Developing Correctly Replicated Databases Using Formal Tools

Nicolas Schiper Vincent Rahli Robbert Van Renesse Mark Bickford Robert L. Constable
Department of Computer Science, Cornell University

***Abstract*—Fault-tolerant distributed systems often contain complex error handling code. Such code is hard to test or model-check because there are often too many possible failure scenarios to consider. As we will demonstrate in this paper, formal methods have evolved to a state in which it is possible to generate this code along with correctness guarantees.**

This paper describes our experience with building highly-available databases using replication protocols that were generated with the help of correct-by-construction formal methods. The goal of our project is to obtain databases with unsurpassed reliability while providing good performance.

We report on our experience using a total order broadcast protocol based on Paxos and specified using a new formal language called EventML. We compile EventML specifications into a form that can be formally verified while simultaneously obtaining code that can be executed. We have developed two replicated databases based on this code and show that they have performance that is competitive with popular databases in one of the two considered benchmarks.

I. Introduction

Replicated databases form the backbone of much critical software infrastructure, including the storage tiers of large cloud services. Their availability is of utmost importance, but replication code that runs correctly in the normal case and deals correctly with each failure allowed by the system model is difficult to develop. For example, the mature and widely used MySQL database had a bug in the replication code that caused the slave to crash after a query cache filled up. This bug was experienced by many users for months until it was fixed in May 2012.

While the normal case of a replicated database can be tested fairly easily, there are many failure scenarios to consider. Failure scenarios can be hard to generate and testing them all is not usually possible. One may try to leverage model checking tools, but such tools work on models of the actual replication code and even then may not be able to search the space of failure scenarios exhaustively. Testing and model checking tools do not scale well and do not provide a proof that the code is correct.

In recent years, considerable progress has been made in the area of tools that can be used to generate executable code with proven correctness guarantees. An excellent example is seL4 [1], in which a small OS Kernel (less than 10,000 lines of code) with correctness guarantees was developed using the Isabelle/HOL proof assistant [2]. Another such example is the certified CompCert C compiler [3], which has been verified and generated using the Coq proof assistant [4]. While groundbreaking, these works left open whether or not such techniques can scale to distributed systems.

This paper shows that formal tools have now developed to a point where it is possible to build high-assurance databases using such tools. In our work we leverage the availability of various open-source databases, assuming they will fail more-or-less independently. We combine several of these into a single replicated database using a replication protocol. Based upon an early version of a replicated database [5], we have built two such replication protocols, one based on Primary-Backup [6] and one based on State Machine Replication [7]. The first protocol relies on code with formal guarantees for recovery; with the second protocol, both normal case operation (apart from transaction execution) and recovery come with correctness guarantees.

At the core of both replication protocols is a Paxos-based atomic broadcast service. In order to specify this service we used and extended EventML [8], a language to develop asynchronous distributed systems. EventML is an extension of the ML programming language [9] with event recognizers and event handlers. Finding the appropriate level of abstraction for specifying such algorithms was the result of a long-standing cooperation between formal methods and systems researchers at our lab. EventML is high-level enough so that it can be used for formal reasoning (without getting buried in implementation details), but also sufficiently low-level so that specifications can be compiled into executable forms.

The diagram presented below in Fig. 1 gives a high-level overview of our methodology. We use EventML as our specification and runtime environment, and the Nuprl proof assistant [10], [11], [12] to verify that the distributed protocols we generate are correct. Starting from a distributed protocol expressed in EventML, we generate a logical specification as well as the corresponding executable code. An automatic proof verifies that the generated code complies with the logical specification. The correctness properties of the distributed protocol are then proved semi-automatically using an interactive proof assistant.

The contributions of this paper are as follows: (1) We present a methodology to generate provably correct replicated databases; (2) we develop an improved version of the primary-backup based replication protocol [5] and present a new protocol based on state machine replication; (3) we improve two existing models of distributed computing

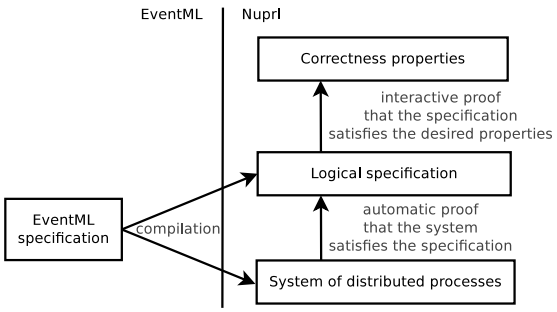

Figure 1. A high-level overview of our methodology.

as well as the EventML programming interface for ease of use and better performance; and (4) we show through experimental evaluations that both replicated databases have performance that is competitive with popular databases in one of the two considered benchmarks. This shows that building a replicated database with formal guarantees can provide good performance.

The remainder of the paper is structured as follows. In Section II we review the formal methodology that we use in our work, and then discuss the scalability of the methodology. The methodology resembles that of the one used for the seL4 OS Kernel but departs in some important ways. Section III describes the design of the replicated database and the two replication protocols that we developed. In Section IV, we evaluate the performance of the broadcast service and the two replicated databases, comparing against appropriate baselines. Section V discusses related work, and we conclude in Section VI.

II. Formal Methodology

This section describes the different components and steps involved in generating running code and proving correctness properties of that code. We make use of the Nuprl proof assistant [10], [11], [12]. Our methodology resembles the one used to verify the seL4 microkernel [1]. With seL4, the verified runnable code is obtained in three steps. In the first step, the OS kernel is specified at a high-level in the Isabelle/HOL proof assistant. This specification is refined to Haskell code before being refined a second time to a subset of C. To show the correctness of their implementation, the authors prove that the refinements are correct. As a result, the C code "satisfies" the high-level specification. Similarly, we use EventML to design protocols, and we compile these EventML specifications to Nuprl programs that can be executed by an interpreter or compiled into Lisp (which we chose for its similarities to Nuprl's programming language). We also compile these EventML specifications to Nuprl specifications and automatically prove that the Nuprl programs implement the Nuprl specifications. One key difference with seL4's approach is that we automatically generate implementations from specifications and automatically prove that they satisfy the corresponding specifications.

We have used this methodology to develop a Paxos based total order broadcast service. Because describing the details of this service is rather involved, we instead illustrate our methodology using an implementation of Lamport's Logical Clocks [7]. Developing this example will touch upon most of the important steps. We then describe how the method scales up to the broadcast service.

A. Overview of Methodology: Fig. 2 provides a detailed illustration of our workflow to obtain formally verified distributed programs from informal high-level specifications. High-level protocol specifications can be specifications written in English or pseudo-code. Given such a high-level specification we manually generate a corresponding EventML specification and formal correctness properties.

EventML is an ML-like [9] event-based functional programming language targeted at developing distributed protocols. EventML expressions can be seen as event recognizers and event handlers or, alternatively, as functions that receive and produce messages. We refer to EventML programs as "constructive specifications" because we can generate running code from them and use them for formal reasoning.

We have extended EventML in such a way that it now provides a workable balance between programming and proving: its level of abstraction is high-level enough so that reasoning about specifications is not impeded by low-level implementation details, while the level is low enough so that we can automatically generate executable programs. EventML's semantics is defined in terms of two models of distributed computation implemented in Nuprl: the Logic of Events (LoE) [13], [14], [15] to specify and reason about the information flow of distributed programs, as well as a General Process Model (GPM) [16] to implement these information flows[1]. Given a specification, we implemented an EventML compiler that generates both an LoE specification and a GPM program (arrows *a* and *b* in Fig. 2). To run an EventML constructive specification, the tool first compiles the EventML specification to a GPM program, which can then be executed. Therefore, EventML's operational semantics is defined in terms of GPM's operational semantics. This means that our compiler cannot be incorrect because it defines the semantics of EventML by providing mappings from EventML abstract syntax trees to LoE and GPM, whatever these mappings are.

Because debugging code by running it tends to be much cheaper than proving correctness (which can take days or even weeks), we often start by running the GPM program to test some of the most critical scenarios. However, to guarantee correctness, we have to show that (1) the GPM program complies with the corresponding LoE specification

[1]To handle distributed programs such as the replicated database discussed in this paper we extended both LoE and GPM by adding new sets of primitive constructors that are more fundamental, easier to compose, and more efficient.

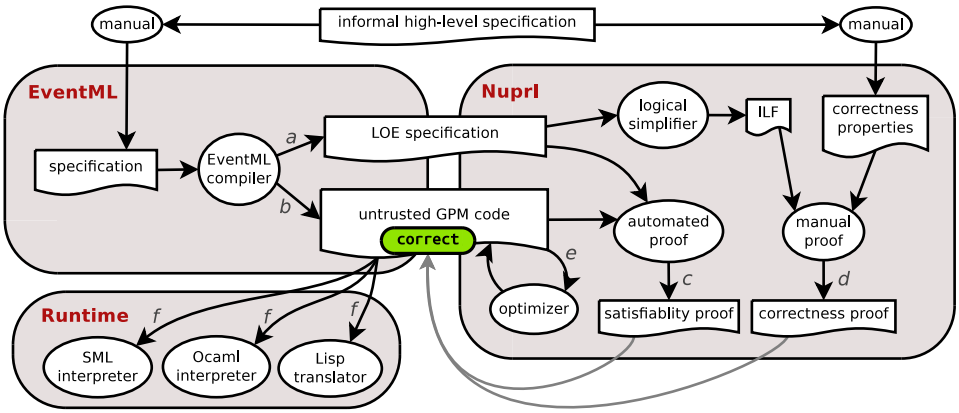

Figure 2. The workflow of our methodology.

(label *c*), and (2) the LoE specification satisfies the correctness properties specified by the high-level specification (label *d*). Both these tasks are carried out within Nuprl.

Nuprl automatically proves that the GPM program implements the LoE specification. If the proof fails that can mean that our EventML compiler has to be modified accordingly. To show that the LoE specification satisfies the desired correctness properties, the proof developer interacts with Nuprl to produce such a proof. At this point, the generated GPM program is considered to be *correct*.

Nuprl provides a program optimizer that can transform programs (for example, by unrolling recursive definitions) and prove that the optimized program is equivalent to the original one (label *e*). Running the program optimizer can significantly reduce the execution time of the GPM program, by a factor of two or more.

A GPM program is a program written in Nuprl's dependently typed functional programming language. A Nuprl program can be executed by an interpreter. In addition we have implemented a translator into Lisp (arrows labeled *f*). We assume that the interpreter and the Lisp translator are correct. Given the simplicity of the Nuprl programming language (an applied, lazy, untyped λ-calculus), it is a relatively safe assumption to make. Confidence can be further improved because EventML provides two interpreters, one written in SML and one in OCaml. We can exploit this diversity for increased reliability by running different replicas in different interpreters (Sec. III-C).

B. Nuprl Concepts: Before illustrating the methodology with an example, we introduce the concepts that are necessary to understand in more details the tasks of Fig. 2.

The EventML compiler generates an LoE specification for reasoning about the code using Nuprl. LoE is an event-based specification language, where events are abstract objects corresponding to points in space/time. The "space" aspect of an event is the physical location at which the event occurs. The "time" aspect of an event is given by a well-founded causal order relation on events similar to Lamport's [7]. Events are tagged with additional information, such as the message that triggered a receive event.

A core LoE abstraction is the *event class*, a function that takes events as inputs and outputs some information. The simplest event classes are what we call *base classes*. They play the role of event recognizers. They pattern match on the headers of the incoming messages and return the content of the message if the header matches, or nothing otherwise.

From such base classes and using LoE combinators we can build more complex event classes. For example, `X || Y` is the parallel composition of the two event classes `X` and `Y`. It recognizes events recognized by either of its two components, and handles these events in parallel to produce the outputs of both `X` and `Y`.

The EventML compiler also produces a GPM program. In GPM a process is modeled as a tail-recursive function that takes an input, produces output, and computes a new process to replace the original one. In our case, these inputs and outputs are messages.

C. Illustration of the methodology: This section illustrates our methodology using a simple running example: Lamport's logical clocks [7]. We show an EventML specification of that protocol and describe how we prove its correctness. We also present some of the program transformations we apply to the code to simplify and optimize it.

1) Specification: Logical clocks partially order events of a distributed system in a way that respects causal order. Events represent any processing done locally on one machine, as well as the sending and reception of a message. The event e_1 *happens before* the event e_2, denoted as $e_1 \rightarrow e_2$, if e_1 happens before e_2 at the same location, or if e_1 is the sending of a message and e_2 its reception. The happens before relation is transitive. Logical clocks associate a timestamp to each event e, written as $LC(e)$, and ensure that if $e_1 \rightarrow e_2$, then $LC(e_1) < LC(e_2)$.

In Lamport's protocol, each process maintains a local clock, a positive integer. When a process sends a message, the message is tagged with the local clock. When a process

```
1 specification CLK
2
3 parameter locs  : Loc Bag
4 parameter MsgVal: Type
5 parameter handle: Loc x MsgVal -> MsgVal x Loc
6
7 type Timestamp = Int
8 internal msg : MsgVal x Timestamp
9
10 import imax
```

```
11 let upd_clock slf (_,timestamp) clock =
12   (imax timestamp clock) + 1 ;;
13 class Clock = State(0,upd_clock,msg'base) ;;
14
15 let on_msg slf (value,_) clock =
16   let (newval,recipient) = handle (slf, value)
17   in {msg'send recipient (newval,clock)} ;;
18 class Handler = on_msg o (msg'base,Clock) ;;
19
20 main Handler @ locs
```

Figure 3. A simple protocol that implements Lamport clocks.

receives a message, the process clock is updated to the maximum of the process clock value and the logical clock tagged to the message, incremented by one.

Fig. 3 presents an EventML specification of Lamport clocks, which we call `CLK`. It is parametrized by the following three variables (lines 3-5): `locs` is the collection of processes of the specified distributed system, `MsgVal` is the type of the information contained in the messages exchanged by the processes, and `handle` is the function used to handle incoming messages. When a process running at location `loc` receives a message `msg`, it applies the `handle` function to `loc` and `msg`, computing a new message `newmsg` and the location of a recipient process to which `newmsg` has to be sent.

A `Timestamp` is a logical clock, represented by an integer (line 7), and a message body is a pair consisting of a `MsgVal` and a `Timestamp` (line 8). We refer to the first component of a message as its value and to the second component as its timestamp. The `msg` declaration (line 8) implicitly declares the `msg'base` event class that recognizes such messages, and extracts their content. It also implicitly declares the `msg'send` message constructor that takes a recipient `loc` and a message content `(value,timestamp)` and builds the instruction "send message with content `(value,timestamp)` to `loc`".

Next, we define the event class `Clock` in charge of keeping track of the current clock at each process location (line 13). EventML's `State` keyword defines a state machine; `Clock` specifies a state machine with initial state `0`. On each message, `Clock` invokes `upd_clock` (lines 11-12), which extracts the timestamp of the message, computes the maximum `max` between this timestamp and its current clock (given by the parameter `clock` in `upd_clock`'s definition), sets its clock to `max + 1`, and returns its new clock.

We define the event class `Handler` using the EventML *composition combinator* `o` (line 18). On each input message, if `msg'base` produces `(value,timestamp)`, and if `Clock` produces `clock`, then `Handler` produces the result of `(on_msg slf (value,timestamp) clock)`, where `slf` is the location at which the process is running. First, `on_msg` (lines 15-17) applies the function `handle` (a parameter of the system) to `slf` and to `value` to produce a new value `newval`. Then, using `msg'send`, `on_msg` builds a "send message" instruction with data `newval` and timestamped with the current clock `clock` provided by `Clock`.

Finally, the declaration `main Handler @ locs` declares that `CLK` is composed of processes implementing `Handler` and running at each location in `locs` (line 20).

2) Verification: We have to prove that $e_1 \rightarrow e_2$ implies $LC(e_1) < LC(e_2)$. Lamport specifies two conditions C1 and C2 that are enough to prove that this so-called *Clock Condition* holds [7]. C1 is the local property that the clock of a process as specified by `Clock` can only strictly increase over time. In EventML and Nuprl, this is called a *progress* property. We specify it as follows in EventML:

```
progress strict_inc on clock1 then clock2 in Clock
  with msg'base
  == clock1 < clock2 ;;
```

This says that if the clock (as specified by `Clock`) of a process is `clock1` at some event e_1, and `clock2` at some later event e_2 (i.e., the process received a message in between the two events), then `clock1` is strictly less than `clock2`. C2 is the global property that the clock of the sender of a message must be less than the clock of the receiver of that message. Our definition of `upd_clock` enforces C2.

To prove in Nuprl that `CLK` satisfies Lamport's Clock Condition, we first define the recursive "happened before" relation as follows, using LoE's causal order relation[2] `<:`

```
e1 → e2 ==r ∃e:E.
   ((e < e2)
   ∧ ((¬(loc(e) = loc(e2))) ⇒ (e2 caused by e))
   ∧ ((e = e1) ∨ e1 → e))
```

This formula says that event `e1` happens before event `e2` if there exists an event `e` in between such that: (1) `e` happens causally before `e2` in LoE; (2) if `e` and `e2` do not happen at the same location then `e2` was triggered by a message sent at `e` (this is captured by the "caused by" Nuprl abstraction); and (3) because the "happens before" relation is transitive then either `e` is `e1` or `e1` happens before `e` (this is the recursive call to the "happens before" relation).

Nuprl provides a tool to translate an LoE specification into a so-called *Inductive Logical Form* (ILF). Using the ILF, distributed system properties can be proven by induction on causal order. An ILF characterizes the outputs of the LoE specification in terms of its state and inputs. ILF formulas are called "inductive" because in general they describe

[2]LoE's causal order relation is more abstract that Lamport's "happens before" relation. In LoE, the causal ordering of the events of a system is an axiom. It might follow from more subtle and complicated communication means than exchanges of messages.

```
1.  ∀[MsgVal:ValueAllype]. ∀[locs:bag(Id)]. ∀[handle:Id → MsgVal → (MsgVal × Id)].
2.  ∀[f:headers_type]. ∀[eo:EO(f)]. ∀[e:E(eo)]. ∀[d:ℤ]. ∀[i:Id]. ∀[m:Message(f)].
3.    {<d, i, m> ∈ CLK(MsgVal;locs;handle;f)(e)
4.    ⟺ loc(e) ↓∈ locs
5.        ∧ (header(e) = ‘‘msg‘‘)
6.        ∧ has-es-info-type(eo;e;f;MsgVal × ℤ)
7.        ∧ (d = 0)
8.        ∧ (i = (snd((handle loc(e) (fst(msgval(e)))))))
9.        ∧ (m = make-Msg(‘‘msg‘‘;<fst((handle loc(e) (fst(msgval(e))))), ClockVal(MsgVal;f)@e>))}
```

Figure 4. The Inductive Logical Form of CLK.

the information computed at some events in terms of the information computed at prior events.

Fig. 4 shows CLK's ILF. This ILF is automatically generated by Nuprl using various logical simplifications and using characterizations of the LoE combinators. It says that in CLK, an event e results in message m being sent to location i (line 3) iff[3]: event e happens at one of the locations in locs (line 4); ‘‘msg‘‘ is the header of the message that triggered e to happen (line 5); the type of the data contained in that triggering message is MsgVal × ℤ (line 6); the location i is computed by applying handle to e's location and to the value of the triggering message (line 8); m's value is also computed using handle, and its timestamp is the current clock, given by ClockVal(MsgVal;f)@e (line 9). ClockVal is automatically generated from the Clock event class. Because at each event Clock returns one and only one value, ClockVal is the function that takes an event and returns that value. Clock is what we call a *single-valued* event class. All our state classes are single-valued.

It turns out that the ILF presented in Fig. 4 is not explicitly inductive in the sense that it does not mention earlier events. The inductive character of that particular ILF comes from the occurrence of ClockVal. ClockVal's value at a particular event e is defined in terms of prior events, as expressed by the equality presented in Fig. 5. These equalities are generated and proved semi-automatically as required.

Using the ILF presented in Fig. 4 and the characterization presented in Fig. 5, it is then easy to prove that CLK satisfies Lamport's Clock Condition (this corresponds to task labeled *d* in Fig. 2). Fig. 6 shows a Nuprl statement of this fact. To prove this property, we assume that events can only be caused by messages with headers ‘‘msg‘‘ as specified by the hypothesis (∀e:E(eo).↑e ∈$_b$ msg'base(MsgVal;f)).

Note that the ILF transformation is optional—it is possible, though more tedious and difficult, to prove correctness using the LoE specification directly in Nuprl.

3) Process optimizations: Having proved that CLK satisfies the desired correctness properties we can now trust the GPM code generated by EventML's compiler. However, this code is often hard to read and inefficient because GPM programs are built using several combinators defined as recursive functions, leading to programs composed of several nested recursive functions. Also, event classes typically occur more than once in specifications, leading to unnecessary duplication of code.

To overcome these inefficiencies, we have built a tool that can optimize GPM programs and prove that the optimized program has the same computational behavior as the non-optimized program. Our optimizer merges nested recursive functions into one and also applies common subexpression elimination. Besides producing more efficient code, the optimized code tends to be easier to read as it is closer to what one would write by hand.

Fig. 7 shows the statement (lines 1–10) and proof (line 11) that the non-optimized GPM program generated by EventML, called CLK-program in that figure (line 1), is equivalent to the optimized code (lines 2–10) presented on the right-hand-side of the ∼ symbol (which is a bisimulation relation) as generated by our program optimizer. This program is a *distributed system generator*. It takes a location, slf (line 2), and returns the process that is meant to run at that location. If slf is not a member of locs then our process generator returns the halted process halt (line 10). Otherwise, it returns the application of the process R to the initial state 0 (line 9), the initial timestamp. The process R (lines 3–8) is a recursive function that takes a state s, a message m, and depending on whether or not the header of that message is ‘‘msg‘‘ (line 4) does one of two things: if m's header is ‘‘msg‘‘, R builds a new state s' (line 5), a collection out of messages to send in response to m (line 6), and returns the pair of the new process R(s') and the outputs out (line 7); otherwise R returns the process R(s) and an empty list of outgoing messages (line 8). Given such a pair <R(s'),out>, EventML sends the messages in out, and R(s') replaces R(s).

D. Generation and Verification of a broadcast service: Fault handling in a distributed system can be difficult because there may be disagreement among the participants about who is up and who is down. At the core of the solution is a consensus protocol that participants use to reach agreement on competing proposals for recovery actions.

Implementing such protocols is difficult even if the protocol itself is well-understood. For instance, Google describes an extension to Paxos that allows them to recover from disk corruption [17]. Unfortunately, this extension has a bug in it. A Paxos acceptor could promise one leader not to accept ballots lower than b, lose this state after a disk corruption,

[3]Variable d in Fig. 4 is a period of time the process must wait before sending the message. These delays are useful, e.g., to implement timers.

```
∀[MsgType:ValueAllType]. ∀[f:headers_type]. ∀[es:EO(f)]. ∀[e:E(eo)].
  (ClockVal(MsgType;f)@e = if e ∈b msg'base(MsgType;f)
                           then if first(e) then imax(snd(msg'base(MsgType;f)@e);0) + 1
                                else imax(snd(msg'base(MsgType;f)@e);ClockVal(MsgType;f)@pred(e)) + 1 fi
                           else if first(e) then 0 else ClockVal(MsgType;f)@pred(e) fi )
```

Figure 5. A characterization of the `Clock` event class in Nuprl.

```
∀MsgVal:ValueAllType. ∀locs:bag(Id). ∀handle:Id → MsgVal → (MsgVal × Id).
∀f:headers_type. ∀es:EO(f). ∀e1,e2:E(eo). ∀clk1,clk2:ℤ.
  ((∀e:E(eo).↑ e ∈b msg'base(MsgVal;f)) ∧ e1 → e2 ∧ clk1 ∈ Clock(MsgVal;f)(e1) ∧ clk2 ∈ Clock(MsgVal;f)(e2))
  ⇒ (clk1 < clk2)
```

Figure 6. The clock condition as stated in Nuprl.

```
1.  ⊢ ∀[MsgVal,locs,handle,f:Top]. (CLK-program(MsgVal;locs;handle;f)
2.  |    ∼ λslf.if slf∈blocs
3.  |           then let rec R(s) = run (λm.let hdr,body = m
4.  |                                     in if name_eq(hdr;``msg'')
5.  |                                        then let s'  ← upd_clock(MsgVal) slf body s in
6.  |                                             let out ← (on_msg(MsgVal;handle;f) slf body s') @ [] in
7.  |                                             <R(s'), out>
8.  |                                        else let s' ← s in <R(s'), []>)
9.  |                in R(0)
10. |           else halt)
11. BY SqequalProcProve2
```

Figure 7. An optimized implementation of `CLK`.

	EventML spec.	LoE spec.	GPM prog.	opt. GPM prog.	correctness properties	correctness proofs
CLK	79N (1H)	590N	452N	249N	73N (1H)	1A/3M (2H)
TwoThird Consensus	646N (4H)	1398N	1343N	1752N	122N (1H)	8A/6M (3D)
Paxos-Synod	1729N (2D)	2673N	2625N	3165N	97N (1H)	24A/75M (3W)
Broadcast Service	820N (2D)	1434N	1352N	1245N	418N (1H)	0A/22M (1W)

Table I
SOME STATISTICS REGARDING SPECIFICATION, VERIFICATION, CODE GENERATION OF VARIOUS MODULES. "N" STANDS FOR NUMBER OF AST NODES, "A" FOR AUTOMATICALLY PROVED LEMMAS, "M" FOR MANUALLY PROVED LEMMAS, "H", FOR HOURS, "D" FOR DAYS, AND "W" FOR WEEKS.

and subsequently accept lower ballots. Such bugs cannot be found easily with either testing or model checking.

During the past several years, we drew from our experience in building formal method tools, programming languages, and distributed systems to develop tools and libraries of definitions and lemmas that offer the right level of abstraction to build formally verified distributed algorithms. We have learned from our experiences that it is best to adapt logical methods to follow the way system designers build and reason about systems. For example, LoE captures some design patterns that distributed system developers often use such as the delegation of a task to a sub-process. Our LoE delegation combinator allows us to specify distributed programs using a modular or "divide and conquer" method, which makes human reasoning tractable. Also, instead of deriving code in a top-down way from abstract specifications using refinements maps, our tools directly generate both code and specifications from pseudo-code specifications, which allows us to test the code before doing any formal reasoning.

We started our experiments with a consensus protocol that we call TwoThird Consensus, based on the One Third Rule algorithm [18]. Simpler than Paxos, it is a leaderless, round-based protocol that is fully symmetric. With the tools that we developed, it now takes us a few days to specify it in EventML and prove its safety properties.

Table I presents some statistics. While the CLK specification contains 79 nodes in the EventML Abstract Syntax Tree (AST), TwoThird Consensus is almost 10 times larger. We developed the CLK specification in under an hour, and a person experienced with our environment can develop the EventML specification of TwoThird Consensus from an informal specification in an afternoon. The next three columns show the sizes (in Nuprl AST nodes) of the automatically generated LoE specification, the GPM program, and the size of the GPM program after optimization. Note that we can run and test GPM programs even before we proved any properties about them.

The "correctness properties" column shows the approximate time required to formalize the correctness properties from informal specification. In each case, we were able to do so in under an hour. Proving the properties is another issue. Nuprl is a tactic-based prover in the style of LCF [19]. These hand-crafted tactics try to find a proof of a lemma automatically. This is not always successful. The last column specifies the number of lemmas that were proved automatically by Nuprl without interactive assistance, and the number of lemmas that required manual help from us. In the case

of CLK, we were able to do all this in a matter of a few hours, and it took us only three days to formally prove the safety properties of the TwoThird Consensus specification. This is in part due to the rich library of tactics and lemmas about LoE that we developed over the past several years.

At the time, we only proved safety properties. Interestingly, using manual inspection of the code we found that our initial version was not live because of a deadlock scenario. We would have found this if we had tried proving liveness properties. Fixing the specification as well as fixing the proofs turned out to be easy. About two lines of code had to change and it took less than a day to fix the proofs.

We then moved on to the multi-decree Paxos Synod protocol, the heart of the same protocol in the Paxos implementation used by Google. We started from an existing informal English specification of the protocol and its correctness properties [20]. While it took us years when counting the development of the tools and libraries of tactics and lemmas, it is now possible to formally specify Synod in a few days and verify it in a few weeks.

We made a mistake in an early version of our EventML specification of Synod. Running and testing the unverified GPM code did not reveal the bug—instead, we found the bug when we were unable to prove the safety properties of our specification. Thankfully, fixing the specification as well as the proofs turned out to be a fairly easy task (only a few lines of code had to change and it only took us a few days to fix the proofs). Not shown in the table, we also specified and verified the Multi-Paxos protocol, including the state machine replicas. Again, we made mistakes but fixing the mistakes required mostly machine time.

Finally, we built a total order broadcast service that we use in the replicated databases described in the next section. The total order broadcast service guarantees that the participating processes deliver the same messages and in the same order [21]. The total order broadcast service builds upon consensus protocols, and is able to switch between protocols for different messages. Currently, the total order broadcast service can use both the TwoThird Consensus and the Paxos multi-decree Synod consensus modules. This demonstrates that we can develop complex services in a modular fashion.

III. Building a Replicated Database

The total order broadcast service is a powerful building block for implementing various well-known fault-tolerant replication protocols such as primary-backup [6], state machine replication [7], deferred update replication [22], and chain replication [23].

We have developed a replicated database, ShadowDB, that can be configured with either primary-backup (PBR) or state machine replication (SMR). In both cases, *strict serializability consistency* is ensured [24]: to clients it appears as if transactions were executed sequentially, each at some point between the time that a client submitted the transaction and the client received the result. We assume that sequential transaction execution is deterministic.

For both primary-backup and state machine replication we assume a partially-synchronous environment [25] and crash failures only—failure detection is unreliable in either case. The participants communicate over TCP channels, and we assume that correct processes can eventually communicate with one another. ShadowDB does not currently mask bugs that lead databases to corrupt data, improperly handle concurrency, or give unauthorized access to users. Mandelbugs and Heisenbugs can cause replica states to diverge even if transactions are ordered. Dealing with these is the subject of future work.

In the case of primary-backup, we make a distinction between normal case processing and failure handling. The normal case protocol is relatively simple and hand-written (it deals with ordering transactions only). If either the primary or a backup is suspected of having failed, the total order broadcast service is used to propose and decide on a new configuration. With state machine replication, transactions are ordered by broadcasting them to the replicas using the total order broadcast service. For both primary-backup and state machine replication, the crash of all but one of the database replicas can be masked. It is worth noting that the number of tolerated database replica failures is different from the number of failures tolerated by the total order broadcast service. When using Paxos to broadcast messages in order, only a minority of failures can be tolerated, that is, if we deploy the broadcast service on three replicas, then at most one failure can be masked.

In this section, we present in more detail how ShadowDB handles transactions in both types of replication, and how we exploit diversity to improve reliability. In the presentation below, T is a transaction. Submitting a transaction T involves sending T's type and its parameters to a server. In case of failures, clients may timeout and resend transactions to the replicas. To ensure that a transaction is executed only once, each replica has to keep track of which transactions have been performed already, treating duplicates as no-ops. This can be done efficiently by recording the sequence number of the last transaction submitted by each client.

A. Primary-backup: Our primary-backup protocol handles a transaction T in the normal case similar to other primary-backup protocols [6]: (i) the client sends T to the primary database, (ii) upon first reception of T, the primary

executes and commits[4] T and forwards T to the backups,[5] (iii) the backups, upon receipt of T, also execute and commit T, and send an acknowledgment back to the primary, (iv) the primary waits to receive an acknowledgment from *all* backups before notifying the client of the transaction's success. The notification contains the transaction's result set, if any. Transaction execution is sequential both at the primary and at backups.

If one or more replicas crash or become unreachable, the protocol is unable to make progress. To detect failures, the primary and backups monitor each other by periodically exchanging heartbeats. The recovery procedure allows surviving replicas to propose new configurations that exclude suspect replicas and optionally replace them with new ones. Different replicas could propose conflicting configurations. The recovery procedure uses the total order broadcast service to ensure agreement on the sequence of configurations.

Each configuration is identified by a sequence number. The initial configuration has sequence number 0. During normal operation, the primary tags transactions with the sequence number of its configuration. Backups only accept and execute transactions if the sequence number tag matches their current configuration.

When a replica r suspects a subset of replicas to have crashed, recovery happens as follows:

1) r stops executing transactions in the current configuration, ensuring that the configuration can no longer order new transactions even if the failure suspicions are inaccurate.

2) r creates a proposal for a new configuration and broadcasts its proposal using the total order broadcast service. This message is tagged with the current configuration's sequence number and a list of replicas, where replicas that are suspected of having crashed have been removed and possibly replaced by new replicas.

3) Upon receipt of a proposal for a new configuration, a replica r' first checks if the proposal's sequence number g corresponds to its current configuration. If not, r' ignores the proposal. This way only the first proposal is considered by the replicas. To elect a new primary, replica r' sends $(g+1, seq_{r'})$ to all replicas in the new configuration (over TCP channels). Here $seq_{r'}$ is the sequence number of the last executed transaction by r'. If r' was not part of the previous configuration, r' sends $(g+1, 0)$.

4) Each replica in the new configuration waits to hear from all replicas. The new primary is the replica with the largest sequence number. In case of a tie the replica with the smallest identifier wins.

[4]The execution of the transaction may lead the operations within the transaction to request an abort. Because we assume transactions are deterministic, all replicas will abort the transaction. Databases unable to commit a transaction for other reasons are treated as crashed replicas.

[5]The primary does not extract state updates resulting from the execution because we are using unmodified databases and cannot extract state updates in general.

5) Where possible, the new primary sends missing transactions to those backups that need to catch up. If this is not possible (each replica only caches a limited number of executed transactions), the new primary sends a snapshot of its entire database.

6) Each backup sends an acknowledgment to the primary upon recovery.

7) When the primary has received an acknowledgment from all backups, the normal protocol resumes and the primary can start ordering new transactions.

If failures occur during recovery, the procedure is restarted. It is easy to show that the recovery procedure maintains two important properties:

- **Durability:** Once a client receives a transaction's answer, the execution of this transaction is permanently reflected in the state of the surviving replicas;
- **State-agreement:** In each configuration, replicas that process transactions start in the same state.

The recovery protocol sends the entire database snapshot to new replicas. This leads to a long disruption of the service when the database is large. In some cases, it is possible to overlap state transfer with the normal case protocol however. If there are at least three replicas and at least one other replica has been brought up-to-date by the primary, we can resume normal execution and propagate the database snapshot to the other backups in parallel. Such recovering replicas buffer incoming transactions until they have the full snapshot. The primary waits only for acknowledgments from replicas that have recovered. While this takes place, the maximum number of tolerated failures is one fewer than the number of recovered replicas. If failures occur during recovery, the recovery procedure is restarted but only replicas with a full copy of the database can be candidates to become primary.

B. State Machine Replication: With state machine replication, all transactions are ordered by the total order broadcast service. Executing a transaction T happens as follows: (i) the client broadcasts T to all replicas using the broadcast service, (ii) upon delivering T, each database executes and commits the transaction and sends the answer to the client, (iii) the client waits to receive the first answer.

When a replica crashes, the protocol proceeds normally with no interruptions as long as at least one replica survives. If a replica suspects another replica to have crashed, it creates a snapshot of its database and broadcasts a reconfiguration request to add a new replica and remove the crashed one. This request contains the sequence number of the last ordered transaction but not the snapshot. The new replica obtains the snapshot from the proposer (a recovering replica can potentially fetch only the transactions it is missing).

C. Diversity: While the total order broadcast service has provable correctness guarantees at the level of Nuprl programs, for the rest ShadowDB relies on an environment

that is hand-written and may contain bugs. This environment contains the Nuprl program interpreters, the operating systems, compilers, libraries, and of course the databases themselves. We employ diversity to attempt to mask correlated failures in the environment [26].

As we shall see, we deploy different databases in ShadowDB. Our implementation allows to easily plug in any JDBC-enabled database by specifying the database driver and the connection URL. We could easily go further and compile these with different compilers, and run these on different operating systems and hardware.

The total order broadcast service itself can also benefit from diversity. We currently have Nuprl program interpreters available in SML and Ocaml. Such interpreters are easy to build and test as Nuprl programs are built from few constructs with well-defined semantics. Nuprl programs can also be translated into other functional languages. Indeed, we developed a translator from Nuprl programs to Lisp. We can then compile the Lisp code using different compilers and run it in different environments.

IV. Evaluation

In this section, we evaluate the performance of the broadcast service and compare ShadowDB with primary-backup replication (ShadowDB-PBR) and with state machine replication (ShadowDB-SMR) to popular databases. We do so on a cluster of quad-core 3.6GHz Intel Xeons connected with a gigabit switch. Each machine runs Red Hat Linux 5.8 and is equipped with 4GB of memory. The hand-written part of ShadowDB is coded in Java and respectively contains 1,199 and 292 lines of code for PBR and SMR. ShadowDB and the broadcast service interact using TCP sockets.

A. The broadcast service: We measure the time needed to broadcast a message and receive a deliver notification from the broadcast service when running Paxos on three machines ($f = 1$). Each experiment consists of 500 messages broadcast per client when the service runs in the SML interpreter, and 10,000 messages per client when we run the service translated into Lisp. Each message contains 140 bytes of payload. All versions of the broadcast service implement batching, that is, multiple messages can be bundled in one Paxos proposal. In Fig. 8, we report the average delivery latency as a function of the load and we vary the number of clients broadcasting messages between 1 and 43. With one client, the non-optimized service run in the SML interpreter takes 122ms to deliver a message. The optimized version reduces this latency to 69.4ms. At their maximum throughput of respectively 27 and 65 messages delivered per second, both interpreted versions are CPU-bound.

Not surprisingly, the broadcast service greatly benefits from being translated into Lisp: only 8.8ms are required to deliver a message with one client, and the maximum throughput reaches 900 messages per second. At this

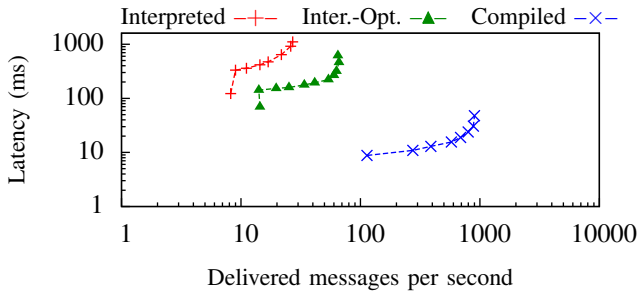

Figure 8. The performance of the broadcast service with Paxos.

throughput the execution is CPU-bound. Although compiling the service brings a significant speed-up, performance remains one order of magnitude slower than a hand-coded Paxos. However, Sec. IV-B shows that the Lisp broadcast service is fast enough to let ShadowDB with state machine replication match the performance of its primary-backup counterpart under one of the two considered benchmarks.

B. ShadowDB: We assess the performance of ShadowDB-PBR and ShadowDB-SMR using a micro-benchmark and TPC-C [27]. The micro-benchmark consists of a database of bank accounts, each having an identifier, an owner, and a balance. For the sake of diversity, we deploy each ShadowDB replica with a different in-memory embedded SQL databases: H2 1.3.170, HSQLDB 2.2.9, or Apache Derby 10.9.1. For a few setups, all replicas are deployed with the same database to make comparisons fair.

In all experiments below, group configurations contain two databases ($f = 1$); the third database is used by ShadowDB-PBR to replace the backup when we crash the primary. The broadcast service relies on the Paxos protocol and is deployed on three servers (Paxos needs three machines to tolerate one failure). We run the broadcast service in the interpreter with ShadowDB-PBR, and we rely on the Lisp service for ShadowDB-SMR. Databases are co-located with the processes of the broadcast service. Clients run on a separate machine.

Normal Case: Fig. 9(a) plots the average latency as a function of the number of committed transactions per second. We increase the load imposed on the system by varying the number of clients between 1 and 32, each submitting 35,000 update transactions. These transactions deposit money on a randomly selected account. Rows are 16 bytes in length and the database contains 50,000 rows.

We compare the performance of ShadowDB with the stand-alone H2 database (the fastest database among H2, Derby, and HSQLDB), the built-in H2 replication protocol, and MySQL replication. To make the comparison fair we deploy ShadowDB with H2 both at the primary and at the backup. ShadowDB-PBR reaches a throughput of more than 4,600 update transactions per second or 72% of the maximal throughput attained by a stand-alone H2 database.

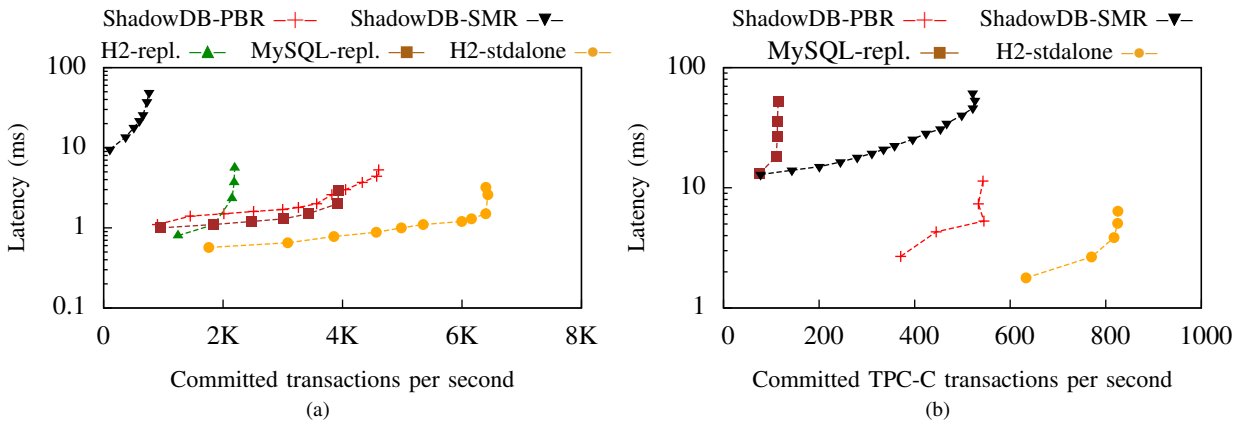

Figure 9. The performance of ShadowDB compared to other replication protocols: (a) our micro-benchmark; (b) the TPC-C benchmark.

This is the best performance attained by any of the replicated databases considered and is a consequence of ShadowDB-PBR's design: the normal case is hand-coded for efficiency. This performance comes with confidence in the code's correctness: normal case processing is simple and can easily be tested; recovery relies on code with correctness guarantees.

The H2 replication protocol quickly reaches its maximum throughput as can be seen in Fig. 9(a). This happens when contention is too high and transactions timeout when trying to lock the database table (H2 does not offer row-level locks). comparison, ShadowDB-PBR executes transactions sequentially at each replica, thereby avoiding this problem, and commits transactions in batches, to provide high throughput. The in-memory storage engine of MySQL only provides table locking and thus suffers from a similar issue. The maximum throughput attained is 3,900 transactions per second. Adding more clients results in even higher contention and lower overall throughput. We also benchmarked MySQL replication with an InnoDB table and synchronous writes turned off. Since InnoDB uses row-level locks, this lowered the abort rate, but the maximum throughput reached is lower than with an in-memory table. ShadowDB-SMR reaches a maximal throughput of 760 transactions per second and is the slowest replicated database under the micro-benchmark. At this throughput, transaction execution consumes a significant amount of CPU and this prevents the Lisp broadcast service from reaching its maximal throughput since databases and Paxos processes are co-located.

In Figure 9(b) the same databases are compared using the TPC-C benchmark configured with 1 warehouse. We report the average transaction execution latency, considering all five TPC-C transaction types, as a function of the load. Experiments consist of between 1 and 10 clients, each submitting 3,000 TPC-C transactions. H2 replication suffers from contention on the table locks and can only sustain a maximum of 62 TPC-C transactions per second; the curve is therefore omitted from the graph. ShadowDB-PBR reaches a maximum throughput of 550 transactions per second, or 66% of the maximum throughput of a standalone H2 database. Interestingly, ShadowDB-SMR provides a similar maximum throughput of 526 transactions per second. Recall that with ShadowDB-SMR, all but transaction execution comes with correctness properties. This shows that using formal methods to build replicated databases is not only feasible but it can also provide good performance.

For TPC-C, we run MySQL with the InnoDB storage engine, sufficient buffer space to hold the entire database in memory, and synchronous disk writes disabled—the memory engine provides lower performance than InnoDB due to operations on indices such as "less than" and "order by" that are not optimized. TPC-C transactions involve several round-trips between the client and the database for each of the five transaction types. The ShadowDB-PBR and ShadowDB-SMR replicas execute the transactions in the same JVM as the database, which lowers latency and improves throughput significantly compared to running them in separate JVMs.

Recovery: Fig. 10(a) illustrates an execution of ShadowDB-PBR using the micro-benchmark where we crash the primary (with ShadowDB-SMR, a crash of a replica is transparent as long as one replica is functioning). We plot the instantaneous throughput of committed transactions as a function of time. The experiment consists of 10 clients with H2 on the primary, HSQLDB on the backup, and Derby on the spare backup. After 15 seconds of execution we crash the primary, and 10 seconds later the backup detects this crash (detection time is configurable). The new group configuration is delivered about 69ms after its broadcast, and the remaining of the recovery protocol, including state transfer, takes 3.8 seconds (the database contains 50,000 tuples, each 16 bytes long). At time 40 seconds clients resume their execution.

Fig. 10(b) presents the time it takes to transfer the database state from one replica to another. With ShadowDB-PBR, this happens when a crash has occurred and the primary transfers its state to the backups. With ShadowDB-SMR, a state transfer happens when we add a replica to the group using the broadcast service (possibly long after a crash occurred). State transfer consists in selecting the rows of each table, sending the rows in batches, and inserting

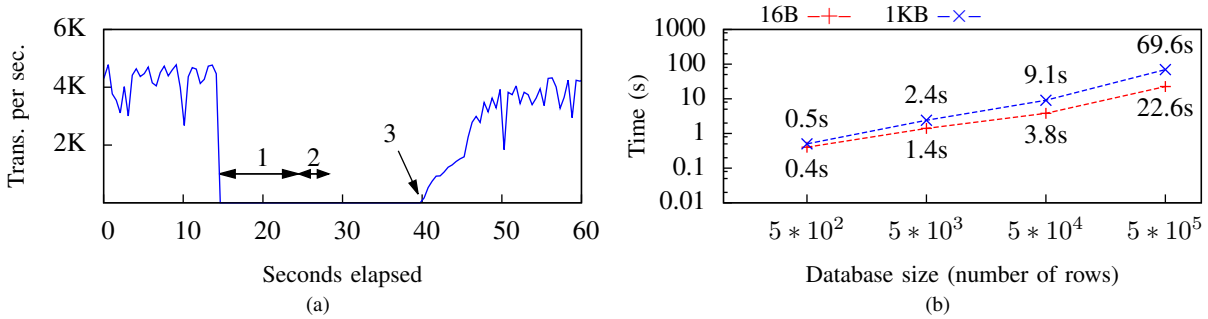

Figure 10. An execution with a crash of the primary (a) (1: crash detection in 10 sec, 2: group reconfiguration and state transfer in 3.8 sec, 3: clients resume their execution) and the overhead of state transfer (b).

them in the corresponding table at the destination replica. We consider rows of 16 bytes and 1 kilobyte with respectively 3 and 4 columns, and a number of rows varying from 500 to 500,000. For both row sizes, the batch size was chosen such that it would be close to 50 kilobytes in serialized form. Our state transfer technique allows to do state transfer with any JDBC-enabled database but is at best one order of magnitude slower than what the network can accomplish. In all experiments, row insertion speed constitutes the bottleneck of state transfer.

With a TPC-C database configured with 1 warehouse, or the equivalent of about 100MB of data, state transfer takes 54.5 seconds. In the case of TPC-C, serializing table rows has a higher overhead compared to our micro-benchmark because tables have many more columns (serialization overhead is proportional to the number of table columns). As a consequence, serializing the equivalent of 100MB of data takes 77% of the time it takes to transfer a database of 500MB with rows of 4 columns (the state transfer time for 500,000 rows of 1KB in Fig. 10(b)).

V. Related Work

Our methodology resembles the one of seL4, the first machine-verified operating system kernel [1]. Both our work and seL4 start with an informal specification. From the informal specification, formal correctness properties are derived, as well as a specification of the system. In the case of seL4, the formal correctness properties are specified in Isabelle/HOL [2], and the system specification in Haskell [28]. For our work, we used Nuprl and EventML respectively. In both cases, the system specification is translated into a specification in the respective formal environment, and a manual correctness proof is performed to show that the specification satisfies the desired correctness properties.

Our methodologies depart from there. For seL4, the executable code was handwritten in a subset of C and has to be related to the specification using a refinement mapping in Isabelle/HOL. Because EventML operates at a different abstraction level than Haskell, we were able to generate executable Nuprl code directly from the EventML specification and automatically prove the correspondence to the LoE specification. Also, exploiting Nuprl's Inductive Logical Form translator, we were able to prove correctness of distributed system properties by induction on causal order.

Another similar approach is taken by the Formally Verifiable Networking (FVN) project [29]. They use Network Datalog (NDlog), a variant of Datalog [30], to specify routing protocols in a declarative fashion. In our system we use EventML. An NDlog specification can be translated into axioms of PVS [31], and correctness proofs can be carried out interactively. This corresponds to carrying out proofs at the level of LoE. In FVN, it is also possible to translate PVS axioms into NDlog. Within the P2 framework [32], the NDlog specification is compiled into a dataflow program, with dataflow elements written in C++. In our work, we generate GPM programs from EventML specifications. P2 has been extended with cardinality abstractions [33]. This technique allows formal reasoning on the effects of applications of declarative rules directly.

EventML has strong similarities to Orc [34], [35], a programming language for structured concurrent programming. Like EventML, Orc has a small set of combinators that perform basic services, and Orc expressions are similar to our event classes. Although there are formal semantics of Orc, to the best of our knowledge none of them are formalized in a theorem prover.

There are a variety of other languages to specify distributed systems concisely. Mace [36] specifications can be model-checked and translated into C++ code. MOMMIE [37] specifications can be translated to runnable code as well as to a TLA+ specification, to be verified. Nomadic Pict [38] is a language designed for programming mobile agents, with a precise semantics that allows reasoning about correctness. In none of these cases is there a verifiable link between the correctness of the specification and that of the generated code that is executed.

VI. Conclusion

We presented our methodology to build highly-available databases using new formal tools that allow the generation

of correct distributed protocols. Based on a total order broadcast service whose code comes with correctness properties, we built two replicated databases: one is based on primary-backup replication and the other is based on state machine replication.

Primary-backup replication offers performance similar or superior to the popular MySQL database. Although not as fast as primary-backup replication on all benchmarks, we showed that the state machine replication protocol provides similar peak throughput on the TPC-C benchmark. With state machine replication, normal case operation relies on the broadcast service, and a larger proportion of the code comes with correctness guarantees. This shows that building replicated databases with formal guarantees is not only feasible but it can also provide good performance.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *SOSP'09*. ACM, 2009, pp. 207–220.

[2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[3] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL'06*. ACM, 2006, pp. 42–54.

[4] "The Coq Proof Assistant," http://coq.inria.fr/.

[5] N. Schiper, V. Rahli, R. V. Renesse, M. Bickford, and R. L. Constable, "ShadowDB: A replicated database on a synthesized consensus core," in *HotDep'12*, 2012. [Online]. Available: http://nuprl.org/KB/show.php?ID=696

[6] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed systems (2nd Ed.)*, S. Mullender, Ed. New York, NY: ACM Press/Addison-Wesley Publishing Co., 1993.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[8] V. Rahli, "Interfacing with proof assistants for domain specific programming using EventML," presented at UITP 2012.

[9] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation.*, ser. LNCS. Springer-Verlag, 1979, vol. 78.

[10] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing mathematics with the Nuprl proof development system*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[11] C. Kreitz, *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*, Cornell University, Ithaca, NY, 2002, www.nuprl.org/html/02cucs-NuprlManual.pdf.

[12] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran, "Innovations in computational type theory using Nuprl," *J. Applied Logic*, vol. 4, no. 4, pp. 428–469, 2006.

[13] M. Bickford, "Component specification using event classes," in *Component-Based Software Engineering, 12th Int'l Symp.*, ser. LNCS, vol. 5582. Springer, 2009, pp. 140–155.

[14] M. Bickford and R. L. Constable, "Formal foundations of computer security," in *NATO Science for Peace and Security Series, D: Information and Communication Security*, 2008, vol. 14, pp. 29–52.

[15] M. Bickford, R. L. Constable, and V. Rahli, "Logic of Events, a framework to reason about distributed systems," in *Languages for Distributed Algorithms Workshop*, Philadelphia, PA, 2012. [Online]. Available: http://www.nuprl.org/documents/Bickford/LOE-LADA2012.html

[16] M. Bickford, R. L. Constable, and D. Guaspari, "Generating event logics with higher-order processes as realizers," Cornell University, Tech. Rep., 2010.

[17] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC'07*. Portland, OR: ACM, 2007, pp. 398–407.

[18] B. Charron-Bost and A. Schiper, "The Heard-Of model: computing in distributed systems with benign failures," *Distributed Computing*, vol. 22, no. 1, pp. 49–71, 2009.

[19] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: a mechanised logic of computation*, ser. LNCS. Springer-Verlag, 1979, vol. 78.

[20] R. V. Renesse, "Paxos made moderately complex," Cornell University, Tech. Rep., 2011.

[21] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[23] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04*. USENIX Association, 2004, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251261

[24] C. Papadimitrou, "The serializability of concurrent updates in databases," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[25] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[26] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *FTCS'77*. Los Alamitos, CA: IEEE Computer Society Press, 1977.

[27] "Transaction processing performance council benchmark c http://www.tpc.org/tpcc/."

[28] S. Jones, *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[29] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," in *HotNets*. ACM SIGCOMM, 2009.

[30] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison Wesley, 1995.

[31] S. Owre, N. Shankar, and J. Rushby, "PVS: A prototype verification system," in *CADE 11*, Saratoga Springs, NY, Jun. 1992.

[32] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 75–90, Oct. 2005.

[33] J. A. Pérez, A. Rybalchenko, and A. Singh, "Cardinality abstraction for declarative networking applications," in *CAV'09*. Springer-Verlag, 2009, pp. 584–598.

[34] D. Kitchin, W. R. Cook, and J. Misra, "A language for task orchestration and its semantic properties," in *CONCUR 2006 - Concurrency Theory, 17th Int'l Conf.*, ser. LNCS, vol. 4137. Springer, 2006, pp. 477–491.

[35] D. Kitchin, A. Quark, W. R. Cook, and J. Misra, "The Orc programming language," in *Formal Techniques for Distributed Systems*, ser. LNCS, vol. 5522. Springer, 2009, pp. 1–25.

[36] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: language support for building distributed systems," in *PLDI'07*. ACM, 2007, pp. 179–188.

[37] P. Maniatis, M. Dietz, and C. Papamanthou, "Mommie knows best: systematic optimizations for verifiable distributed algorithms," in *HotOS'11*. USENIX Association, 2011, pp. 30–30.

[38] P. Sewell, P. Wojciechowski, and A. Unyapoth, "Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation," *Trans. on Programming Languages and Systems*, vol. 32, no. 4, 2010.