Toward Domain-Specific Solvers for Distributed Consistency

Lindsey Kuper
University of California, Santa Cruz, USA
lkuper@ucsc.edu

Peter Alvaro
University of California, Santa Cruz, USA
palvaro@ucsc.edu

Abstract

To guard against machine failures, modern internet services store multiple *replicas* of the same application data within and across data centers, which introduces the problem of keeping geo-distributed replicas *consistent* with one another in the face of network partitions and unpredictable message latency. To avoid costly and conservative synchronization protocols, many real-world systems provide only *weak* consistency guarantees (e.g., eventual, causal, or PRAM consistency), which permit certain kinds of disagreement among replicas.

There has been much recent interest in language support for specifying and verifying such consistency properties. Although these properties are usually beyond the scope of what traditional type checkers or compiler analyses can guarantee, *solver-aided languages* are up to the task. Inspired by systems like Liquid Haskell [43] and Rosette [42], we believe that close integration between a language and a solver is the right path to consistent-by-construction distributed applications. Unfortunately, verifying distributed consistency properties requires reasoning about transitive relations (e.g., causality or happens-before), partial orders (e.g., the lattice of replica states under a convergent merge operation), and properties relevant to message processing or API invocation (e.g., commutativity and idempotence) that cannot be easily or efficiently carried out by general-purpose SMT solvers that lack *native* support for this kind of reasoning.

We argue that domain-specific SMT-based tools that exploit the mathematical foundations of distributed consistency would enable both more efficient verification and improved ease of use for domain experts. The principle of *exploiting domain knowledge for efficiency and expressivity* that has borne fruit elsewhere – such as in the development of high-performance domain-specific languages that trade off generality to gain both performance and productivity – also applies here. Languages augmented with domain-specific, *consistency-aware* solvers would support the rapid implementation of formally verified programming abstractions that guarantee distributed consistency. In the long run, we aim to democratize the development of such domain-specific solvers by creating a *framework for domain-specific solver development* that brings new theory solver implementation within the reach of programmers who are not necessarily SMT solver internals experts.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Consistency; Software and its engineering → Software verification

Keywords and phrases distributed consistency, SMT solving, theory solvers

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.10

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 10; pp. 10:1–10:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern internet services store multiple *replicas* of the same application data within and across data centers. Replication aids fault tolerance and data locality: if one replica fails or is unreachable due to network partitions or congestion, another will be available in its place. In addressing those problems, though, replication introduces a new problem: the problem of keeping geo-distributed replicas consistent with one another.

In a replicated system that enforces *strong consistency*, clients cannot observe that the data has been replicated at all – but strong consistency must come at the expense of *availability*, the guarantee that every request from a client receives a meaningful response. Consider a banking application in which a user's account balance is stored in a replicated object, and where the application must maintain the invariant that a user's account balance is greater than zero. When the user withdraws from the account, to maintain the $balance \geq 0$ application invariant, the replica processing the withdrawal must make sure that any concurrent withdrawals at other replicas have also been applied to its local state before it allows a new withdraw operation to proceed. In other words, the withdraw operation requires replicas to *synchronize*.

However, not every operation against a distributed data store requires strong consistency. For instance, in our banking application, replicas need not synchronize with each other for a deposit operation. Instead, the replica that processes the deposit can report success to the user immediately and remain available to process more operations, while asynchronously updating other replicas with the new balance at some point in the future.

Although it might seem like a good thing that at least some operations can proceed without synchronization, the differing synchronization requirements for different operations hugely complicate the application programmer's task. In general, different operations on the same data may require drastically different amounts of synchronization in order to maintain application-level invariants.

1.1 Language-level tools for taming the consistency zoo

To make it easier for application developers to navigate this "consistency zoo", a number of lines of research on language-level abstractions and tools for programming against replicated data have emerged. For instance:

- *Replicated data types* (RDTs), such as conflict-free replicated data types (CRDTs) [38], replicated abstract data types [36], Cloud Types [9], and replicated lists [4], are data structures designed for replication, with an interface that limits the permissible operations to those that will ensure convergence of replicated state despite message reordering or duplication.
- *Mixed-consistency programming models* augment existing languages with sophisticated type systems, contract systems, or analyses for specifying and verifying various combinations of consistency properties. Recent representative examples of this line of work include the MixT domain-specific language for mixed-consistency transactions [32], Consistency Types [20], and the Quelea contract system [39].
- *Fault injection infrastructures* for distributed systems, such as lineage-driven fault injection (LDFI) [3], systematically inject faults, including machine crashes and network partitions, to test whether (ostensibly) fault-tolerant replicated systems maintain their claimed consistency guarantees under these contingencies. In particular, to use LDFI for implementing a distributed protocol, the programmer specifies the protocol as a program in a Datalog-like distributed programming language; the LDFI system then simulates execution of the protocol in the presence of faults and tries to determine the smallest (or most likely) set of faults that reveal bugs in the program.

All of these approaches try to lift the question of whether a given program upholds a particular application-level correctness property to the level of the programming language. Language-level tools for ensuring program correctness most often manifest as type systems or program analyses that statically (and conservatively) rule out badly-behaved programs. But traditional type systems and analyses are usually not expressive enough to statically rule out programs that violate consistency properties. The desire to be able to enforce program properties that are richer than those that can be enforced by traditional type checkers or compiler analyses has led to a proliferation of work that relies on extending programming languages with SAT or SMT solvers such as Z3 [15], CVC4 [5] or MathSAT [8]. Indeed, Quelea [39] and LDFI [3] both work by augmenting languages (Quelea's contract language and LDFI's protocol specification language, respectively) with solvers.

However, verifying distributed consistency properties requires reasoning about transitive relations (e.g., causality or happens-before), partial orders (e.g., the lattice of replica states under a convergent merge operation), and properties relevant to message processing or API invocation (e.g., commutativity and idempotence) that cannot be easily or efficiently carried out by general-purpose SMT solvers that lack *native* support for this kind of reasoning. For example, ensuring the correctness of certain flavors of CRDT implementations involves showing that replica states constitute a join-semilattice and the replica merge operation computes a least upper bound over this lattice. Existing general-purpose SMT solvers lack native support for reasoning efficiently about such order-theoretic properties [18]. Consequently, proving these properties in an SMT-aided language like Liquid Haskell is harder than it needs to be. Likewise, solver-aided tools like Quelea and LDFI must make use of simplifying assumptions, compromises, or hacks in order to be able to use the solver to reason about distributed programs.

1.2 Toward consistency-aware solvers and consistency-aware solver-aided languages

We see an opportunity to address all of these shortcomings while unifying existing lines of work on programming language support for consistency, replicated data types and SMT-based tools for mixed-consistency programming. In doing so, we follow the lead of long traditions of work on *high-performance domain-specific theory solvers* [22, 23] and *high-performance domain-specific languages* [12]. Specifically, we advocate the development of domain-specific SMT-based tools that bake in support for the mathematical foundations of consistency to support the implementation of language-level abstractions and tools for ensuring the correctness of distributed programs.

We aim to make the implementation and use of these tools accessible not only to systems programmers who would ordinarily implement replicated data storage systems, but to application programmers – the people who are usually most familiar with the application-level invariants that their operations on replicated data will ultimately need to satisfy. Domain-specific solvers for distributed consistency should be a double win, enabling both improved ease of use by domain experts (because the constraints to be discharged to the solver can be encoded in a more familiar way) *and* efficiency advantages over general-purpose, off-the-shelf solvers (because the solver will be able to reason about those constraints at a higher level).

An especially exciting way to make use of the considerable power of SAT and SMT solvers is by means of *solver-aided languages*[1] such as Liquid Haskell [43], which augments Haskell's type system with *refinement types* [37] which are compiled to equivalent constraints that can

[1] The term "solver-aided languages" was coined by Torlak and Bodik in their work on Rosette [41].

be discharged by an SMT solver. With the help of the external solver, Liquid Haskell can check refinement types at compile time. We believe that close integration between a language and a solver, as pioneered by systems like Rosette [42] and Liquid Haskell, is the right path to consistent-by-construction distributed applications. Consistency-aware theory solvers would be usable from existing solver-aided languages like Liquid Haskell, and they would dovetail with Rosette's support for building new solver-aided DSLs. New domain-specific solvers call for new domain-specific solver-aided languages, and we hypothesize that building consistency-aware languages on top of our proposed consistency-aware solvers would be an ideal application of Rosette.

Finally, we hope to use the development of consistency-aware solvers as a jumping-off point for a broader research agenda. Today, new theory solver implementation is considered the territory of SMT internals experts. Even though the architecture of modern SMT solvers appears to lend itself to a modular style of development in which theory solvers could be developed independently, in practice it would seem that SMT solvers are monolithic and SMT internals expertise is required for theory solver development. We aim to create a theory solver development framework, inspired by existing frameworks for rapid development of high-performance DSLs [12], to democratize the implementation of domain-specific theory solvers. Our goal is to make it possible for domain experts – including and especially distributed systems programmers – to implement their own domain-specific theory solvers that modularly extend existing SMT solvers.

The rest of this paper is organized as follows. In Section 2, we give a brief tour of the zoo of distributed consistency models and the guarantees that they do (and don't) provide, in the context of our example banking application with its $balance \geq 0$ invariant. In Section 3, we dig into three example use cases for a consistency-aware solver: efficient verification of CRDTs with Liquid Haskell, reasoning about message reorderings in LDFI, and precisely reasoning about the transitive closure of relations in the Quelea contract language. Finally, in Section 4, we consider SMT and theory solver implementation, and the broader problem of how to democratize the implementation of domain-specific theory solvers like the consistency-aware solvers we aim to build.

2 Consistency anomalies: a brief tour of the zoo

The desire for applications to provide a responsive, "always-on" [16] experience to users has motivated much work on systems that trade strong consistency for *eventual consistency* [40, 46] and high availability. Under eventual consistency, updates may arrive at each replica in arbitrary order, and replicas may diverge for an unbounded amount of time and only eventually come to agree.

Between the extremes of eventual consistency and strong consistency are a bewildering variety of intermediate consistency options [40, 30, 1, 31, 29, 34]. For instance, under *causal consistency* [1, 27], if message m_1 is sent before message m_2 (in the sense of Lamport's happens-before relation [27]), then m_1 must also be received before m_2. Among other things, this means that in our banking application, if, say, replica A processes a deposit of \$25 followed by a withdrawal of \$30, and sends a message to replica B for each operation in that order, then replica B must apply them in that order, as well. The idea is that because the deposit of \$25 happened before the \$30 withdrawal on replica A, the deposit *potentially caused* the withdrawal, and so the withdrawal must not be allowed to happen on replica B until the deposit has happened there first. This guarantee – that operations will occur in

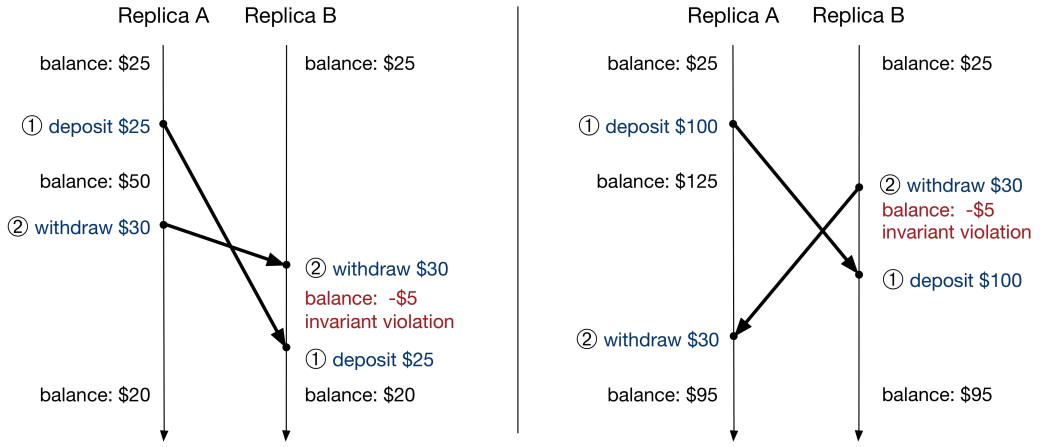

Figure 1 Examples of invariant-violating executions that are disallowed (left) and allowed (right) under causal consistency. In the execution on the left, event 2 follows event 1 in the causal order, and so the execution shown violates causal consistency. The execution on the right exhibits a different violation of the same application invariant, but in this case, one that causal consistency does *not* rule out: events 1 and 2 have no causal order, but the *balance* ≥ 0 invariant is still violated.

causal order – is enough to rule out a number of consistency-related anomalies, and suffices for many applications. The execution shown in Figure 1 (left) shows a violation of the *balance* ≥ 0 invariant in our bank application that would be ruled out by causal consistency.

For many applications, though, causal consistency is not enough in general. For our banking application, if a deposit takes place on replica A and a *concurrent* withdrawal takes place on replica B, then causal consistency says nothing about the order of the two operations. Enforcing the *balance* ≥ 0 application invariant demands a stronger consistency guarantee. Figure 1 (right) shows such an execution, which respects causal consistency but nevertheless violates the *balance* ≥ 0 invariant.

The execution in Figure 1 (left) in fact also violates a slightly weaker form of consistency, known as PRAM or FIFO consistency [30]. Under PRAM consistency, if two operations take place in a particular order on a given replica, then on any other replica *on which both operations take place*, the first operation must take place first. Figure 2 shows an execution allowed under PRAM, but ruled out by causal consistency. A withdrawal originating on replica B is delivered with no problem at replica A, but at replica C, it causes an invariant violation because a causally earlier deposit has not yet arrived. On the other hand, even PRAM consistency may be overkill for the deposit operation: if two deposits are delivered in different orders on different replicas, then we have violated PRAM (and causal) consistency, but no application invariant is violated, and we have saved the potentially substantial cost of having to synchronize between replicas.

Clearly, choosing the consistency guarantee for each operation that will enforce exactly as much synchronization as is necessary between replicas – but no more than that – can be a daunting task for programmers, even when the data store or stores offer only two or three consistency options to choose from [32, 39]. When there are more choices, the job only gets harder. For example, *read-your-writes* consistency [40] occupies a space between PRAM and weak consistency and is incomparable with eventual consistency. A recent survey paper [45] catalogues over 50 distinct notions of consistency from the literature, ordering them by their semantic strength. Figure 3 illustrates a small slice of this consistency hierarchy.

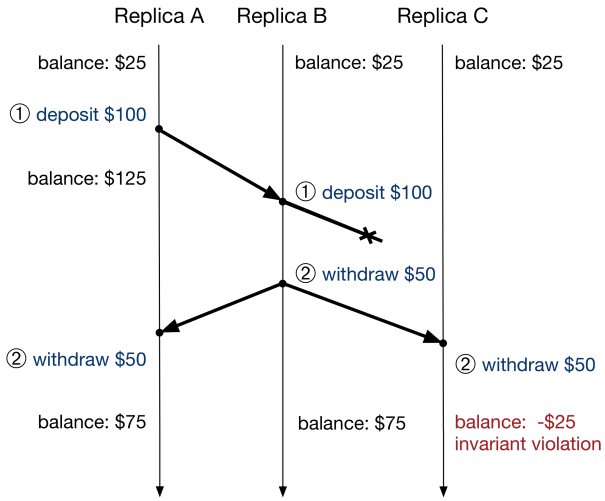

Figure 2 An invariant-violating execution allowed by PRAM consistency, but disallowed by causal consistency. Here, replica B attempts to send event 1 to replica C, but the message is lost or delayed in transit. Meanwhile, the (causally later) event 2 arrives at replica C.

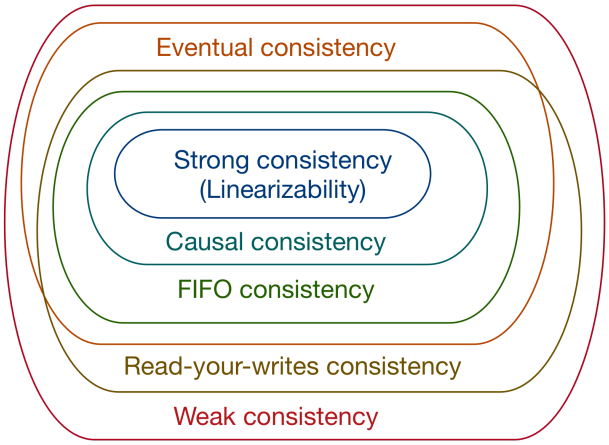

Figure 3 Some popular models of distributed consistency, ordered by strength: smaller regions admit fewer executions.

2.1 Discussion

The current state of the art of language-level support for distributed consistency discussed in Section 1.1 can help to address some of the difficulties that arise when navigating the consistency zoo. For example, if we programmed our banking application with a mixed-consistency programming system like Quelea, we might be able to obtain assurance that deposit operations may proceed without synchronization, and a warning that even casual consistency is not sufficient to ensure that concurrent withdrawal operations are safe. Alternatively, we might use RDTs to ensure that replicas eventually converge to the same (possibly negative) balance after all operations are applied, and we could use LDFI to ensure that the effects of these operations remain durable in the face of message loss and machine crashes.

In principle, we could even combine these approaches in the same system. For instance, a single solver-aided language could both identify which pairs of operations (e.g., deposits and withdrawals that witnessed them) must be causally ordered, as in the Quelea contract language, *and* ensure that a particular implementation of an RDT upholds its convergence guarantee (e.g., for concurrent withdrawals), by means of rich type specifications like those expressible in Liquid Haskell. The underlying solver for such a language would need to reason efficiently about the causality relation and partial orders, respectively. Unfortunately, general-purpose solvers are not necessarily well suited for such reasoning, as we elaborate on in the next section.

3 The case for consistency-aware solvers

What could we do if we had a solver capable of natively reasoning about distributed consistency? In this section, we motivate the need for consistency-aware solvers with three example use cases, based both on our own experiences [3] and on our reading of the literature on solver-aided language verification tools [39, 43, 44].

3.1 Efficient verification of CRDTs with Liquid Haskell

Liquid Haskell [43] is a tool that augments Haskell's type system with refinement types, calling off to an SMT solver (Z3 by default) under the hood for type checking and inference. Refinement type checking and inference is undecidable in general, but Liquid Haskell gets around this issue by employing *liquid types* [37], which require one to specify up front a fixed set of *logical qualifiers* from which refinement predicates can be built. The solver can then search over that set, returning the problem to the realm of decidability.

Liquid Haskell can be used to verify the commutativity, associativity, and idempotence of functions – all properties that one would want to guarantee about operations that modify replicated data structures. These are, in fact, exactly the properties that *must* hold of the merge operation in CRDTs. Ensuring that an RDT implementation is correct amounts to showing that certain order-theoretic properties (e.g., that replica states constitute a join-semilattice and the replica merge operation computes a least upper bound over this lattice) hold over program state. Unfortunately, these properties are difficult to verify in practice. Even just specifying the desired RDT behavior is a hard problem in itself [10, 6], and to our knowledge, the only *mechanized* proofs of correctness of RDT implementations have been done in the context of a theorem prover [19, 48], rather than in a solver-aided language like Liquid Haskell that might be usable for real implementations.

In order to verify such properties in Liquid Haskell, though, one would need to make use of its recently added *refinement reflection* mechanism [44], which goes far beyond what traditional refinement types can express and which is only well understood by a handful of experts. Furthermore, checking that the ordering laws hold for concurrent sets, for example, took 40 seconds and hundreds of SMT queries [44]. What if, by hooking up Liquid Haskell to a domain-specific solver with built-in knowledge of ordering constraints, we could get that 40 seconds down to 4 seconds or 0.4 seconds? Doing so would allow for fast, interactive verification in the REPL, in the way that Haskell programmers are used to interacting with the type checker.

3.2 Reasoning about message reorderings in a lineage-driven fault injection tool

Lineage-driven fault injection (LDFI) [3] is a methodology for testing distributed systems for fault tolerance by systematically causing certain messages to be dropped. LDFI makes use of the concept of *data lineage* – that is, the combination of data and messages that led to a particular successful execution outcome – to make decisions about which message omissions would be most likely to reveal bugs.

However, the existing LDFI implementation [3] does not consider message reorderings at all. Rather, it assumes a fixed, deterministic message ordering for a successful execution and then uses a SAT solver to exhaustively produce possible ways to make the execution fail. Yet many real-world bugs in distributed systems [47, 28] are triggered by a combination of message omission faults with *message races*, in which (for example) messages delivered in a unexpected order compromise a system's response to a fault event. In order to bound the (already astronomically large, due to the combinatorial explosion of possible faults) space of possible executions to consider, LDFI must assume that messages are delivered in a fixed, deterministic order. Hence bugs that are triggered by combinations of faults and message races could be missed.

A naive solution that, for each combination of faults selected by LDFI to inject, exercised all possible message delivery orders would be intractable for anything but small systems. Of course, in a great many cases, the order in which messages are delivered to a particular replica has no effect on the final state or behavior of that replica, because those messages *commute* with respect to the downstream message-processing logic. A possible solution to this problem would be to reason directly about the mathematical properties of the message-processing program logic. This resembles the problem of verifying the properties of the CRDT merge operation. With the help of a solver capable of reasoning about commutativity, we might be able to prove that particular pairs of messages commute, and pay the cost of exploring a larger state space only for those pairs of operations for which we cannot find such a proof.

3.3 Precise reasoning about transitive closure in a language of consistency contracts

The Quelea programming model [39] offers an innovative approach to programming against replicated data stores that offer a mix of consistency guarantees. The idea is that programmers annotate functions that operate on replicated data with contracts that describe application-level invariants, expressed in a small contract language. For example, to enforce the $balance \geq 0$ invariant from our example banking application, the programmer can annotate the withdraw operation with a contract specifying that calls to it must synchronize with other withdraws or deposits. A given operation might be executed against several different backing stores,

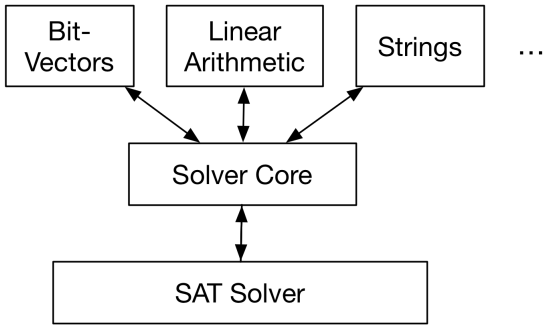

Figure 4 A lazy SMT architecture.

each of which has its own consistency guarantee, specified using the same contract language. Quelea then calls off to an SMT solver, which performs *contract classification* to determine the weakest consistency level at which each operation must be run (and therefore the backing store against which it can run with the least synchronization). Under this approach, the application programmer is able to think in terms of the application-level invariants that their code needs to satisfy, instead of having to think about the whole zoo of consistency options.

The Quelea contract language had to be carefully crafted to make contract classification decidable. For example, causal consistency is defined in terms of a transitive happens-before relation: if event 1 happens before event 2 and event 2 happens before event 3, then event 1 happens before event 3. However, transitive closure is not expressible in first-order logic, so the developers of Quelea have to make do with a relation that expresses a *superset* of transitive closure. As a result, in Quelea's notion of happens-before, some events that are actually independent are instead considered to have a happens-before relationship, leading to more synchronization than is strictly necessary to enforce causal consistency. A custom consistency-aware solver could make it possible to avoid such compromises.

4 Building a consistency-aware solver

SMT solvers allow us to check that an implementation satisfies a specification by encoding both as a formula understood by the solver, where satisfiability (or unsatisfiability) of the formula corresponds to the truth of the property we want to verify. The SMT problem is a generalization of the famous SAT problem of determining whether a Boolean formula is satisfiable. With SMT, formulas may additionally contain expressions that come from various *theories* – the "T" in "SMT". For instance, in the theory of linear real arithmetic, formulas can contain real-valued variables, addition, subtraction, and multiplication operations, and equalities or inequalities over the real numbers. Modern solvers such as Z3 come with a variety of built-in theories, such as linear arithmetic, bit-vectors, strings, and so on.

SMT solvers may operate in either an *eager* or an *lazy* manner. In the eager approach, the solver takes the SMT formula provided as input and essentially compiles it down to a Boolean SAT formula, which it can then hand off to a SAT solver. This is possible to do as long as the theory of the input formula is decidable, but in the process of compiling to the Boolean SAT formula, one may lose the high-level structure of the problem, and with it the ability to efficiently apply domain knowledge from the original theory. Modern SMT solvers therefore tend to use the lazy approach, which also involves an underlying SAT solver, but additionally involves a collection of *theory solvers* that are each specific to a certain theory. Theory solvers reason at a higher level of abstraction than the underlying SAT solver, with which they communicate via a *solver core*, as shown in Figure 4.

The efficiency advantage of the lazy approach to SMT solving is an example of the more general principle of *exploiting domain knowledge for efficiency*. For example, high-performance embedded domain-specific languages (DSLs) such as those built using the Delite DSL framework [33, 12, 7] trade off generality to gain both expressivity and efficiency; a high-level representation of programmer intent enables the compiler to do optimizations that it would not have enough information to do otherwise.

One particularly compelling recent example of a domain-specific SMT solver is the Reluplex SMT solver for verifying properties of neural networks [22, 23]. In SMT-based neural network verification, one encodes a description of a trained network and a property to be proved about it as a formula that the solver can determine the (un)satisfiability of. The tractability of the verification task depends on the ability of the solver to reason efficiently about *activation functions*, which allow networks to learn potentially extremely complex non-linear functions (indeed, without them, a network could only compute a linear combination of its inputs). The Reluplex solver enables efficient reasoning about a particular kind of activation function, the rectified linear unit (ReLU), by extending an existing theory solver for linear real arithmetic to additionally handle a new ReLU primitive and allowing SMT constraints representing ReLUs to be lazily split into disjunctions. This "lazy ReLU splitting" approach changes many verification problems from intractable to tractable and has made possible the verification of networks one to two orders of magnitude larger than the previous state of the art could handle [35].

Although we are interested in a different domain than Reluplex, we are inspired by how it illustrates the power of domain-specific solvers to bring about significant improvements in solving efficiency. It might even be possible to apply the lessons learned from the Reluplex work to our own domain of reasoning about distributed consistency. However, it is worth pointing out that a consistency-aware solver might not necessarily extend an existing theory solver with new primitives as Reluplex does. Instead, the best efficiency improvement might come from taking functionality *away* from existing theory solvers. For instance, Ge et al. [18] developed an custom solver for reasoning about ordering constraints in concurrent programs by starting with an existing theory of integer difference logic and then customizing it to remove unwanted functionality that was irrelevant to the problem at hand.

Regardless of the approach taken, a consistency-aware theory solver would need to reason about *partial orders*, in both strict (irreflexive) and non-strict (reflexive) varieties. Partial orders are essential for specifying CRDTs [38] and notions of consistency such as causal consistency [27, 1], and have been a recurring theme in previous work on concurrent and distributed programming models [2, 13, 24, 26, 25]. However, no off-the-shelf solver that we are aware of provides a built-in theory of partial orders. Indeed, off-the-shelf solvers have difficulty handling transitive relations, forcing systems to implement conservative workarounds [39]. Partial orders are ubiquitous in computer science, in areas as diverse as static analysis [14], information-flow security [17], and the semantics of inheritance in object-oriented programming [11], and we anticipate that a solver capable of efficient native reasoning about partial orders would have applications beyond our intended domain of distributed consistency.

Finally, as part of our proposed research agenda of building consistency-aware solvers, we want to consider the broader problem of theory solver development. The architecture of a lazy SMT solver, as shown in Figure 4, appears to lend itself very well to a modular style of development in which theory solvers can be developed independently. Unfortunately, in practice it would seem that SMT solvers are monolithic, and SMT internals expertise is

required for implementing new theory solvers.[2] Although much research in programming languages (and distributed systems) now makes use of SMT solvers, the solver itself is generally treated as a black box, and theory solver implementation is considered the territory of the same SMT internals experts who implement the solver core.

We argue that *programmers should not have to be SMT internals experts in order to implement theory solvers for their domain of interest.* We propose to evaluate that claim by developing a framework for implementing custom, efficient domain-specific solvers. In doing so, we hope to democratize theory solver development and make it accessible to programmers who are not SMT internals experts, in the same way that Delite aimed to democratize DSL implementation and make it accessible to programmers who are not compiler experts.

References

1. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.
2. Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.
3. Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, New York, NY, USA, 2015. ACM. doi:10.1145/2723372.2723711.
4. Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933090.
5. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.
6. Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying Eventual Consistency of Optimistic Replication Systems. *SIGPLAN Not.*, 49(1):285–296, January 2014. doi:10.1145/2578855.2535877.
7. Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/PACT.2011.15.
8. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70545-1_28.
9. Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_14.

[2] The architecture of the Reluplex solver is a case in point: the implementation was originally based on an off-the-shelf linear programming (LP) solver, but required invasively modifying the existing LP solver instead of building on top of it in a modular fashion, and the published work based on the off-the-shelf LP solver was later discarded in favor of building a new solver from scratch [21].

10 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

11 Luca Cardelli. A Semantics of Multiple Inheritance. In *Information and Computation*, pages 51–67. Springer-Verlag, 1988.

12 Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. doi:10.1145/1941553.1941561.

13 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM. doi:10.1145/2391229.2391230.

14 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973.

15 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

16 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. doi:10.1145/1294261.1294281.

17 Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976. doi:10.1145/360051.360056.

18 Cunjing Ge, Feifei Ma, Jeff Huang, and Jian Zhang. SMT solving for the theory of ordering constraints. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 287–302, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-29778-1_18.

19 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, October 2017. doi:10.1145/3133933.

20 Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM. doi:10.1145/2987550.2987559.

21 Guy Katz and Clark Barrett. Personal communication, 2017.

22 Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 97–117, 2017. doi:10.1007/978-3-319-63387-9_5.

23 Lindsey Kuper, Guy Katz, Justin Gottschlich, Kyle Julian, Clark Barrett, and Mykel J. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks. *CoRR*, abs/1801.05950, 2018. arXiv:1801.05950.

24 Lindsey Kuper and Ryan R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM. doi:10.1145/2502323.2502326.

25 Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 2–14, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594312.

26 Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze After Writing: Quasi-deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535842.

27 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

28 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 517–530, New York, NY, USA, 2016. ACM. doi:10.1145/2872362.2872374.

29 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

30 R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report no. 180, Princeton University, Department of Computer Science, 1988.

31 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. doi:10.1145/2043556.2043593.

32 Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192375.

33 Kunle Olukotun. High Performance Embedded Domain Specific Languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 139–140, New York, NY, USA, 2012. ACM. doi:10.1145/2364527.2364548.

34 Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Causal Consistency: Beyond Memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 26:1–26:12, New York, NY, USA, 2016. ACM. doi:10.1145/2851141.2851170.

35 Luca Pulina and Armando Tacchella. An Abstraction-refinement Approach to Verification of Artificial Neural Networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 243–257, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-14295-6_24.

36 Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011. doi:10.1016/j.jpdc.2010.12.006.

37 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. doi:10.1145/1375581.1375602.

38 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642.

39 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737981.

40 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

41 Emina Torlak and Rastislav Bodik. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.

42 Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594340.

43 Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 39–51, New York, NY, USA, 2014. ACM. doi:10.1145/2633357.2633366.

44 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, December 2017. doi:10.1145/3158141.

45 Paolo Viotti and Marko Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016. doi:10.1145/2926965.

46 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432.

47 Pamela Zave. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012. doi:10.1145/2185376.2185383.

48 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal Specification and Verification of CRDTs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.