Toward Rigorous Design of Domain-Specific Distributed Systems

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

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

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

ABSTRACT

The advent of data center, cloud computing and IoT has thrust distributed systems building into the programming mainstream. Building correct distributed systems is notoriously hard, yet today's developers have little training and few tools to aid them in reasoning about these complex systems. To that end, we present DS2 – a domain-specific language and integrated framework for specifying, synthesizing, and reasoning. The DS2 language is parsimonious, and comes with an operational semantics that lends semantic clarity and enables formal analysis. A variety of techniques for model exploration, active testing, and synthesis of detailed implementations from higher level specifications are being developed. This paper details these aspects of DS2 and provides a roadmap of its evolution.

Keywords

Distributed Systems; Fault Tolerance; Formal Methods; Concurrency; Actors

1. INTRODUCTION

A recent book on the principles of Distributed Systems [10] states: *All implementations of mutable shared state in a geographically distributed system are either slow (require coordination when updating data) or weird (provide weak consistency only).* As this quote implies, the need to deploy loosely coordinated distributed systems and services is accelerating, but in terms of reasoning about the correctness of these systems, tools and techniques are lacking.

Today, distributed systems exist at all scales: cloud and enterprise data-center computing, and "the Internet of Things (IoT)" connecting home, vehicle, and factory automation systems. Unfortunately, distributed system *design* is marked by rich, but poorly principled approaches. On one hand, the abundance of cheap hardware and multiple programming languages (for example, Erlang, Akka, and Go) enable programmers to quickly create and deploy distributed systems. On the other hand, productive ways of specification and verification are severely lagging [22]. The state-space of a distributed system enormous; additional complexities such as loose interaction, replication of states, and faults make the overall specification of desired functionality even harder. Modern languages such as Akka, while well-conceived, only come with English specifications, and there is no clear path to ensuring behavioral consistency across platforms.

We present DS2, our project on rigorous design of domain specific distributed systems, whose principal goals are to enhance productivity while ensuring correctness of implementations. We aim to eliminate needless variety of coding patterns. Instead, we aim to derive prevalidated compositions of detailed designs from higher level specifications. The DS2 project is inspired by projects such as DeLite [27] demonstrated in the context of tightly coupled parallel computing systems. Our goals in DS2 go far beyond the traditional notions of race-freedom and automated code parallelization; we aim to bring in correctness notions specific to distributed systems (e.g., linearizability) while also enhancing the level of automation in deriving implementations. To this end, we embark on a "clean-slate design" based on a parsimonious language that still is adequately expressive. We provide a rigorous operational semantics and a state model that is conducive to formal analysis through various "active testing" methods [22].

Our approach is a fusion of formalism into practice using abstraction and syntax that captures both aspects of distributed systems while hiding formal complexities from practitioners. It makes the following contributions:

- A generic model for exploring distributed systems built on but not limited to, source or target[1], the actors model.
- Its formal operational semantics.

FormaliSE'16, May 15 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4159-2/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897667.2897674

[1]Our aim is to provide only one backend that outputs Scala Akka code for synthesis, as well as one front end for DS2 language that uses scala as a proof of concept

- A new approach for asynchronous, deterministic, fault tolerant non-byzantine networked distributed systems development and checking.
- Syntax and semantics guided synthesis of property-conforming systems by capturing the higher intent of developers and baked in model exploration for their implementations in one language, DS2.
- Modularizing local computations and communication in one communicating process model that is easy to understand and reason about.

2. BACKGROUND

To better understand the key issues in building distributed systems, we have modeled [13] several influential and popular protocols including Chord [25], Zab [18, 17], and (Multi-)Paxos [19, 20] in Akka [6]. Our observations highlight the problems of distributed systems building and motivate DS2; they are summarized below:
Extreme non-determinism. Programmers are used to assuming fast access to shared memory; a single, strong time source; fault-freedom; and cheap, strong ordering. Distributed systems violate all of those assumptions. Reasoning about and even modeling the simplest distributed systems is challenging; code size, complexity, and state-space all explode in this environment [26].
Language generality/imprecision. Each developer chooses their own language and design. The generality of these languages leave the developer with an enormous number of implementation decisions, each of which obscures the relationship of the implementation to the intended protocol. These details can leave the system imperceptibly incomplete or broken. Furthermore, varying languages and environment lead to implicits and domain assumptions that are not clearly stated or are inconsistently applied, making it impossible to safely compose with other systems or to automatically verify.

The above issues drive DS2's design; specifically:
Domain-specific language. DS2 is a domain-specific language to address this imprecision and mismatch. A domain-specific language with formal operational semantics makes assumptions clear. It boosts productivity by easily producing and composing complex but proven code from concise specifications. At the same time, it takes advantage of domain specific knowledge and implicits to both restrict developers enough relieving them from mundane and potentially dangerous design decisions, and to synthesize implementations (even targeting multiple backend environments).
Actor model. A concurrency model should be as expressive as needed to aid development and performance, but as simple possible to aid both formal and informal reasoning. DS2 relies on small, modular rules-based (Akka) actors [15, 7, 14, 6]. These actors yield three benefits. First, they work naturally in distributed systems, which are often driven by internal events (messages) and external events (outside of the control of the system; machine failures, for example). Second, they enable easy local reasoning about actions, since each action induced by a message executes atomically. Finally and most importantly, they provide a strong semantics for reasoning about concurrency and simplify model exploration. Actors' strong similarity to Hoare's Communication Sequential Processes (CSP) [16] makes it easy to formalize its semantics, and they are familiar to programmers thanks to recent programming languages that build on CSP (for example, Go and Rust). However, Akka actors lack a formal operational semantics specification, and we discovered disconnects between its documentation and implementation. Consequently, we provide an operational semantics for DS2, which we outline in the following section (the full operational semantics is available online [13]).

Motivated by these studies and findings, we present our language design next. We will illustrate our work on a case study after, based on a primary backup replicated service similar to Zab.

3. LANGUAGE DESIGN

Our model, upon which our language is to be built, captures distributed systems using communicating, event driven, with parallel periodic tasks processes called *agents*. Each agent manifests a *behavior* (inspired by Akka's behaviors [6, 15, 7]) that maps events to actions. Behaviors can be configured at runtime according to the role the agent is performing. A collection of agents is a *distributed system*. Agents have life cycle hooks that can be overridden for certain "special events"; `onStart`, `onJoin`, `onRejoin`, for example . This model adheres closely to Akka's, which captures a wide gamut of real-world, production distributed systems. Figures 1 and 2 (respectively) describe the DS2 architecture and the basic types used to specify systems in DS2. Those manifest as DS2 components, now described:
• **Statement:** a lambda `c` of type $\mathscr{C}$ of distributed system code wrapped together with meta data.
• **Action:** a sequence of statements.
• **Agent:** a communicating process with multiple enabled behaviors, a local state, incoming queue, a delay queue called "stash", and a behaviors stack.
• **Behavior:** a map from message received (sender encapsulated inside the message) to actions acting on the receiving agent's state with multiple outcomes: generating more messages, creating/killing another agent(s), changing local state, or changing behavior of the receiver.
• **Future:** the only synchronization construct in the model, encapsulating a value and providing blocking (i.e. stopping a process from proceeding) of the accessor agent task till it is resolved.
• **DistributedSystem:** a collection of all agents, behaviors, actions, and messages and it encodes all operational semantics of a distributed system relevant operations. It is the "context" object for our model that is based on the "Strategy" OO design pattern, where the scheduler is the strategy.

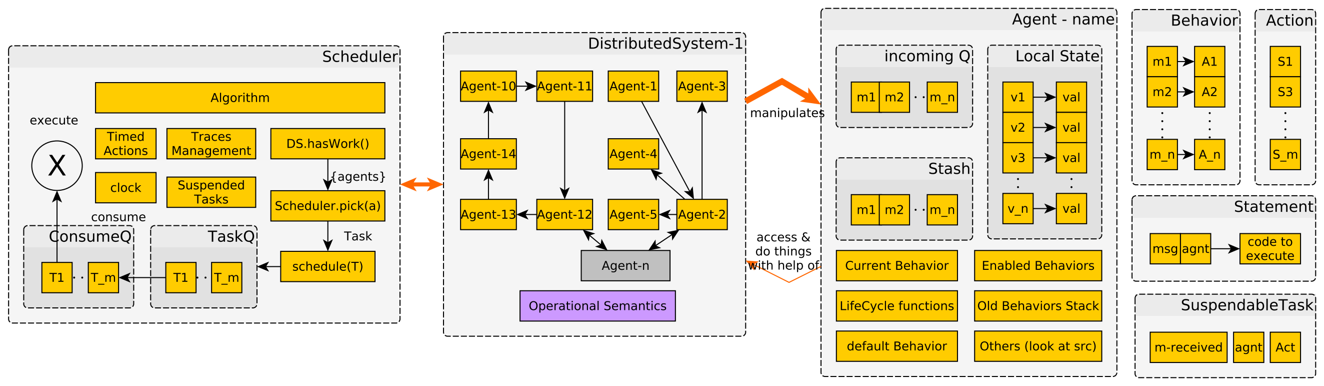

Figure 1: DS2 Architecture: DS2 Runtime data structures and their inter-relationship

$\mathscr{A}$ the set of all agents in the distributed system, synonymous to actors in Akka or Erlang.

$\mathscr{M}$ the set of all messages in the distributed system.

$\mathscr{K}$ ∈ {NONE, SEND, ASK, RESOLVE, CREATE, START, STOP, KILL, LOCK, UNLOCK, STOP_CONSUME, RESUME_CONSUME, BECOME, UNBECOME, STASH, UNSTASH, UNSTASH_ALL, GET, GET_TIMED, BOOT_STRAP, BOOT_STRAP_ALL, MODIFY_STATE}; the statement's kind.

$\mathscr{B}$ the set of *blocks of statements*. Similar to a basic block in C; here, a block of Scala statements.

$\mathscr{C} \in \mathscr{M} \times \mathscr{A} \rightarrow \mathscr{K} \times \mathscr{B}$; the "code" data type.

$\mathscr{P}$ the universe of all programs. It includes both the host language statements (in our case Scala) as well as our IR data structures and scheduler's statements. So, $s \in \mathscr{P}$ is a generic statement, while $t \in \mathscr{K} \setminus \{NONE\}$ is a distributed system specific statement for our operational-semantics-guided reasoning.

$\mathscr{D}$ the universe of all distributed systems.

Figure 2: The Language Model.

• **Scheduler:** the exploration/analysis algorithm acting on the context object (Distributed System).
• **Communication patterns**: an asynchronous "send and forget", and an asynchronous send called "ask" that returns a future that can be used to wait for a result.

A user of DS2 creates agents and specifies behaviors for each. Behaviors are specified with a simple rule-based language that maps messages to actions. Agents that require fault tolerance can specify fault tolerance strategies for their state. The user can also specify temporal events for liveness checking. Overall, each agent is a basic (event → reaction) specification, whose fault tolerance and liveness code is synthesized by the language.

Inter-agent communication patterns are expressed in DS2 with sets and/or regular expressions over agent's names. For example, one can specify a replication rule (discussed in more details later) as:

```
replicated[main][* \ {server2}][primary](data-elem)
```

This description means *replicate on all agents except* `server2`. Next, the user creates a `DistributedSystem` and attaches a `Scheduler`. Different schedulers may have different purposes: one scheduler may run the system on a real cluster, another may model check the system or do coverage-directed systematic testing, yet another may expose controls to allow the user to drive the system toward goal (or edge) cases.

Overall, the core is built around an extensible "strategy" object oriented design pattern. A `DistributedSystem` a context, and schedulers are the strategies exploring the context. Custom analyzing schedulers can easily extend the `Scheduler` class for coverage-directed active testing methods such as depth bounded and lineage-directed [22] testing. Custom schedulers also allow random or synthetic events to be generated to drive a distributed system towards a certain goal (or failure) state; the importance of fault injection is exemplified by systems such as Jepsen [22].

To support complex scheduler policies and to support the parallelization of model exploration, the complete distributed system state can be snapshotted and restored. For example, this can be used to externalize the state, to fork parallel schedulers, or for backtracking schedulers to reset state. This feature in the final stages of development, testing, and debugging.

Schedulers can also selectively drop, duplicate, or reorder messages to weed out incorrect assumptions about the underlying network model. Consequently, it can also simulate arbitrary network partitions, and the death and rejoin of agents (via life cycle hooks): an extremely common source of bugs in distributed systems [8].

Overall, the model readily supports static analysis, dynamic analysis, and post-mortum analysis with a single, simple model. For example, the existing implementation already supports trace output in JSON. We are currently beginning the integration of a graph query library (Graph4s [3]) to support post-analysis of distributed system state and static deadlock detection. Eventually, Graph4s will represent the distributed system's state as a graph and support the scheduler by allowing it to extract sub graphs pertaining to its analysis. Finally, dot support is being integrated to support static and dynamic visualization of systems. Ultimately, the plan is to support a stepper/visual-debugger for the runtime.

4. OPERATIONAL SEMANTICS

In this section we present a sketch of our model, while the full length of operational semantics is given at [13]. Almost all naming conventions in DS2 are borrowed from Akka's, except renaming Actor to Agent to avoid confusion between implementational and model aspects. Agents do not share state. Communication between agents is achieved using two primary patterns:

- **Send and forget**: a send operation is simply a message enqueued (with the sender reference encapsulated inside the message) into the destination's incoming queue of messages.
- **Ask**: a send operation that returns a handle called a *future*, which the sender can selectively block on (halt execution on) until the future is resolved.

A future is resolved by sending a special message, "Resolve Future" (RF), to the waiting agent. Resolving a future is a system task, so no reaction is mapped to it in the agent's current behavior.

The behavior of agents is decided by the statements executed on prior received message(s). A received messages induces the associated action. Since all of the statement metadata is known a priori, flexible scheduling methods are facilitated, as outlined in §3.

The previous example shows the importance of the ask pattern. It cleanly exposes the interleaving of messages and their effects on resolving a future. First, there is a send operation for a message from a source to a destination agent. Then, there is an implicitly returned future with an associated return RF message, which comes later, to resolve the future which the sender is blocking on. The absence of an RF message results in a deadlock. Prior to writing the operational semantics for DS2, it was not clear how ask patterns could be forced to lead to a deadlock in a distributed setup. We plan to develop specific deadlock detection strategies and queries on the state-space using our Graph4s support.

5. WORKING EXAMPLE

Now we look at an implementation for a simplistic echo server and client in the IR described previously and shown in Figure 1. First we begin by constructing a distributed system object, then we link it to a scheduler, then return it. The scheduler then does its job by running the distributed system in any desired way. Currently, we have a base scheduler that can be used as-is or extended by automated schedulers to explore the distributed system state.

Listing 1 shows an example echo server and client distributed system constructed using our DS2 model. First a distributed system object, whose name/id is `echo`, is created. Two agents are then created and given their respective names, `client` and `server`. `sc` forms the statement's code to execute by the server upon receiving a message whose name is `Show`. That statement has to be inside an action `sa`, and associated with the message object that will invoke it to create a *reaction* `sr`. Another reaction `ser` is created in similar way from `se` comprising the error action `sea`. Then, both reactions are added to the `default` behavior of the server `reactions`. The agent `s` is then updated with both its `defaultBehavior` and `reactions`, latter meaning the currently enabled behavior of the agent. After that, the client is constructed in the same way the server was. All messages have to be defined in the distributed system, so `m` is added to the distributed system's collection of messages. Agents are then added to the distributed system, the distributed system is refreshed, then returned ready for execution. Refreshing a distributed system is important since one of its tasks is to link all actions, their statements, and several entities with the containing agent's reference. This is to allow actions act on the agent's state.

```
val ds = new DistributedSystem("echo")
ds.enableTracing
ds.scheduler = Scheduler(ds,Scheduler.Basic)
val c=new Agent("client");val s=new Agent("server")
val sc= (m:Message,a:Agent) => { println("Received: " + m.payload.mkString)}
val sa = new Action; sa + Statement(sc)
val se = Statement((m:Message,a:Agent) => {if(m.name != "Show") println("ERROR!")})
val sea = new Action; sea + se
val sr = (new Message("Show"), sa);
val ser = (new Message, sea)
val reactions = new Behavior("default")
reactions += sr; reactions += ser
s.defaultBehavior = reactions
s.reactions = reactions
val m = new Message("Show","Whatever to show")
ds.messages += m
val c1 = (m:Message,a:Agent)=>{ds.send(a,m,s)}
val ca = new Action
ca + Statement(c1)
c.specialReactions += (new Start, ca)
ds + s + c; ds.refresh; ds
```

Listing 1: Distributed system Construction and attaching a scheduler

Now, the distributed system is ready to be explored by the scheduler `sch`. Listing 2 shows one schedule that distributed system can exhibit. It is a correct schedule and is the expected normal operation. However, any particular schedule can be forced on the distributed system to expose a certain bug/behavior. Faults can also be injected at any point of execution anywhere in the entire runtime to witness how the distributed system evolves from there on following the remaining correct execution schedule. This is an important feature to enable systematic and targeted exploration of the distributed system's state.

The schedule in Listing 2 first boot straps the server followed by the client, by sending a `Start` message to them. Then, the `Start` task from the server is scheduled, followed by scheduling the client's. The scheduler then *consumes* these two tasks, i.e. referring to Figure 1 moving them from the task queue to the consume queue, in order for them to be executed. In between scheduling and picking the tasks, as well as between scheduling and consuming, there is a plenty of room to do whatever is fit for analyses. More over, multithreading can still be simulated using a single threaded scheduler by shuffling the consume queue. Much flexibility is allowed to achieve various goals. Returning to our walk through, the scheduler executes the server's task on a virtual thread whose id is 1, unlocking the server's incoming queue, and does the same for the client's task waiting in the consume queue. Executing the client's task, however, not only unlocks the client's incoming queue, but also fires a `Show` message to the server. So, now the server's queue is still holding one more task. So, it schedule's it, consumes it, then executes it printing "Received: Whatever to show". Then the distributed system is shutdown. It is clear, from this example, that even termination analysis can be done for example.

```
ds.scheduler.enableTracing
ds.bootStrap(ds.get("server"))
ds.bootStrap(ds.get("client"))
val sch = ds.scheduler
sch.schedule(sch.pick(ds.get("server")))
sch.schedule(sch.pick(ds.get("client")))
sch.consume;sch.consume;sch.execute;sch.execute
sch.schedule(sch.pick(ds.get("server")))
sch.consume;sch.execute;ds.shutdownSystem
```

Listing 2: Enforcing a single schedule

Running the schedule from Listing 2 produces a trace, part of which is shown in Listing 3. A sequence of trace entries includes the prior distributed system state, the event, and the posterior distributed system state; the trace format keeps all of the distributed system's state intact. A graph of system states can also be extracted to determine which statement blocks on a future and which statement resolves it. Analyzing the graph for cycles that contain blocking and no resolving a priori nor in that cycle that leads to resolving that future, then we know it is a deadlock.

```
Pick(server: queue=Start(sender = boot
DoSchedule(Task: from 'server'
Pick(client: queue=Start(sender = boot
DoSchedule(Task: from 'client'
Consume(Task: from 'server'
```

Listing 3: Trimmed down part of trace showing events and some details encapsulated

The trace above shows that it is not clear what event leads to which other events. We call these nested events. Our trace format includes *event-end tags* and events have ID's to indicate when an event starts and stops. End events are not shown in the example for brevity.

6. CASE STUDY

Primary-backup replication is pervasive in distributed systems. It is commonly implemented from scratch, since it has a simple intuition. In its simplest form, a set of servers form a group with one marked as primary; the primary accepts and orders all operations, and it synchronously replicates operations to backups for fault tolerance. A naïve implementation can lead to blocking and/or inconsistencies, and many deployed implementations are vulnerable to data corruption under network partitions (that is, arbitrary patterns of agent failure).

As an example, imagine a simple primary-backup replication implementation that assumes a reliable network that delivers messages in bounded time. Even with these unrealistically strong network assumptions and an assumption of clean fail-restart agents, there are many scenarios that must be handled for correct operation. For example, imagine three servers, Main, S1, and S2. Main replicates an item, d, to S1 and S2. Each member of the replication group is aware of its peers, and Main tracks the most recent state of d acknowledged by each of its backups. Ideally, this group of 3 servers should be able to tolerate 2 simultaneous failures and continue to make safe forward progress.

For correct replication, the implementation must handle many scenarios. First, data on all replicas must be durable even if an agent crashes and restarts. Second, whenever Main (or a successor) crashes a single new primary must be chosen. This is possible on a reliable network with bounded delays by relying on static "priorities" between the peers, but under realistic (asynchronous) network assumptions this requires complex consensus protocols [19, 20, 23, 18] in order to safely choose a new primary without blocking or risking "split brain". Third and hardest, a newly elected primary must ensure that stale state from another peer cannot subvert its updates later. Imagine S1 fails and restarts. In the meantime, S2 accepts new updates from Main. Finally, Main and S2 crash. If S1 returns to the cluster, it may become primary, which would result in lost updates. Worse, S1 may accept some updates and fail, then Main might return and accept updates. Such a scenario results in arbitrary forks in the state of the system. This final problem, in particular, is difficult to get right.

Though, this is one of the most basic (and common) approaches to replication and fault tolerance, it is an ordeal in any generic programming language where each case must be coded manually. Timed events, asynchronous events, and outside failures complicate code flow; they result in code that must be prepared to "change direction" as new information and events are discovered. Even with (overly) strong network assumptions there many scenarios to consider and standard asynchronous network assumptions yield a code base of thousands of lines in any conventional programming language, usually with several layers of concurrent code.

6.1 Primary-Backup in DS2

DS2 domain-specific language eliminates the need to reason through all of these complex cases. Instead, a developer states their underlying network and failure model assumptions and then can express safe primary-backup among a set of peers in a single line:

```
replicated[main][s1,s2][primary](d).on(3 updates)
```

This declarative statement indicates replication

- with Main initially as the *primary* agent;
- together Main, S1, S2 form the replication group, and that all replicas receive all operations;
- d of type T should be replicated;
- that the *primary*-backup replication protocol should be used, though the precise algorithm used may depend on user stated network and agent failure assumptions and/or clock drift assumptions;
- finally, the frequency or batching of replication operations (here every "3 updates" to d).

Our goal is for DS2 to automatically synthesize code to handle these scenarios and produce a "correct by design" system. The rest of the section describes how we expect synthesis to work for the above statement.

6.2 Synthesizing Replication

To synthesize the replication scenario from above, start by assuming an existing distributed system with agents Main, S1, and S2. The specification above will guarantee these things, but the details are straightforward. The first significant effect of the `replicated` statement is that each agent is augmented with additional state to track replication progress:

```
d = 0 // data item
cd = 0 // count of updates to 'd'
vd = 0 // version ID of 'd'
csd = d.hashCode() // check-sum of 'd'
replicatedOn = {d: [s1,s2],...}
alive-agents = [s1,s2]
```

The details of this extra state vary depending on system-wide assumptions (§6.1). These fields are maintained by additional synthesized code. S1 and S2 are similarly augmented. Most importantly, vd tracks the version of d that is stored in each agent. The type of this version can vary depending underlying system-wide assumptions (for example, when correct synthesis requires consensus this version may be a pair that indicates a view number along with a intra-view monotonic version number).

Each agent sends a *heartbeat* as a DS2 timed action (`doEvery(5 seconds, ds.send(this, others))`, which agents to determine the status of replication peers. `alive-agents` is a set of agent names believed to be alive, initially all should be. Each agent's view of the system evolves over time, and agents do not always have an identical view of the system. The `others` set differs for each agent; it is the `recvr.replicatedOn` map that tracks what data elements are replicated on which agents.

The basic replication operation of the primary mostly issues simple, parallel replication messages:

```
cd++; vd++; csd += d.hashCode()
if (cd%3 == 0) {
  m = Message("Replicate", payload = [d, vd])
  ds.send(main, m, s1); ds.send(main, m, s2)}
```

This code is adapted for each sender to contain the correct set of replication receivers (on S1 and S2). The code takes care of updating d's associated `vd` and `counter` fields on each agent. In addition, the if statement implements the "on 3 updates" batching. Each time the data element d is updated, the above code is executed.

Finally, each agent is augmented to process heartbeat messages (`recMsg`, below) and to detect missing updates at the sender and receiver by comparing version IDs for d. Upon version mismatch, the handler either updates its local value or sends its value back to the sender.

```
// 'd' was updated; recvr needs to catchup
if (m.payload(3) > recvr.vd)
  // just one batch update happened
  if(recMsg.payload(2) - recvr.vd ==3)
    update(recvr.locals , recMsg)
  // > 1 batch update, recvr missed >= 1 update
  else if (recMsg.payload(2) - recvr.vd >3 )
    updateElaborated(recvr,recMsg)
  // recvr ahead, let other's know
  else if (recMsg.payload(2) - recvr.vd < 0 )
  { m = Message("Replicate", payload = [d,vd, csd])
    replicateTo(replicatedOn , m)}
else // more sophisticated fault-tolerance work
  somethingIsWrong(m) // use checksum+other means
```

In the future, DS2 will aim to address all of the above code synthesis with support for different and extensible strategies for fault tolerance that cleanly adapt to user-specified network and agent failures assumptions.

7. RELATED WORK

P. [12] is a language developed for synthesizing verified device drivers in C. Our language lends itself more naturally to the distributed systems domain.

P#. [11] is a language developed by Microsoft Corp. for addressing asynchronous programming, analysis, and testing using the state machine approach. It does not address concerns in distributed systems, including failure handling.

Q-ImPrESS [4] addresses Service Oriented Architectures (SOA) and Quality of Service (QoS), as opposed to our focus on distributed systems.

Save-IDE. [24] An IDE that integrates the design of hardware and software of embedded systems modeled after timed automata verified using UPAAL [9]. Our model, on the other hand, addresses both timed (with time drift) and deterministic, i.e. event driven, aspects of a rich distributed system running over an asynchronous network of embedded devices.

Verdi. [28] is a framework for formally verifying distributed systems implementations, in a possibly faulty network. It requires users to specify properties where they see fit. We believe that many developers of distributed systems may not know what and/or where those properties need be specified, nor be able to express them in Coq [2].

Coq [2], TLA+ [21], Promela [5], and Alloy [1]. While these formal analysis systems are available, they have not been directly connected to distributed system design and high-level synthesis concerns.

8. CONCLUSION

We have argued for prototyping and correct by construction synthesis tools for distributed systems. To this end, we detailed the design of DS2, a domain-specific language for specifying, verifying, and constructing distributed systems. We described its operational semantics and gave case studies to illustrate how today's error-prone manual work can be replaced by higher-level constructs. DS2's core (Figure 1) and user-directed scheduler are operational, and we are currently developing a front-end to parse Akka actor code that emits DS2 models for exploration. A deadlock detector and a linearizability checker are under design. As more and more developers are designing and building distributed systems, higher level languages with integrated model exploration, verification, and system synthesis will become essential (but hitherto missing) pieces of many developer's toolboxes.

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

9. REFERENCES

[1] Alloy, mit. http://alloy.mit.edu/alloy/, Retrieved 2/2716.

[2] Coq, inria. https://coq.inria.fr/, Retrieved Feb 27, 2016.

[3] Graph for scala. Retrieved Jan 31, 2016.

[4] Q-impress. http://www.q-impress.eu/wordpress/index.html, 2/27/16.

[5] Spin. http://spinroot.com/spin/whatispin.html, Retrieved Feb 27, 2016.

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

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

[8] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20, 2014.

[9] Howard Bowman, Giorgio P. Faconti, and Mieke Massink. Specification and verification of media constraints using UPAAL. pages 261–277, 1998.

[10] Sebastian Burkhardt. *Principles of Eventual Consistency*. Now, 2014.

[11] Pantazis Deligiannis, Alastair Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *PLDI*. ACM, June 2015.

[12] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, 2013.

[13] Distributed systems abstract model, 2016. http://formalverification.cs.utah.edu/parachute/reports/report.pdf, Retrieved Jan 31, 2016.

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

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

[16] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), August 1978.

[17] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. USENIX ATC '10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

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

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

[20] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

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

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

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

[24] Séverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-ide - A tool for design, analysis and implementation of component-based embedded systems. In *ICSE*, 2009.

[25] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 2003.

[26] Ryan Stutsman, Collin Lee, and John Ousterhout. Experience with rules-based programming for distributed, concurrent, fault-tolerant code. In *USENIX ATC*, Santa Clara, CA, July 2015.

[27] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Emb. Comp. Syst.*, 13(4s), 2014.

[28] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015.