run[$o$] ∧ ¬Run[$s$] ∧ ¬Reply
  verify ¬ (failover ∧ nofailover)
  case {
    failover ∧ ¬nofailover  ⇒
      $\underline{RunBackend}(n, \overline{t}, s)$;
      break
    ¬failover ∧ nofailover  ⇒
      $\underline{RunBackend}(n, \overline{t}, o)$;
      break
    otherwise  ⇒
      $\underline{RunBackend}(n, \overline{t}, o) + \underline{RunBackend}(n, \overline{t}, s)$
        otherwise[$\overline{t}$] $\underline{complain}()$;
        # Here could implement more robust handling,
        # to retry $\underline{RunBackend}$ () for example.
  };
  # Don't wait too long for completion, prioritize
  # throughput.
  wait [$m$] Reply otherwise[$\overline{t}$] return;
  # If Reply hasn't been reset in line above then this
  # junction won't be scheduled again because of guard.
  retract [] Reply;
  restore($m$, ...);
  $\lfloor H_3 \rceil$;
def $\underline{Watch}(\overline{tgt}, \overline{prop})$ ◀
  | for $\widetilde{tgt} \in \{o, s\}$   init prop ¬Run[$\widetilde{tgt}$]
  | init prop ¬$\overline{prop}$
  ⟨|assert [$\overline{tgt}$] $\overline{prop}$; assert [$f$] $\overline{prop}$|⟩ otherwise $\underline{complain}()$
def $\tau_w :: c_s()$ ◀
  | guard $\neg\mathcal{S}(o) \wedge \mathcal{S}(s) \wedge \mathcal{S}(f)$
  $\underline{Watch}(s, \text{failover})$
def $\tau_w :: c_o()$ ◀
  | guard $\neg\mathcal{S}(s) \wedge \mathcal{S}(o) \wedge \mathcal{S}(f)$
  $\underline{Watch}(o, \text{nofailover})$
def $\tau_w :: c_{\mathrm{unrecov}}()$ ◀
  | guard $\neg\mathcal{S}(s) \wedge \neg\mathcal{S}(o) \vee \neg\mathcal{S}(f)$
  $\underline{complain}()$
```

Figure 16: First half of the code for §7.4. Note the proposition name being passed as the second parameter to the function $\underline{Watch}$; it must be resolvable at compile-time since functions behave as templates in this language.

```
def $\underline{reply}(\overline{t}, \overline{other})$ ◀
  verify $\neg$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 $\overline{\text{Reply}}$ mode.
  verify $\neg$Reply@$\overline{other}$
  ⟨save(…, $m$);
   write($m$, $f$);
   assert [$f$] Reply;
  ⟩ otherwise[$\overline{t}$] $\underline{complain}()$;
def $\tau_s :: (\overline{t})$ ◀
  | for $\widetilde{tgt} \in \{s\}$   init prop $\neg$Run[$\widetilde{tgt}$]
  | init prop $\neg$Reply
  | init data $n$       | init data $m$
  | guard Run[$s$]
  verify $\neg$Reply
  restore(…, $n$);
  $\lfloor H_2 \rceil$;
  retract [$f$] Run[$s$];
    otherwise[$\overline{t}$] $\underline{complain}()$;
  case {
    failover ⇒
      $\underline{reply}(\overline{t}, o)$;
      retract [] Reply;
      break;
    otherwise ⇒ ⌊skip⌉
  };
def $\tau_o :: (\overline{t})$ ◀
  | for $\widetilde{tgt} \in \{o\}$   init prop $\neg$Run[$\widetilde{tgt}$]
  | init prop $\neg$Reply
  | init data $n$       | init data $m$
  | guard Run[$o$]
  verify $\neg$Reply
  restore(…, $n$);
  $\lfloor H_2 \rceil$;
  retract [$f$] Run[$o$];
    otherwise[$\overline{t}$] $\underline{complain}()$;
  $\underline{reply}(\overline{t}, s)$;
  retract [] Reply
```

Figure 17: Second half of the code for §7.4.

$$\begin{tikzcd}
 & \boxed{\textbf{Unsched}\ f} & & & \boxed{\textbf{Unsched}\ g} & \\
 & \mathsf{Rd}_f(\text{Work},\ \mathtt{ff}) \arrow[u] & & \mathsf{Wr}_f(\text{Work},\ \mathtt{ff}) \arrow[ll, dotted] \arrow[ur] & & \\
\mathsf{Wr}_f(\text{Work},\ \mathtt{tt}) \arrow[ur] & & \mathsf{Wr}_g(\text{Work},\ \mathtt{tt}) \arrow[ul] \arrow[dddrr, dotted] & & & \mathsf{Wr}_g(\text{Work},\ \mathtt{ff}) \arrow[uul] \\
 & \mathsf{Wr}_g(n,\ *) \arrow[ul] \arrow[ur] \arrow[drrr, dotted] & & & \boxed{\textbf{Sched}\ g} \arrow[uul] \arrow[ur] & \\
 & \mathsf{Wr}_f(n,\ *) \arrow[u] & & & \mathsf{Rd}_g(n,\ *) \arrow[u] & \\
 & \boxed{\textbf{Sched}\ f} \arrow[u] & & & \mathsf{Rd}_g(\text{Work},\ \mathtt{tt}) \arrow[u] &
\end{tikzcd}$$

Figure 18: Part of the event structure for Fig. 3. All arrows are enablement arrows, but arrows are dotted to emphasize cross-junction enablement. Scheduling events are shown boxed for emphasis.

$$[\![\lfloor \ldots \rceil \{\vec{V}\}]\!]_J = \left(\bigcup_{v \in \vec{V}} \{\mathsf{Wr}_J(v,\ *)\},\ \emptyset,\ \emptyset \right) \qquad [\![\mathsf{save}(\ldots,\ n)]\!]_J =$$

$$(\{\mathsf{Wr}_J(n,\ *)\},\ \emptyset,\ \emptyset) \qquad [\![\mathsf{write}(\gamma,\ n)]\!]_J = (\{\mathsf{Wr}_\gamma(n,\ *)\},\ \emptyset,\ \emptyset)$$

$$[\![E_1\ +\ E_2]\!]_J = (S[\![E_1]\!]_J \cup S[\![E_2]\!]_J,\ \leq[\![E_1]\!]_J \cup \leq[\![E_2]\!]_J,\ \#[\![E_1]\!]_J \cup \#[\![E_2]\!]_J)$$

Figure 19: Semantic rules for part of the language syntax—the syntax is shown in Table 1.

8 Semantics

This section uses event structures [49] 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 [15], and it seemed like a suitable approach to use for C-Saw.

Fig. 19 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 8.5 contains the rest of the rules.

We take advantage of the graphical notation of event structures to give examples of system behavior. Fig. 18 represents the system from Fig. 3. 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 8.6 contains larger examples based on another example of a DSL expression.

"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.

8.1 Event structures

This section starts by outlining the basic definitions of event structures [49] 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 §8.2. "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. 19, and it will be used in §8.5 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:

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

8.2 Labels

Labels represent the activity taking place during an event. Examples of labels were previously given in §8, and in this section we describe the remaining labels that are used in C-Saw's semantics.

The full set of labels is:

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

Further to the labels described in §8, $\mathsf{Synch}_J(\vec{K})$ 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 §8.5.

The examples before abstract some behavior, such as the $\underline{complain}$ () function:

$$\mathbf{def}\ \underline{complain}()\ \blacktriangleleft\ \ldots$$

in Fig. 4. We represent this abstracted behavior using ad hoc labels such as the "complain" label in §8.6.

8.2.1 Graphical notation

Event structures can be represented graphically as shown in Fig. 18. This section describes the notation more accurately. A larger example will be given in §8.6.

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: " $L_1 \rightarrow L_2$ " iff, taking e_i to correspond with L_i: $e_1 \lneq e_2$ and $\neg\exists e'.\ e_1 \lneq e' \wedge e' \lneq e_2$.

The second is *minimal conflict*, represented by a zizag between two events. This captures a minimal form of conflict: " $L_1 \sim L_2$ " 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:

$$\begin{tikzcd}
L_1 \arrow[dr] & \\
& L_3 \\
L_2 \arrow[ur] &
\end{tikzcd}$$

Fan-out events create parallel chains of event execution:

$$\begin{tikzcd}
& L_2 \\
L_1 \arrow[ur] \arrow[dr] & \\
& L_3
\end{tikzcd}$$

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

$$\begin{tikzcd}
& L_2 \arrow[dd, squiggly, no head] \\
L_1 \arrow[ur] \arrow[dr] & \\
& L_3
\end{tikzcd}$$

8.3 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 $\mathsf{otherwise}$.

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

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:

$$\llbracket \overset{\Rightarrow}{E} \rrbracket = \begin{cases} S & \text{if "}\leq\text{"} = \emptyset \\ \{e \in S \mid \nexists e'.\ e \leq e'\} & \text{otherwise} \end{cases}$$

$$\llbracket \overset{\Leftarrow}{E} \rrbracket = \begin{cases} S & \text{if "}\leq\text{"} = \emptyset \\ \{e \in S \mid \nexists e'.\ e' \leq e\} & \text{otherwise} \end{cases}$$

To make fresh copies of event structures we use a map $\natural(\text{idx}, \llbracket E \rrbracket)$ 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 $e \in S\llbracket E \rrbracket$, we define $\natural_{\text{idx}} e$ to be the unique bijection to $S\natural(\text{idx}, \llbracket E \rrbracket)$. 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.

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

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 [28] (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:

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

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:

$$\begin{tikzcd}
\mathsf{Synch}_J \arrow[r] \arrow[dd, squiggly, no head] & \cdots \\
 & \mathsf{Rd}_J(P,\ \mathsf{tt}) \\
\mathsf{Synch}_J \arrow[ur] \arrow[r] \arrow[dr] & \mathsf{Rd}_J(Q,\ \mathsf{ff}) \\
 & \cdots
\end{tikzcd}$$

8.4 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).
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 §8.5.
4. A post-processing step described in §8.5 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 §8.6.

$$\begin{tikzcd}
& \mathsf{Start}_{init}(Act) \arrow[r] \arrow[dr] & \mathsf{Wr}_{Aud}(\text{Work},\ \mathit{ff}) \\
\mathsf{main} \arrow[ur] \arrow[dr] & & \mathsf{Wr}_{Aud}(\text{Retried},\ \mathit{ff}) \\
& \mathsf{Start}_{init}(Aud) \arrow[r] & \mathsf{Wr}_{Act}(\text{Work},\ \mathit{ff})
\end{tikzcd}$$

8.5 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. 20 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:

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

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:

$$\mathsf{case}(i) = \begin{cases} \begin{tikzcd} \llbracket E_i;\ T_i \rrbracket^{\eta'_i} & \mathsf{case}(i+1) \\ \llbracket F_i \rrbracket^{\eta'_i} \arrow[u] \arrow[r, squiggly, no head] & \llbracket \neg F_i \rrbracket^{\eta'_i} \arrow[u] \end{tikzcd} & \text{if } i < n \\ \llbracket E_n \rrbracket^{\eta'_n} & \text{if } i = n \end{cases}$$

Finally, $\llbracket E \rrbracket^{\eta} = \mathsf{case}(0)$,

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:

$$\begin{tikzcd} A_1 \arrow[dr] & & B_1 \\ \vdots & \mathsf{Wait}_J(\vec{n}, F) \arrow[ur] \arrow[dr] & \vdots \\ A_n \arrow[ur] & & B_m \end{tikzcd}$$

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 $\vec{n}$.

$$\begin{tikzcd} & \mathsf{DNF}(F)_1 \arrow[r] \arrow[ddr] \arrow[dddddddd, squiggly, no head, shift left] \arrow[dddddddd, squiggly, no head, shift right] & \mathsf{Rd}_J(n_1,\ *) \arrow[r] \arrow[ddr] & B_1 \\ & & \vdots & \vdots \\ A_1 \arrow[uur] \arrow[ddddddr] & & \mathsf{Rd}_J(n_p,\ *) \arrow[r] \arrow[uur] & B_m \\ & & \iddots & \\ \vdots & \vdots & & \\ & & \ddots & \\ A_n \arrow[uuuuuur] \arrow[ddr] & & \mathsf{Rd}_J(n_1,\ *) \arrow[r] \arrow[ddr] & B_1 \\ & & \vdots & \vdots \\ & \mathsf{DNF}(F)_q \arrow[r] \arrow[uur] & \mathsf{Rd}_J(n_p,\ *) \arrow[r] \arrow[uur] & B_m \end{tikzcd}$$

8.6 Example

This section uses the graphical notation described in §8.2.1 to illustrate the event structure for the example described in §5.1. The start-up behavior of this example was shown in §8.4.

There are two instances in this example. The behavior of *Act* is shown next, and that of *Aud* is shown in Fig. 22.

The instances interact implicitly by updating propositions in each other's KV-tables. *Act* engages *Aud* at the occurrence of event $\mathsf{Wr}_{\{Act,Aud\}}(\text{Work},\ \mathtt{tt})$, and is engaged back when $\mathsf{Rd}_{Act}(\text{Work},\ \mathtt{ff})$

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

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

$$\llbracket \mathsf{return}\rrbracket_J = \llbracket \eta(\mathsf{return})\rrbracket_J \qquad \llbracket \mathsf{start}\ \iota\rrbracket_J = (\{\mathsf{Start}_J(\iota)\},\ \emptyset,\ \emptyset) \qquad \llbracket \mathsf{stop}\ \iota\rrbracket_J = (\{\mathsf{Stop}_J(\iota)\},\ \emptyset,\ \emptyset)$$

$$\llbracket E_1;\ E_2\rrbracket^\eta_J = \left(S\llbracket E_1\rrbracket_J^{\eta\{\mathsf{sub}\mapsto E_2\}} \cup S\llbracket E_2\rrbracket^\eta_J,\quad \leq\llbracket E_1\rrbracket_J \cup \leq\llbracket E_2\rrbracket_J \cup \bigcup_{e_1\in S\llbracket \overset{\Rightarrow}{E_1}\rrbracket} \left\{(e_1, e_2) \mid e_2 \in S\ \llbracket \overset{\Leftarrow}{E_2}\rrbracket\right\},\quad \#\llbracket E_1\rrbracket_J \cup \#\llbracket E_2\rrbracket_J \right)$$

$$\begin{aligned}\llbracket E_1 \parallel E_2\rrbracket_J = \Big(& S\llbracket E_1\rrbracket_J \cup S\llbracket E_2\rrbracket_J \cup \textstyle\bigcup_{e\in S\llbracket E_1\rrbracket}\{\natural e\} \cup \bigcup_{e\in S\llbracket E_2\rrbracket}\{\natural e\},\\ & \leq\llbracket E_1\rrbracket_J \cup \leq\llbracket E_2\rrbracket_J \cup \\ &\quad \left\{(e, \natural e' \mid e \in S\ \llbracket \overset{\Rightarrow}{E_1}\rrbracket,\ e' \in S\llbracket E_2\rrbracket\right\} \cup \left\{(e, \natural e' \mid e \in S\ \llbracket \overset{\Rightarrow}{E_2}\rrbracket,\ e' \in S\llbracket E_1\rrbracket\right\} \cup \\ &\quad \left\{(e, \natural e \mid e \in S\llbracket E_1\rrbracket \setminus S\ \llbracket \overset{\Rightarrow}{E_1}\rrbracket\right\} \cup \left\{(e, \natural e \mid e \in S\llbracket E_2\rrbracket \setminus S\ \llbracket \overset{\Rightarrow}{E_2}\rrbracket\right\},\\ & \#\llbracket E_1\rrbracket_J \cup \#\llbracket E_2\rrbracket_J \cup \{(e_2, \natural e_1 \mid e_2 \in S\llbracket E_1\rrbracket,\ e_1 \lneq e_2\} \cup \{(e_2, \natural e_1 \mid e_2 \in S\llbracket E_2\rrbracket,\ e_1 \lneq e_2\}\Big)\end{aligned}$$

$$\begin{aligned}\llbracket E_1\ \mathsf{otherwise}\ E_2\rrbracket_J = \Big(& \text{isolate}\,[S\llbracket E_1\rrbracket_J] \cup \textstyle\bigcup_{e\in S\llbracket E_1\rrbracket}\{S\natural(e, \llbracket E_2\rrbracket)\},\\ & \leq\llbracket E_1\rrbracket_J \cup \textstyle\bigcup_{e\in S\llbracket E_1\rrbracket}\{\leq\natural(e, \llbracket E_2\rrbracket)\} \cup \bigcup_{e\in S\llbracket E_1\rrbracket}\{(e', e'') \mid e'' \in \overset{\Leftarrow}{\natural}\ (e, \llbracket E_2\rrbracket),\ e' \lneq e\},\\ & \#\llbracket E_1\rrbracket_J \cup \textstyle\bigcup_{e\in S\llbracket E_1\rrbracket}\{\#\natural(e, \llbracket E_2\rrbracket)\} \cup \bigcup_{e\in S\llbracket E_1\rrbracket}\{(e, e') \mid e' \in \overset{\Leftarrow}{\natural}\ (e, \llbracket E_2\rrbracket)\}\Big)\end{aligned}$$

$$\llbracket \mathsf{reconsider}\rrbracket_J = \llbracket \eta(\mathsf{reconsider})\rrbracket_J \qquad \llbracket \mathsf{retry}\rrbracket_J = \llbracket J\rrbracket_J \qquad \llbracket \mathsf{next}\rrbracket_J = \llbracket \eta(\mathsf{next})\rrbracket_J \qquad \llbracket \mathsf{break}\rrbracket_J = \llbracket \eta(\mathsf{break})\rrbracket_J$$

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

Figure 20: Semantics of DSL statements, continuing from Fig. 19.

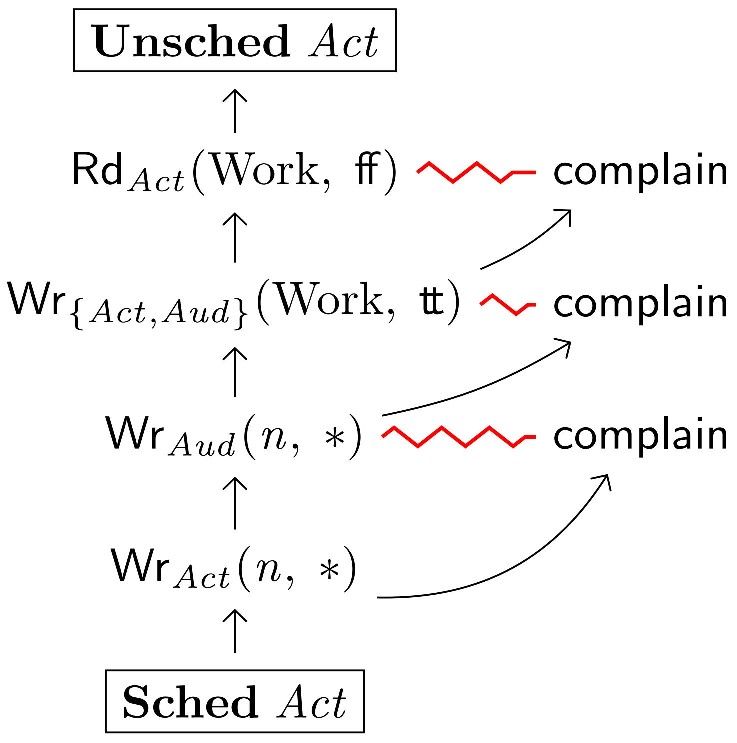


Figure 21: Event structure for *Act* .

occurs. The complexity of *Aud* 's behavior in Fig. 22 arises from the combination of τ_{Auditing}'s retry logic and its failure-handling.

8.7 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:

- $\mathsf{Instances}$ (see §4)
- 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 "$\mathsf{assert}\ [\gamma']\ P$" "$\mathsf{retract}\ [\gamma']\ P$" and "$\mathsf{write}(\gamma',\ n)$" would return the set $\{\gamma'\}$; "$\langle E' \rangle$" evaluates to $\text{Topo}_\gamma(E')$; and "$E_1;\ E_2$" evaluates to $\text{Topo}_\gamma(E_1) \cup \text{Topo}_\gamma(E_2)$.

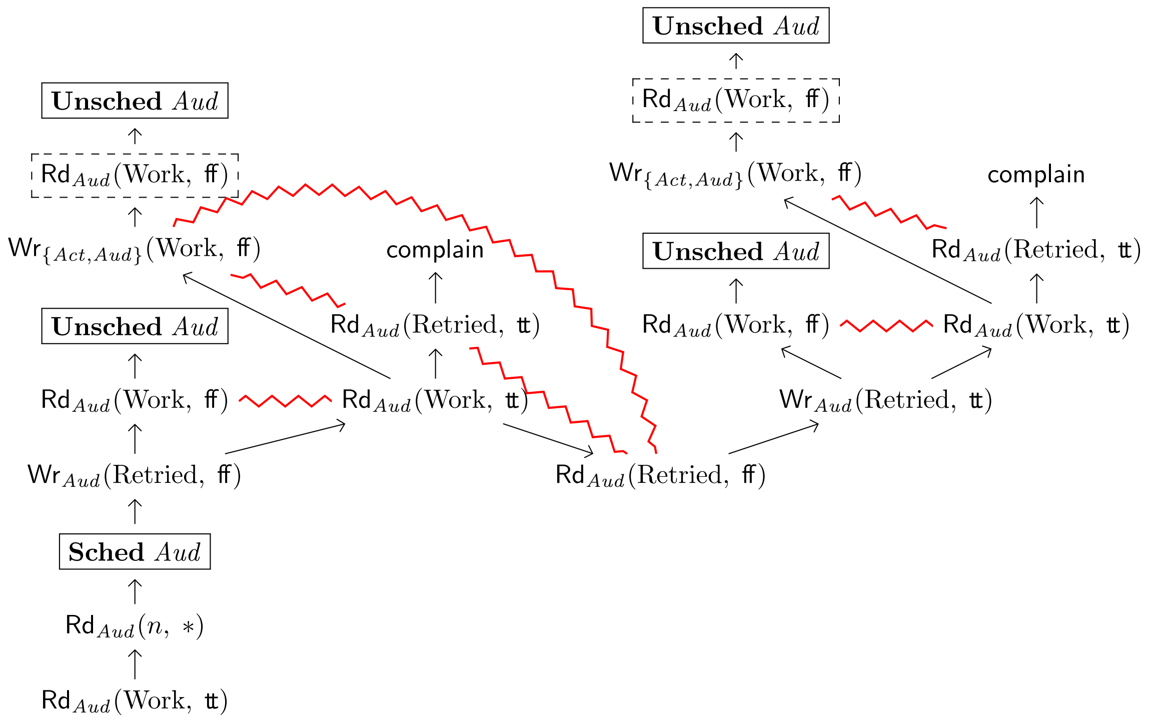


Figure 22: Behavior of the *Aud* instance from §5.1.

9 Serialization Framework

Data-structure serialization is an important supporting primitive for C-Saw's definitional approach to software architecture. Different architectures might require program data to flow differently across instances, and serialization provides the means to capture that data. In languages like C, it is difficult to serialize data structures using existing tools without either (i) limiting the types of data structures that can be expressed or (ii) burdening the programmer with custom serialization of data types. The challenges encountered when working with C datatypes include: a) void pointers which can represent any data type, b) arbitrary casting, and c) implicit size of memory objects that are managed by C's standard library allocator or by a custom allocator.

Various solutions have been devised over the years, including new DSLs [3] intended for implementing RPC, template-based schemes [1] for general types but requiring programmer effort to use their API, techniques to ensure memory safety [41], and specialized approaches for datatypes used in network protocols [12].

C-Saw builds on the C-strider [42] approach for the C language. C-strider implements a type-aware traversal of specific heap objects at runtime, and is guided by user-defined callbacks. It statically analyzes the source code to generate information about each type and generates serialization calls for each field of a type.

Unlike C-strider, C-Saw avoids having the programmer modify their source code or definitions—instead, they `#include` automatically-generated definitions by the C-Saw serializer. This serializer consists of a new libclang-based tool that analyzes C datatype definitions. To use the C-Saw serialization tool, the user specifies the type to serialize, answers some size-related questions if required by the tool, and the serialization file is produced. This tool sacrifices some flexibility but it has been sufficient for the complex datatypes involved in the third-party software that was used to evaluate C-Saw. Though the approach is general, our prototype only supports recursive datatypes up to a maximum, though configurable, recursion depth. For instance, linked lists are only serialized up to a maximum length. Though this might seem like a limitation, it protects against overflowing the serialization buffer. Supporting more flexible serialization through runtime analysis and buffer-resizing is left as future work.

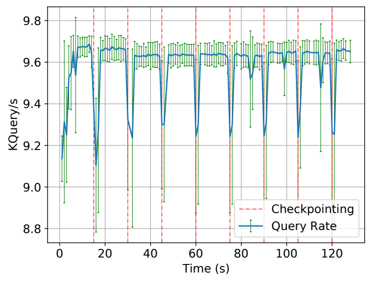
10 Evaluation

We evaluate the **behavior of features implemented in C-Saw** (§10.1) by using reference workloads to measure the effect of DSL-implemented features (from Fig. 1) on performance and reliability. We evaluate **DSL cost and benefit** (§10.2) by measuring the effort of using the DSL when compared to using the host programming language directly; and the **performance overhead** (§10.3) of a C-Saw-based system compared to the original version.

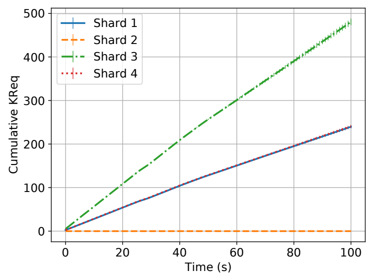
These experiments target Redis v2.0.2, cURL v7.72.0-DEV, and Suricata v6.0.3. All these experiments were carried out on Ubuntu 16.04.7 LTS under Linux kernel version 4.4.0-198-generic, on an Intel i7-3770 CPU machine clocked at 3.4 GHz and with 8GB RAM. Experiments ran in separate VMs allocated 1GB RAM. Experiments were repeated 20 times and averaged and reported with their standard deviation, except for the cumulative distribution function (CDF) data which we obtained directly from `redis-benchmark`.

10.1 Behavior of features implemented in C-Saw

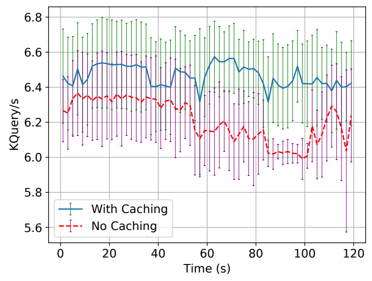
This section evaluates different, newly-added features to Redis and Suricata. Deployments usually build features *around* Redis and Suricata, but in this work we *internalize* important features (Checkpointing, Sharding, Caching) by using the DSL. For Redis we generated workloads by using `redis-benchmark` using its default parameters. For Suricata we used `bigFlows.pcap`, a public packet-capture benchmark that contains several flows from different applications [6].



(a) Response of Query Rate to Checkpoints.

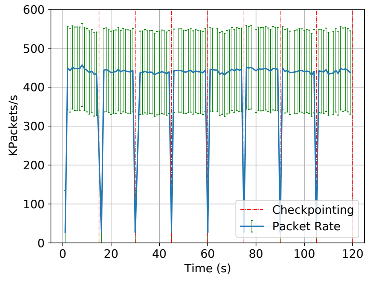


(b) Cumulative requests sharded by key.

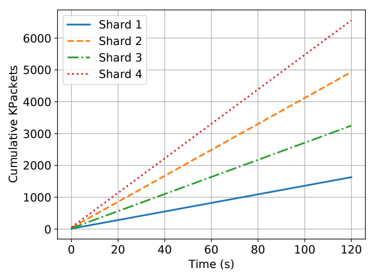


(c) Effect of Caching on Query Rate.

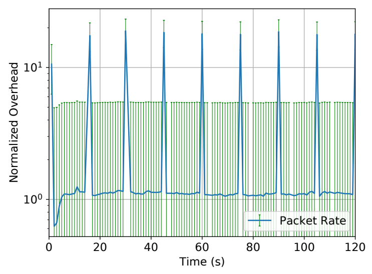
Figure 23: Behavior of Redis reconfigurations. All three graphs show averaged results, and the bars show standard deviation.



(a) Response of Packet Rate to Checkpoints.

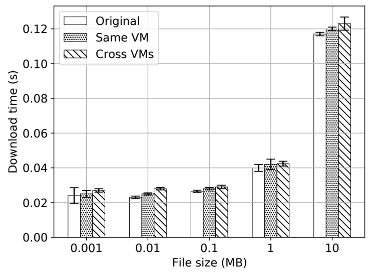


(b) Cumulative requests sharded by 5-tuple.

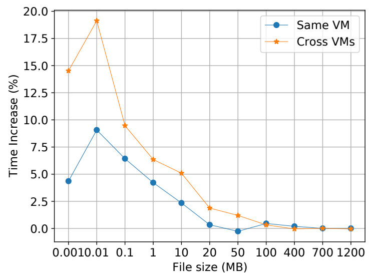


(c) Checkpointing Overhead.

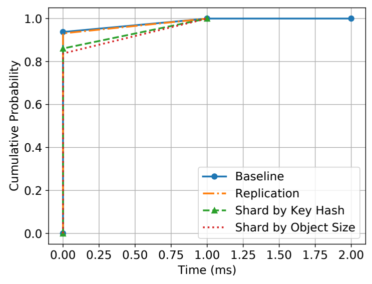
Figure 24: Behavior of Suricata reconfigurations and normalized performance overhead of checkpointing.



(a) cURL performance (averaged). Bars indicate standard deviation.



(b) cURL overhead as percentage. Uses same data as Fig. 25a.

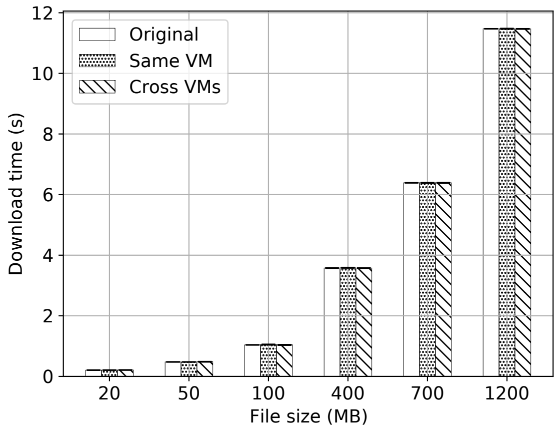


(c) Redis performance overhead (GET).
'Baseline' is unmodified Redis.

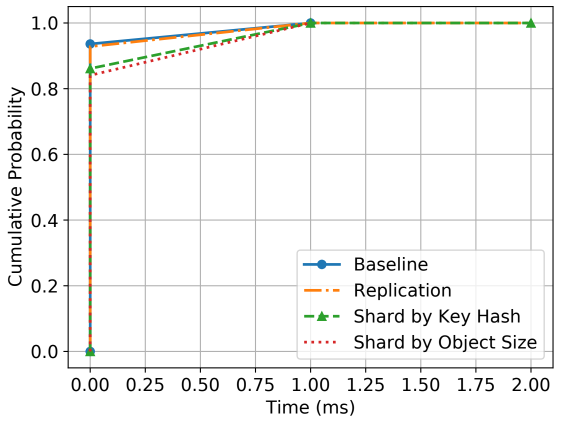
Figure 25: Overhead graphs for rearchitected software.

Checkpointing Checkpointing is a building block for migration and roll-back of state. Fig. 23a shows the results for checkpointing Redis. Redis itself has a default crash handler but it does not checkpoint at intervals. While this handler improves availability, it does not minimize data loss. In this experiment we carry out checkpoints at 15-second intervals and simulate a Redis crash to observe its recovery. A crash is indicated by the vertical red line in the graph. Note that the y-axis starts at 8.0 to show the fine detail of the behavior—the dips in the graph are actually more subtle if we start the y-axis at 0. The same checkpointing logic was used in Suricata and the result is shown in Fig. 24a.

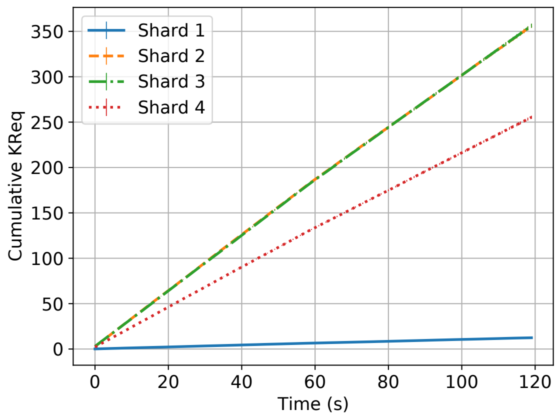
Sharding We extended Redis using the DSL to implement two types of sharding, based on i) key and ii) object-size [23]. In both cases we sharded data into four classes, where each class is serviced by a separate back-end Redis instance. We subjected both types of sharding to even and uneven workloads. Uneven workloads place different pressure on different back-ends. Fig. 23b shows the results for sharding by key, which we hash using the djb2 hashing algorithm [51]. The graph shows the uneven behavior resulting from this workload; we confirmed that the ratio between shards matches that of the workload. The key-based sharding logic was adapted to implement packet-steering in Suricata, with the result shown in Fig. 24b. The 5-tuple of each packet (source and destination IP and port, and protocol) is hashed to determine which of four back-end Suricata instances should process it. The graph shows that the workload is distributed in ratios across the four instances.



(a) Performance of modified cURL.



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



(c) Redis sharding based on object size.

Figure 26: Additional graphs from experiments to Fig. 23, 24, 25

Caching We modified Redis to internalize a cache that is consulted before Redis' own look-up. We used a read-heavy workload to model a scenario where memory-burdened KV databases face a high skew in requests, modeling real-world scenarios [11, 50]. In our scenario, 90% of requests are directed at 10% of the entries. Fig. 23c shows response to this workload. Note that the y-axis starts at 5.6 to show the fine detail of the behavior; the gain from caching on this setup is around 200 queries per second (QPS).

10.2 DSL cost and benefit

We measure cost and benefit by using lines of code (LoC) as a proxy metric for programmer effort. The Suricata and Redis codebases are at a comparable level of difficulty to understand—both are professional projects that are battle-tested by real-world usage. Since the DSL code is embedded as a C library, we give each LoC of DSL code the same weight as a LoC of C code for simplicity, although DSL code is arguably simpler than C syntax since it lacks various operators, unbounded loops, and pointers. The *cost* of using the DSL measures the code changes required to create instances and embed junctions. The *benefit* measures the LoC saved when extending an application by using the DSL compared to using C directly.

	Lines Of Code (LoC) of C syntax			
Feature	DSL in C	Redis(DSL)	Suricata(DSL)	Redis(C)
Checkpointing	79	7	44	332
Sharding	105	1	49	314
Caching	106	6	N/A	306

Table 2: Effort (LoC) needed to support software extensions.

Table 2 shows the LoC needed to support different types of architecture-level features. **DSL in C** refers to the code generated by the DSL-to-C mapping that produces C code that is decoupled from the application-specific logic. Once it is mapped in this way, the code can be used in a junction. **Redis(DSL)** and **Suricata(DSL)** refer to the number of lines edited in the source code to define the junction to host a DSL expression in Redis and Suricata for the different feature types. Defining a junction consists of packaging parameters and calling the "DSL in C" code. For Suricata most of the effort involved creating a new node in Suricata's pipeline that serves as a junction. **Redis(C)** is the LoC needed to rearchitecture directly in C. Redis(C) was developed without knowledge of the DSL, as a control experiment. It includes its own internal management system for communication and synchronization between different instances of Redis, which adds 195 lines to each feature.

Costs The costs of using C-Saw involve typifying software and inserting junctions (§3). In Table 2 this cost is captured by **Redis(DSL)** and **Suricata(DSL)**. In addition to the per-architecture costs shown in Table 2, there is a one-time cost for creating a junction that can be reused for different rearchitectures. For both Redis and Suricata we added a single junction consisting of 98 LoC for Redis and 182 LoC for Suricata. The main function for Redis and Suricata received an additional 5 LoC and 4 LoC respectively to accommodate the junctions.

Benefits The benefits of using C-Saw is three-fold: **i)** DSL expressions can be used across applications, which amortizes the effort of crafting a DSL expression. **ii)** Fewer code changes are needed—we can see this when comparing Redis(C) to the sum of "DSL in C" and Redis(DSL). **iii)** Support for serializing data that is exchanged between instances. The automatically-generated serialization code for the key and value structure used in Redis consists of 182 LoC. The generated serialization code for the packet structure used by Suricata consists of 2380 LoC.

10.3 Performance overhead

Suricata overhead Fig. 24c shows the overhead of the checkpointing reconfiguration of Suricata normalized against the unmodified version. We see that overhead is usually less than 10% and spikes to around 19× during checkpoint-restart-and-resume phases. The performance overhead of the sharding feature is around 60%.

cURL Overhead We changed the architecture of cURL for remote auditing as described in §5. We generated two binaries: for the local and remote instances, respectively. The second binary is intended to receive progress updates from the first in order to audit it. We measured the overhead of executing this system when downloading differently-sized files from a dedicated machine, over 1GbE links on a research testbed. We ran two forms of this experiment: (i) placing both binaries in the same VM, and (ii) placing them in separate VMs to emulate separation between action and audit. Fig. 25b shows the overhead of cURL modified for remote auditing. Fig. 25a is based on the same numbers but presents them in absolute time, and shows standard deviation for more detail. The performance overhead for large files is less intelligible.

Fig. 26a shows the performance of modified cURL when executed over large files, and complements Fig. 25a which focused on small files. The performance difference for large files is less intelligible.

Redis Overhead Fig. 25c shows the response latencies when running GET operations on the original Redis and the three derivatives we developed. The graph shows that the overhead from our modifications are noticeable but low, except for "replication" which involves checkpointing and restarting Redis. While in many cases the average overhead is low, this experiment also features the longest tail latency albeit for a very small percentile. The results for SET are similar.

Fig. 26b shows the performance overhead of various reconfigurations of Redis under a SET workload. It is the complement of the paper's Fig. 25c.

Fig. 26c 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. 23b.

11 Related Work

Existing specification tools for software architecture, such as SysML [7] and arc42 [43], provide standalone descriptions of software architecture. In comparison, C-Saw specifications are embedded *inside* software. It will be interesting future research to look for a synthesis of C-Saw with SysML or arc42.

Split/Merge [40] involves classifying application state into two types depending on the state's scope: general state captures information across a whole application, while local state is scoped to invidual sessions or other units. C-Saw does not impose structure on state, and instead imposes structure on architecture patterns through a powerful DSL and primitives for state management.

C-Saw is inspired by research on process calculi [37] and on coordination languages [9]. Like these frameworks, C-Saw provides primitives for communication and coordination; but in contrast C-Saw provides a terser language that restricts flexibility at runtime. For example, C-Saw does not allow channel creation or passing, or an implicit global tuple-space. Further, it restricts mobility and interactions between processes. These restrictions simplify the runtime's implementation and the provisioning of resources, because they are not considered essential for the class of architectures we surveyed.

12 Conclusion

C-Saw is designed to "separate concerns" [24] between software architecture and its application-specific logic, and it provides a high-level DSL to declaratively compose this logic into an architecture. An architecture can then be reconfigured by editing its DSL expression.

Our main findings are: **(i)** When making them precise through C-Saw's formally-specified DSL, even simple behaviors such as those in Fig. 1 have subtlety and complexity. **(ii)** DSL expressions are reusable, and our prototype reused reconfiguration logic between Redis and Suricata.

Acknowledgments

This work was supported by a Google Research Award and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0106. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of funders.

References

[1] C serialization library. `http://www.happyponyland.net/cserialization/readme.html`.

[2] Open source ids tools: Comparing suricata, snort, bro (zeek). `https://cybersecurity.att.com/blogs/security-essentials/open-source-intrusion-detection-tools-a-quick-overview`.

[3] Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[4] Redis Explained. `https://architecturenotes.co/redis/`.

[5] C-Saw repo. `https://gitlab.com/pitchfork-project`.

[6] Sample Captures. `suricata_sharding_vs_unmodified_diff.pdf`.

[7] OMG Systems Modeling Language. `https://www.omgsysml.org/what-is-sysml.htm`, 2022.

[8] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997.

[9] Farhad Arbab, Marcello M. Bonsangue, and Frank S. de Boer. A Coordination Language for Mobile Components. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 1*, SAC '00, page 166–173, New York, NY, USA, 2000. Association for Computing Machinery.

[10] Alessandro Armando, Gabriele Costa, and Alessio Merlo. Bring Your Own Device, Securely. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1852–1858, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[12] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, Broomfield, CO, October 2014. USENIX Association.

[13] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, April 2016.

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

[15] 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.

[16] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

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

[18] Redis contributors. Partitioning: how to split data among multiple redis instances. `https://redis.io/topics/partitioning`.

[19] Redis contributors. Redis cluster specification. `https://redis.io/topics/cluster-spec`.

[20] Redis contributors. Redis replication. `https://redis.io/topics/replication`.

[21] Redis contributors. Who's using redis? `https://redis.io/topics/whos-using-redis`.

[22] Bill Curtis, Herb Krasner, and Neil Iscoe. A Field Study of the Software Design Process for Large Systems. *Commun. ACM*, 31(11):1268–1287, November 1988.

[23] Diego Didona and Willy Zwaenepoel. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.

[24] Edsger W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982.

[25] The Open Information Security Foundation. Suricata. `https://suricata.io/`.

[26] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[27] David Garlan. Software Architecture: A Travelogue. In *Future of Software Engineering Proceedings*, FOSE 2014, page 29–39, New York, NY, USA, 2014. Association for Computing Machinery.

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

[29] Gernot Heiser. Virtualizing Embedded Systems: Why Bother? In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 901–905, New York, NY, USA, 2011. Association for Computing Machinery.

[30] Jason I. Hong and James A. Landay. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, page 177–189, New York, NY, USA, 2004. Association for Computing Machinery.

[31] Twitter Inc. twemproxy. `https://github.com/twitter/twemproxy`.

[32] Nick Kew. *The Apache Modules Book: Application Development with Apache*. Prentice Hall Professional, 2007.

[33] Donald E. Knuth. Structured Programming with go to Statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.

[34] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[35] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems. *J. Syst. Softw.*, 80(5):687–699, May 2007.

[36] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at What COST? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 14, USA, 2015. USENIX Association.

[37] Robin Milner. *Communicating and mobile systems: the pi calculus.* Cambridge University Press, 1999.

[38] Sape J. Mullender. Distributed Operating Systems. *ACM Comput. Surv.*, 28(1):225–227, March 1996.

[39] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, October 1992.

[40] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 227–240, USA, 2013. USENIX Association.

[41] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, Santa Clara, CA, August 2019. USENIX Association.

[42] Karla Saur, Michael Hicks, and Jeffrey S. Foster. C-strider: type-aware heap traversal for C. *Software: Practice and Experience*, 46(6):767–788, 2016.

[43] Gernot Starke. Documenting software architecture with arc42. `https://www.innoq.com/en/blog/brief-introduction-to-arc42/`, 2022.

[44] Nik Sultana, Henry Zhu, Ke Zhong, Zhilei Zheng, Ruijie Mao, Digvijaysinh Chauhan, Stephen Carrasquillo, Junyong Zhao, Lei Shi, Nikos Vasilakis, and Boon Thau Loo. Towards Practical Application-level Support for Privilege Separation. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*, pages 71–87. ACM, 2022.

[45] https://curl.se/docs/thanks.html The cURL Contributors. Companies using curl in commercial environments. `https://curl.se/docs/companies.html`.

[46] https://curl.se/docs/thanks.html The cURL Contributors. curl. `https://github.com/curl/curl`, 2021.

[47] Craig Walls. *Spring Boot in Action.* Manning Publications Co., USA, 1st edition, 2016.

[48] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, October 2001.

[49] 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.

[50] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[51] Ozan Yigit. Hash functions. `http://www.cse.yorku.ca/~oz/hash.html`.

[52] Donghao Zhou, Zheng Yan, Yulong Fu, and Zhen Yao. A Survey on Network Data Collection. *Journal of Network and Computer Applications*, 116:9–23, 2018.