Toward Bringing Distributed System Design upon Rigorous Footing

Mohammed S. Al-Mahfoudh
School of Computing,
University of Utah,
Salt Lake City, UT 84112, USA
mahfoudh@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing,
University of Utah,
Salt Lake City, UT 84112, USA
ganesh@cs.utah.edu

Ryan Stutsman
School of Computing,
University of Utah,
Salt Lake City, UT 84112, USA
stutsman@cs.utah.edu

***Abstract*—**

The development of distributed systems based on poorly specified abstractions can hinder unambiguous understanding and the creation of common formal analysis methods. In this paper, we outline the design of a system modeling language called DS2, and point out how its primitives are well matched with concerns that naturally arise during distributed system design. We present an operational semantics for DS2 as well as results from an ongoing Scala-based implementation slated to support a variety of state-space exploration techniques. The driving goals of this project are to: (1) provide a prototyping framework within which complex distributed system protocols can be stated and modeled without breaking the primitives down to low level ones, and (2) drive the development of interesting and distributed system-relevant property checking methods (e.g., linearizability).

***Index Terms*—Distributed Systems, Operational Semantics, Scheduling, Concurrency, Actors**

1. Introduction

Distributed systems, both large scale and small, are now critically important in nearly every aspect of computing and in our lives. From embedded microcontrollers to data centers and web applications that span multiple geographic locations, more developers are building distributed systems than ever before. The ability to rapidly construct and deploy correctly functioning distributed systems is a growing necessity; unfortunately, this is hard even for experts. Non-determinism, weak update consistency, and complex failure handling protocols push far beyond what manual reasoning can grapple with, making it hard to offer *basic* safety guarantees.

The formal methods research community has begun responding to some of these challenges associated with building distributed systems. Recent efforts have formalized several famously subtle algorithms [19] and generated correct, synthesized implementations [7], [10]. While these efforts represent significant steps forward, their wider applicability is limited. For example, approaches that rely on automated theorem proving [7] are beyond the expertise of all but a handful of academics and tend to abstract away from situation-specific complexities. Approaches that require developers to model systems using predicate logic [17] or model-checking frameworks [13], [9] provide only limited reasoning capabilities and do not provide the requisite higher-level abstractions necessary for intuitive modeling.

Concurrently with all these developments, new languages and frameworks are continually emerging to help developers rapidly create new distributed systems. Languages such as Go and Rust and frameworks such as Akka are enabling developers to build and deploy distributed systems more easily. However, these approaches provide no formal basis for reasoning about the safety and correctness of the resulting systems. The guarantees that they do provide are informally documented, are non-portable, and low-level. Consequently, the developers' ability to build and deploy complex systems has vastly outpaced their ability to *reason* about their behaviors and detect design flaws. There is however an opportunity here: by combining a simple and expressive programming model with strong formal guarantees, one can achieve the best of both worlds. With that goal in mind, we are developing a domain-specific language for distributed systems called DS2[1].

DS2 is being developed to allow developers specify, test, verify, and synthesize correct distributed systems built on a clear operational semantics. DS2 uses an Actor-based concurrency model [8], [2], [11] that is compatible with the popular Akka distributed systems programming framework [1]. A working prototype of DS2 system written in Scala exists, and many driving examples are being ported using a front-end. Preliminary results from our effort appear in [6]; this paper is the first to offer a formal semantics to DS2. While developing some of the central features underlying state-space exploration of distributed systems expressed in DS2, we realized first-hand how treacherous the corner-cases can be. We detail some of these subtleties in §6. This gave further impetus to our work on writing down a clear operational semantics.

[1]Domain-Specific/Declarative-Specification of Distributed Systems.

The need for clear guiding semantics underlying distributed systems is further exemplified by Chord's [21] erroneous operation [24] that was discovered with help of a manually written model of it in an external model checker. In short, having a semantic basis helps carry out automated semantics-guided verification, supports more objective comparisons between various efforts, and helps prepare the community to handle newer protocols being designed. A variety of formal (e.g., linearizability checking [5]) and semi-formal (e.g., lineage-driven fault-injection [18]) techniques can also be easily supported.

Background: It is important to point some central characteristics of distributed systems, over and above those present in shared-memory concurrency (e.g., PThreads) or traditional message-passing based parallel programming (e.g., using MPI). Distributed systems ingest all these concurrency-related subtleties, and additionally present other challenges:

- Weak (eventual [4]) consistency is the norm rather than the exception. This adds to the non-intuitiveness of distributed systems behaviors.
- Given that faulty operation is the norm rather than the exception, processes of a distributed system hover on different planes of operation at different times (i.e. fault-free or faulty planes). Each time a fault occurs, the correctness guarantees are subsetted to a core set of primitive ones. When faults are correctly handled and normalcy returns, guarantees are elevated. A designer must be empowered with the ability to simulate these fault/recovery driven transitions, and check for the right level of guarantees being delivered.

2. Contributions

A key feature of our approach is the adoption of the simple *strategy* OO design pattern, wherein the scheduler is the algorithm and the distributed system is the context. This design approach provides both flexibility and extensibility both with respect to actual algorithms, as well as implementations. Another feature is that our design of DS2 facilitates the specification of normal behaviors, and introduces – in a layered manner – various scheduling options that mimic faulty behaviors. This matches the fault/recovery-driven transitions referred to earlier. We now present specific primitives of DS2 that facilitate the creation, extension, and systematic exploration of candidate distributed system designs:

- *The* Locking *mechanism:* It allows an agent to control when to (and when not to) receive messages. It also helps model failures such as network partitioning.
- *Futures:* At the implementation level, DS2 employs a mediator agent (temporary agent) to model an *ask* pattern. The use of a mediator agent removes the need to broadcast the handle to the replier, thus freeing the asker from manually handling the future resolution. This is inspired by the actual Akka [1] implementation.
- *Mixins:* Mixins are known as traits in Scala. They enable the designer to incorporate variations to fault models into a scheduler elegantly. This approach provides flexibility for algorithm designers, while at the same time shielding users from distributed systems complexities. This approach also helps avoid making *a priori* assumptions about fault models.
- *State capture* of the global state and resumption from such captured global states allows us to develop backtracking algorithms, on-the-fly scheduler switching, and parallel schedules exploration.
- *Expressive power-wise*, DS2 [3] is a model and a language[2] that provides a rich set of primitives to support flexible state-space exploration methods (as compared with more standard model-checking notations such as TLA+ and Promela).

Our approach is more inline with the needs of a real distributed systems designer. In contrast, some prior language designs (notably Actors [2], have emphasized novel state-space exploration algorithms [22], and have not emphasized (to the same extent) the creation of a guiding operational semantics or emphasis on key primitives as we summarized.

3. Overview

In this section, we give a brief overview of how our design choices fit together synergistically. In §4, we present a more detailed example, walking through most of the operational semantics rules to address the details.

In a real world example, when a distributed system has been constructed, normally one or more agents comprising it should be bootstrapped, i.e. started. It is exactly the same for our model, however with the scheduler taking control of how the state of the system should evolve, by deciding what events happen when and where.

After bootstrapping an agent, we arrange for it to have a `Start` message in its input queue. The scheduler dequeues this message, finds the matching action from the agent's reactions map, and makes a copy of that action-template and instantiates its relevant parameters. After that, the scheduler *schedules* the action into its task queue (making it a task). The scheduler then can choose whether to schedule something from another agent or merely resume executing the scheduled task(s). Consuming a statement leads to that statement being enqueued into the scheduler's consume queue. We employ a consume queue to expose interleaving effects of statements coming from different tasks scheduled by different agents. The scheduler then executes each statement, removing it from the front of the consume queue. This cycle repeats until some specified stopping criteria are met.

The aforesaid simple design gives the scheduler the ability to model different planes/levels of faulty behav-

[2] We focus on the model in this work

iors in isolation (or combination, if chosen). Examples of situations that can be modeled include re-ordering, duplication, and dropping of messages; these are modeled by manipulating the agent's queue. Disconnects and network partitioning can be modeled by *locking* selected agents before specific statements are executed to update their state, hence simulating missing updates. The scheduler consume queue can be manipulated to simulate different interleavings as well as delays of execution, forcing different bug scenarios. Similarly, a crash in a node (agent) can be simulated, and injecting a fault or a message that causes a fault can be simulated.

4. Walk Through Example

A distributed system is constructed by the code in Listing 1. We begin by creating a distributed system `ds`, a client `c` and a server `s`. Then a reaction is added to the client in lines 6 through 10. Two reactions are then added to the server in lines 12 through 16. Finally, the client and server are added to the distributed system that, in turn, is attached to the basic scheduler (*basic* in the sense of being controlled by the user).

```
1  val ds = new DistributedSystem("Echo ack")
   val s = new Agent("Server")
3  val c = new Agent("Client")
   val act1, act2, act3 = new Action
5  // Client setup
   act1 + Statement(UNLOCK,c) // unlocks the agent incoming q
7  act1 + Statement(ASK,c,new Message("Show","Hello!"),s, "vn")
   act1 + Statement(GET,c,"vn","vn2")
9  act1 + Statement(println("I'm Happy!"))
   c.R("Start") = act1 // (Start, act1) to reactions map
11 // Server setup
   act2 + Statement(UNLOCK, s)
13 act2 + Statement(println("Greetings!"))
   act3 + Statement((m:Message,a:Agent)=>println(m.p))
15 act3 + Statement((m:Message,a:Agent)=>send(s,m(p = true),m.s))
   s.R("Start") = act2 ; s.R("Show") = act3
17 ds += Set(s,c) // adding agents to system
   ds.attach(BasicScheduler)
```

Listing 1: An example distributed system, echo server-client interaction with additional blocking for an acknowledgement on client

One execution of such a distributed system is shown in Listing 2. We deliberately chose a schedule that leads to a large number of rule-firings. The listing is explained via the comments, and Figure 1 visualizes the resulting sequence of states step-by-step.

```
   val sch = ds.scheduler
2  sch.boot(s); sch.boot(c) // sends Start msg to s and to c
   sch.schedule(s) // schedule start task from s
4  sch.schedule(c) // schedule start task from c
   sch.consume(s)  // consume UNLOCK stmt from s task
6  sch.consume(s)  // consume "greeting" stmt from s task
   sch.consume(c)  // consume UNLOCK stmt from c task
8  sch.consume(c)  // consume ASK stmt from c task
   sch.executeOne  // UNLOCK s stmt, IsLocked(s) == false
10 sch.executeOne  // "greeting" s stmt
   sch.executeOne  // UNLOCK c stmt, IsLocked(c) == false
12 sch.executeOne  // ASK s stmt, T = {t} temporary agent
                   // and s.q == [Show("Hello",s=t)]
14 sch.schedule(s) // schedule "Show" task from s
   sch.consume(s)  // consume print("Hello") stmt
16 sch.consume(c)  // consume GET stmt from c task
   sch.consume(s)  // consume resolving send(..) stmt
18                 // note GET blocks, then it is resolved
   sch.consume(c)  // consume "happy" stmt from c task
20 sch.executeOne  // s print("Hello")
   sch.executeOne  // c blocks on GET, doesn't progress
22                 // putting back all stmts after it
                   // from cq back to front of task.xq in order
24 sch.executeOne  // resolving send(..), t.q != empty
                   // things happen to t.L("vn") future resolved
26                 // and then c.q = [RF(f,s=s)], note sender
                   // is s, not t
28 sch.handel(c)   // handling the RF message, unblocking c
   sch.consume(c)  // consuming GET from c again
30 sch.consume(c)  // consuming "happy" stmt from c
   sch.executeOne  // R GET c stmt, won't block (resolved)
32                 // c.L("vn2") = c.L("vn").val
   sch.executeOne  // print("I'm happy")
34 // DONE happy schedule, other schedules are not this happy
```

Listing 2: Example schedule invoking SCHEDULE, CONSUME, ASK, BLK-GET, R-SEND, and R-GET rules. The equivalent visualization of this execution is shown in Figure 1

A design-time question might be: what schedule might lead to a client avoiding blocking on a future? The answer is to swap the consuming of the resolving send (on line 17 of Listing 2), with consuming the blocking get (on line 16 of the same listing). Another intent maybe to exhibit a deadlock by the client. We then keep the same schedule, but we remove all statements and actions, consumed and scheduled respectively, from the server, and reset the server state (i.e. locking the server, then making q = ϵ, and making its local state $\mathscr{L}$ empty) before it executes the resolving send. By doing the latter, we simulated a crash of the server and there is no way that client gets its future resolved, ending it in the simplest forms of a deadlock. One last attempt is to simulate a message drop for a message sent by the same resolving send to resolve the client's future (say we drop *RF* from client's queue, since getting the messages into the queue does not mean its delivered, but rather means it is *in flight* and only considered delivered after it gets scheduled and/or handled by the receiving agent).

All these are illustrations that highlight the flexibility offered by the DS2 approach in creating faulty situations, and forcing a plethora of semi-formal state exploration methods to cover them.

5. Operational Semantics

Despite DS2's simplicity, creating a concise semantics for it proved to be a challenge. Our initial operational semantics spanned dozens of pages [6], which led us to invent several abstract state predicates. The result is a concise semantics expressed in eight simple rules that still retain the full expressiveness that real-world developers demand. For example, it provides both asynchronous communication and synchronization primitives. At the same time, it forms the minimal rule set needed to understand and reason about patterns (§5.2). This conciseness gives us confidence that our operational semantics and model will benefit designers who seek to build formal method tools and those seeking to model and understand new and existing distributed protocols.

We achieved this parsimony by defining a set of structures representing the state of a distributed system, a

(a) Initial State (b) Executed line 2 (c) Executed lines 3,4 (d) Executed lines 5-8 (e) Executed lines 9-11

(f) Executing line 12 (g) Executed line 12 (h) Executed 14-19 (i) Executed 20 (j) Executed 21

(k) Executing 24-(1/4) (l) Executing 24-(2/4) (m) Executing 24-(3/4) (n) Executing 24-(4/4) (o) Executed line 28

(p) Executed 29,30 (q) Executed 31 (r) Executed 33

Figure 1: Example run invoking majority of operational semantics rules. Subfigures refer to line numbers in Listing 2

set of predicates to de-clutter the rules from mathematical details, and a set of conventions to make our presentation intuitive.

5.1. Structures

Now, we introduce the core structures of our semantics. **Distributed System** is a tuple $\langle \mathscr{A}, \mathscr{T}, \Sigma \rangle$ with the set of agents in a distributed system $\mathscr{A}$, the set of all *temporary* agents $\mathscr{T}$ (initially empty; each such agent handles resolving a single future, and then disappears), and a scheduler Σ. We will first elaborate the primitives in our model, and then return to the components of the distributed system.

Message is a tuple $\langle s, p_0, \ldots, p_n \rangle$ where s is the sender agent, and $p_0, \ldots, p_n$ is the payload of the message. The set of messages is symbolized by $\mathscr{M}$.

Statement is a wrapper around the code to execute. It is a tuple $\langle \gamma, code, k, p^* \rangle$ where γ is the action containing this statement, $code$ is the actual code to execute, k indicates the kind of a statement e.g. **Send** for send statement, and p^* are the parameters of the statement. A statement holds a reference to its containing action γ to provide access to the message m that invoked the action and agent a whose local state could be modi-

$$\text{SCHEDULE}\ \frac{ChoseSchedule(\Sigma,\alpha) \wedge \gamma_\alpha\ as\ \langle \mu,\alpha,stmts,\zeta_x\rangle = \mathscr{R}(\mu) \wedge stmts_\gamma \neq \epsilon \wedge \zeta_{x,\gamma} = stmts_\gamma}{\langle \mathscr{A}(\alpha(\mu.q,,\neg b,\mathscr{R},,)),\mathscr{T},\Sigma(,\zeta_t,)\rangle \rightarrow \langle \mathscr{A}(\alpha(q,,\neg b,\mathscr{R},,)),\mathscr{T},\Sigma(,\zeta_t.\gamma_\alpha,)\rangle}$$

$$\text{CONSUME}\ \frac{ChoseConsume(\Sigma,\alpha)}{\langle \mathscr{A}(\alpha(,,\neg b,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,s.\zeta_x),\ldots),\zeta_c)\rangle \rightarrow \langle \mathscr{A}(\alpha(,,\neg b,,,)),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,,\zeta_x),\ldots),\zeta_c.s)\rangle}$$

$$\text{SEND}\ \frac{CurrSnd(\Sigma) \wedge \langle \alpha_s,\mu,\alpha_d\rangle = Involved(s) \wedge \neg IsRSnd(\mathscr{T},s) \wedge ChoseExOne(\Sigma)}{\langle \mathscr{A}(\{\alpha_s(,,\neg b,,,),\alpha_d(q,\neg l,,,,)\}),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle \rightarrow \langle \mathscr{A}(\{\alpha_s(,,\neg b,,,),\alpha_d(q.\mu,\neg l,,,,)\}),\mathscr{T},\Sigma(,,\zeta_c)\rangle}$$

$$\text{ASK}\ \frac{\begin{array}{c}CurrAsk(\Sigma) \wedge \langle \alpha_s,\mu,\alpha_d,vn\rangle = Involved(s) \wedge ChoseExOne(\Sigma) \wedge \\ f = fresh(Future) \wedge \alpha_t = fresh(Agent) \wedge where = fresh(Ids)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\{\alpha_s(,,\neg b,,,),\alpha_d(q,\neg l,,)\}),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_s(,,,,,\mathscr{L}\cup(vn,f)),\alpha_d(q.\mu(\alpha_t,),\neg l,,)),\mathscr{T}\cup\alpha_t(,\neg l,\neg b,,,\mathscr{L}\cup\{(vn,f),(where,\alpha_s)\}),\Sigma(,,\zeta_c)\rangle\end{array}}$$

$$\text{R-SEND}\ \frac{CurrSnd(\Sigma) \wedge \langle \alpha_s,\mu,\alpha_t\rangle = Involved(s) \wedge IsRSnd(\mathscr{T},s) \wedge ChoseExOne(\Sigma) \wedge RF = fresh(Message)}{\begin{array}{c}\langle \mathscr{A}(\alpha_s(,,\neg b,,,),\alpha_d(RF^*.q,\neg l,,,,\mathscr{L}\cup(vn,f))),\mathscr{T}\cup\ \alpha_t(,\neg l,,,,\mathscr{L}\cup\{(vn,f),(where,\alpha_d)\}),\Sigma(,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha_s(,,\neg b,,,),\alpha_d(RF^*.RF(\alpha_s,f(true,\mu[p_0])).q,),\neg l,,,,\mathscr{L}\cup(vn,f)), \\ \mathscr{T}\setminus\{\alpha_t(,,,,,\mathscr{L}\cup\{(vn,f(true,\mu[p_0])),(where,\alpha_d)\})\},\Sigma(,,\zeta_c)\rangle\end{array}}$$

$$\text{R-GET}\ \frac{\begin{array}{c}(CurrGet(\Sigma) \vee CurrTGet(\Sigma)) \wedge \\ (\langle \alpha,vn,vn2\rangle = Involved(s) \vee \langle \alpha,vn,vn2,to\rangle = Involved(s)) \wedge ChoseExOne(\Sigma)\end{array}}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(true,val)))),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup\{(vn,f(true,val)),(vn2,val_f)\})),\mathscr{T},\Sigma(,,\zeta_c)\rangle\end{array}}$$

$$\text{BLK-GET}\ \frac{CurrGet(\Sigma) \wedge \langle \alpha,vn,vn2\rangle = Involved(s) \wedge ChoseExOne(\Sigma) \wedge ss = PreEmpted(\Sigma,\alpha)}{\begin{array}{c}\langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(false,)))),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,\zeta_x),\ldots),s.\zeta_c)\rangle \rightarrow \\ \langle \mathscr{A}(\alpha(,,b,,,\mathscr{L}\cup(vn,f(false,)))),\mathscr{T},\Sigma(,\zeta_t(\ldots,\gamma_{\Sigma,\alpha}(,\alpha,,s.ss.\zeta_x),\ldots),\zeta_c\setminus ss)\rangle\end{array}}$$

$$\text{T-GET}\ \frac{CurrTGet(\Sigma) \wedge \langle \alpha,vn,vn2,to\rangle = Involved(s) \wedge ChoseExOne(\Sigma) \wedge ss = PreEmpted(\Sigma,\alpha) \wedge to \leq 0}{\langle \mathscr{A}(\alpha(,,,,\mathscr{L}\cup(vn,f(false,))),\mathscr{T},\Sigma(,,s.\zeta_c)\rangle \rightarrow \langle \mathscr{A}(\alpha(,,\neg b,,,\mathscr{L}\cup(vn,f(false,))),\mathscr{T},\Sigma(,,\zeta_c.ss)\rangle}$$

Figure 2: DS2 Operational Semantics

fied/accessed by the statement. In addition, in case a statement was preempted, it is known where it should be put back (to the front of the to execute queue of the action γ); A walk-through the rules in §5.4 clears this.

Action is essentially the sequence of statements to execute. More specifically, it is a tuple $\langle m,a,stmts,\zeta_x\rangle$ where a is the agent containing this action (whose local state may get modified/accessed by this action's statements), m is the message that invoked this action, $stmts$ is a sequence of statements or the template. ζ_x is a reference to the *to-execute queue* of statements. The set of all actions is Γ.

Timed Action is an action associated with two values of time. It is the tuple $\langle t1,t2,\gamma\rangle$ where γ is scheduled to execute every $t1$ time with tolerance of $t2$ amount of time, usually used to model *heart beat*. The tolerance time $t2$ is there to add delay-tolerance for message delivery, since it cannot be predicted.

Future is a synchronization construct, which is a tuple $\langle r,val\rangle$ where r is a boolean indicating whether the future is resolved (defaults to $false$) and val is the value resolving this future (defaults to $\perp$, or no value). A future object is returned by the **Ask** statement type, as will be presented shortly.

Agent is a communicating autonomous process. It is a tuple $\langle q,l,b,\mathscr{R},\tau,\mathscr{L}\rangle$ where q is its receive queue, the *reactions* of the agent $\mathscr{R}:\mathscr{M}\rightarrow\Gamma$, timed actions set $\tau=\{\langle t1,t2,\gamma\rangle\}$, $\mathscr{L}$ is the local state of an agent $\mathscr{L}:Ids\rightarrow Vals$. Blocked b and locked l are flags whose initial values are `false` and `true`, respectively. A blocked agent cannot execute statements, consume, or schedule tasks, an agent is blocked if a future it tries to access is not resolved. Locked l is used by the scheduler to tell if the process is receiving messages or not, e.g. its server socket is not open, to model a disconnect, or to model network partitioning.

Scheduler encodes the runtime (an algorithm) of the distributed system. It is a tuple $\langle c,\zeta_t,\zeta_c\rangle$ where ζ_t is the task queue (a queue of actions), ζ_c is the consume queue (queue of statements), and c is a logical clock.

5.2. Communication & Synchronization

Our model uses a minimal set of four communication and synchronization primitives. Such a restriction does not limit the expressivity of the language, while at the same time facilitating understanding. These primitives are now explained.

Send is a fire-and-forget type of statement. Its signature is $send(\alpha_{src},\mu,\alpha_{dst})$. **Ask** is a fire and re-

turn a handle (future) statement. Its signature is $ask(\alpha_{src}, \mu, \alpha_{dst}, vn)$. Its details are similar to that of send, except for vn which is a *variable name* where the handle (future) f is stored in source agent α_{src} local state. **Get** is the statement used to *block on* a future object. Blocking means not executing any more statements till that future object is resolved, then the agent is unblocked to schedule, consume, and/or execute tasks/statements. The blocking-get signature is $get(\alpha, vn, vn2)$ and the timed-get signature is $get(\alpha, vn, vn, to)$. The agent calling it is α, the variable holding the future is vn, the variable that holds resolved value is $vn2$, and the time out limit is to.

5.3. Predicates and Conventions

We first need to state our conventions. **References and Types** are inferred directly from alphabets $\alpha, \mu, \gamma, s, f, vn$ and $where$ as follows: an agent with $\alpha \in \mathscr{A}$, message with $\mu \in \mathscr{M}$, action $\gamma \in \Gamma$, statement $s \in Statement$, and future with $f \in \mathscr{F}$. We, also, refer to variable names with $\{vn, where\} \in Ids$, appended with a number if more than one. The same thing goes to other types. **Subscripts** are used in two ways. First, in parameters to indicate the function of the parameter e.g. α_s for source agent. Second, we use it to indicate where the entity belongs e.g. $\mathscr{L}_\alpha$ for local state of agent α. **Specific task in a scheduler** $\gamma_{\Sigma,\alpha}$ means a front-most action in a scheduler's task queue ζ_t that was scheduled by agent α. We, also, use the same **structure (tuple) as a predicate** with commas indicating place inside the tuple representing them as opposed to dots that are used to state the flexibility of location inside a tuple/sequence. `Involved(s)` returns a variable length tuple according to the kind of current statement. The tuple represents the arguments *involved* in this statement.

$$Invovled(s) = \begin{cases} \langle \alpha_{src}, \mu, \alpha_{dst} \rangle & \text{if } s[k] = SEND \\ \langle \alpha_{src}, \mu, \alpha_{dst}, vn \rangle & \text{if } s[k] = ASK \\ \langle \alpha, f, vn, vn2 \rangle & \text{if } s[k] = GET \\ \langle \alpha, f, vn, vn2, to \rangle & \text{if } s[k] = TGET \\ undefined & otherwise \end{cases}$$

The meanings of symbols inside each tuple returned are as explained in §5.2.

5.4. Rules Walkthrough

Now, we explain the rules in Figure 2 one by one, and illustrate the effects of these rules with the aid of the subfigures of Figure 1 (when applicable). Let us start by explaining one rule, namely `SCHEDULE`.

The `SCHEDULE` rule states that to schedule a task from an agent α, the agent needs to be in the unblocked ($\neg b$) state. Further, the agent should have an entry in its reactions map $\mathscr{R}$ that maps the message received, μ, residing at the head of its queue q (indicated by $\mu.q$), to an action. Moreover, the action field formal parameters m and a must be set to μ and α, respectively. Further, the action's execute queue ζ_x must refer to $stmts$ (indicated by $\zeta_{x,\gamma} = stmts$), where $stmts$ is nonempty. In addition, the `ChoseSchedule(`Σ, α`)` predicate must be true. The predicate means the scheduler *chose to schedule* a task from agent α. We use the "as" notation, as in Ocaml, to serve as an alias; for instance, $\gamma_a \; as \; \langle \mu, \alpha, stmts, \zeta_x \rangle$ uses γ_a as an alias for $\langle \mu, \alpha, stmts, \zeta_x \rangle$. The state transitions to one where the message μ is removed from the front of agent α's queue, and the action γ_α is appended to the scheduler's task queue (ζ_t). An example of this rule before it triggered twice is shown in Figure 1b and after it triggered in Figure 1c.

The `CONSUME` rule fires when the scheduler (Σ) has a task in its task queue (ζ_t). In addition, that task is scheduled by a currently non-blocked ($\neg b$) agent α. Further, the predicate `ChoseConsume(`Σ, α`)` must return true, which means that the scheduler *chose to consume* from a front-most task (symbolized by $\gamma_{\Sigma,\alpha}$) that was scheduled by agent α. Then, the system transitions by (1) popping the statement that is at the front of the task's execute queue ($s.\zeta_x$) (2) appending that statement to the back of the scheduler's consume queue ($\zeta_c.s$). An example of this rule before triggering four times is shown in Figure 1c and after it triggered in Figure 1d.

Now we explain the `SEND` rule. This rule states that if the current statement s at the front of the scheduler consume queue ζ_c is a send statement (`CurrSnd()`), and that statement parameters returned by `Involved(s)` are $\langle \alpha_s, \mu, \alpha_d \rangle$, and the statement is not a resolving send ($\neg IsRSend(\mathscr{T}, s)$), i.e. the destination α_d isn't a member of the temporary agents. In addition, neither the source agent α_s is blocked ($\neg b$) nor the destination agent α_d is locked. Then, if the scheduler *chose to execute* a statement (`ChoseExOne(`Σ`)`), the transition happens. That is, the message μ is appended to agent α_d queue ($q.\mu$). In addition, the statement is removed from the front of the scheduler's consume queue (ζ_c). An example of this rule when it triggers is shown in Figure 1f and after it completes is shown in Figure 1g.

`ASK` rule states that if the current statement to execute is an ask, predicate `CurAsk(`Σ`)`, and like in `SEND` rule, the source agent is in unblocked state ($\neg b$) and the destination agent α_d is in unlocked state ($\neg l$). Then, if the scheduler *chose to execute* a statement, a fresh temporary agent α_t (i.e. α_t =`fresh(Agent)`) is created along with a fresh future $f \in \mathscr{F}$ (i.e. f =`fresh(Future)`). Then, the transition happens: (1) the temporary agent α_t is added to the temporary agents $\mathscr{T}$ (2) the future f is added to both the temporary agent and the source agent α_s local states under the key vn, i.e. $\mathscr{L} \cup (vn, f)$ (3) the temporary agent local state updated with the $(where, \alpha_s)$, to keep track *where to* forward the RF (Resolve Future) message in case the future was resolved. (4) the sender field of the message updated to be the temporary, $\mu(\alpha_t,)$ (5) the ask message enqueued at the destination agent α_d's receive queue, $q.\mu$. An example operation of this rule is visualized in multiple frames of Figure 1, namely in frames 1e, 1f,

and 1g. Important to notice that the updates to local states of both temporary and source agent stated here are *not* shown in the figure due to space constraints.

R-SEND rule states the same guards as a regular SEND rule except that the destination is a temporary agent, i.e. predicate $IsRSnd(\mathcal{T},s)$. As such, additional work need be done over a normal send by α_t. So, if the scheduler *chose to execute* the statement, the transition happens: (1) The temporary agent updates its future resolved status to `true` and that future's value from the first payload of the message μ, and encapsulates it in a fresh RF message setting its sender to the original sender α_s, i.e. $RF(\alpha_s, f(true, \mu[p_0]))$ (2) The RF message is then *inserted* into the destination agent α_d queue with the resolved future, before all non-Resolve-Future messages but after all other RF messages, as shown by $RF^*.RF(\alpha_s, f(true, \mu[p_0])).q$ (3) The temporary removes itself from the temporary agent's set, $\mathcal{T} \setminus \{\alpha_t(,,,,,\mathcal{L} \cup \{(vn, f(true, \mu[p_0])), (where, \alpha_d)\})\}$. Up to this point, the future is considered resolved, however it is up to the scheduler implemented to decide when to update α_d local state with the resolved future, as can be told from the post state of the destination agent local state, $\mathcal{L} \cup (vn, f)$. An example of a resolving send executing is shown in multiple frames in Figure 1, namely frames: 1k, 1l, 1m and 1n.

R-GET rule states that if the current statement is either a blocking-get (i.e. `CurrGet(Σ)`) or a timed-get (i.e. `CurrTGet(Σ)`), and the future they try to retrieve is already resolved,$f(true, val)$. In addition, that future is stored in α's local state under entry vn. Further, the scheduler *chose to execute* a statement. Then, that future's value is retrieved and stored in another entry in the local state of the same agent, i.e. $\mathcal{L} \cup \{(vn, f(true, val)), (vn2, val_f)$. Figure 1p shows the state before this rule triggered, and Figure 1q shows the effect after it is triggered by agent c.

BLK-GET rule states the same guards stated by R-GET except that: (1) the future in this case is *not* resolved (2) the current statement is a blocking-get ($CurrGet$) (3) and the future is unresolved $f(false,)$. If the scheduler *chose to execute* a statement, a transition happens: All statements indicated by ss, that are returned by the $PreEmpted(\Sigma)$ in the same order they were consumed, are *appended* to the current statement and the resulting sequence of statements *prepended* to the front-most task execute queue, i.e. $s.ss.\zeta_x$. `PreEmpted(Σ)` determines all those statements, that were consumed from the same agent α tasks and returns them. Then, these statements in ss are removed from the consume queue, as in $\zeta_c \setminus ss$. Lastly, the agent blocking status is updated from unblocked $\neg b$ to blocked b. An Example of this rule prior it triggers is shown in Figure 1i and after it triggers in Figure 1j.

T-GET rule states the same guards as in BLK-GET rule except it does not block indefinitely. It only blocks temporarily until it times out ($to \leq 0$), delaying execution of all those statements till the agent unblocks. That is the time for them to have been consumed, i.e. appended to the consume queue of the scheduler, $\zeta_c.ss$ (skipping the statement s). Agent α's state changes to unblocking.

6. Bug Discovery in Snapshot of Runtime

The operational semantics have already proven useful, even in the design and implementation of DS2 itself. For example, before we had an operational semantics, we struggled to correctly capture snapshots of the runtime state (which includes both the distributed system and the scheduler attached to it). For the most part, a system and its scheduler must be "deep copied," to create an isomorphic state; however, not everything can be copied verbatim. When copying a distributed system, the steps must be carried out in two phases: (1) a copy phase (creating objects but leaving references untouched) for all entities in it, followed by (2) a link phase (*re-wiring* references to entities from new snapshot) for all of them. However, in our original implementation, this was not the case. For example, an action's ζ_x statements kept on referring and affecting the original distributed system's agents even after the linking phase. More over, ζ_x statements were not the same statements from the snapshot's `stmts` template. So, when the link operation updated the agent field `a`, which in turn updates all of the template statements, it did not reflect in those inside of ζ_x. The operational semantics exposed the bug in the original snapshotting implementation that left actions in the snapshot *still attached* to the parent distributed system instead of their snapshotted counterparts. The operational semantics were essential in correcting snapshotting, which will be the cornerstone of DS2's model checking and testing functionality.

After correcting this mistake, DS2 can now easily and reliably capture, fork, and restore distributed system states. Listing 3 shows how succinctly one can now snapshot a whole distributed system along with its scheduler state and then restore from it.

```
1 val state = saveState(sch)
2 restoreState(sch,state)
```

Listing 3: Capturing and restoring to runtime state

A key detail here is that the scheduler `sch` *need not be the same scheduler that saved the state*. This flexibility enables different schedulers, for example, those running truly concurrently to explore different paths in the distributed system's state-space; and/or cooperation between different analyzing schedulers, i.e. switching on the fly between different kinds of analyses to be performed by different schedulers. An additional importance of this snapshot feature is enabling backtracking/stateful algorithms.

7. Related Work

Several distributed-system related projects have emphasized the use of formal semantics. Simg-

Grid [20]focuses on the simulation and model checking of distributed systems. InVerdi [7], the Coq system is used to develop formal operational semantics fornetwork and node-failure modelsto synthesize distributed systems from specifications. Our work focuses on targeted correctness of distributed systems, as opposed to performance simulation as in SimGrid. Our goal is to allow designers to model a variety of distributed systems in the DS2 language; the formal definition of a modeling language is not targeted in Verdi.

MoDist [23] is a transparent operating system-agnostic model checker for unmodified distributed systems. Our work is extensible to address specific needs of distributed systems using custom schedulers. MaceMC is an execution based model checker for distributed systems specific to the Mace language. It benefits from coarse-grained interleaving exploration and random walk [14].

IronFleet [10] is a layered approach to TLA-style [17] state-machine refinement and Hoare-logic [12] based verification to synthesize provably correct distributed systems, developed by Microsoft Research.

8. Conclusion

Current implementation of our DS2 core, in its final stages of testing, tightly follows the formal operational semantics explained in this work. We are working on developing an extended version of linearizability checking scheduler for distributed systems inspired by Line-up [5] and guided by the operational semantics explained in this work. Our direct target is developing a front-end parsing the benchmarks we developed in Akka: Paxos [16], Chord [21], and Zab [15] for our linearizability checker.

Our long term future targets are semantics-aware distributed systems automated testing, analysis, and synthesis. We are developing many use-cases [3], [6] to drive our work forward along these lines.

Acknowledgments: We acknowledge NSF grant CCF 7298529, and help by Heath French, Zepeng Zhao, and Anushree Singh.

References

[1] Akka, February 2016. http://akka.io/.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT, 1985.

[3] Mohammed Al-Mahfoudh, Ganesh Gopalakrishnan, and Ryan Stutsman. Toward rigorous design of domain-specific distributed systems. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, FormaliSE '16, pages 42–48, New York, NY, USA, 2016. ACM.

[4] Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.

[5] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, Inc., June 2010.

[6] Distributed systems abstract model, 2016. http://formalverification.cs.utah.edu/ds2, Retrieved Jan 31, 2016.

[7] David Grove and Steve Blackburn, editors. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015.

[8] Philipp Haller and Martin Odersky. Event-Based Programming Without Inversion of Control. Springer Berlin Heidelberg, 2006.

[9] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

[10] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015.

[11] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. 1973.

[12] Hoare logic, 2016. https://en.wikipedia.org/wiki/Hoare_logic, Retrieved Apr 22, 2016.

[13] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2000.

[14] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[15] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256, 2011.

[16] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[17] Leslie Lamport. *Specifying Systems*. Addison Wesley, 2003.

[18] Caitie McCaffrey. The verification of a distributed system. *Communications of the ACM*, 59(2), feb 2016.

[19] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, Philadelphia, PA, June 2014. USENIX Association.

[20] Henning Schulzrinne, Karl Aberer, and Anwitaman Datta, editors. *Proceedings P2P 2009, Ninth International Conference on Peer-to-Peer Computing, 9-11 September 2009, Seattle, Washington, USA*. IEEE, 2009.

[21] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[22] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *NSDI*, pages 213–228. USENIX Association, 2009.

[24] Pamela Zave. How to make chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015.