Working Draft — Do Not Distribute

Stepwise Development of Streaming Software Architectures

Don Batory and Taylor L. Riché

Department of Computer Science
The University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{batory,riche}@cs.utexas.edu

Abstract

We present a Model Driven Engineering approach to explain, verify, build, and test dataflow or streaming software architectures that are parallelized for performance or availability. Component-connector models are incrementally elaborated by transformations that refine or optimize architectural designs. We re-engineered two significant case studies to illustrate the generality of our work: (1) recoverable crash fault tolerant servers and (2) join parallelizations in database machines.

1. Introduction

Model Driven Engineering (MDE) is a paradigm of increasing importance to everyday software design and development. In our experience, MDE has allowed us to focus more on problem essentials and less on accidental complexities, and has freed us from low-level details of common programming languages (Java, C#) and platforms (Windows, Linux).

Our interest in MDE arose from implementing *Recoverable Crash Fault Tolerant (RCFT)* servers that are examples of *streaming architectures* (also called *dataflow* and *pipe-and-filter architectures*) [9, 26]. The component-connector models of these servers are so complicated that we needed a way to convince ourselves and others of the correctness of the designs. We re-engineered prototypes built by domain experts as we wanted to synthesize the systems they had laboriously crafted by hand. As we are not domain experts, it was not obvious to us how their architectures worked or why they were correct. We needed a structured way to explain, verify, build, and test our versions of their systems.

We used *stepwise development (SWD)* to achieve our goals. SWD is a fundamental technique for controlling complexity. It has been successfully used in developing programs [2, 15, 16, 29] and software architectures [4, 7, 8, 10, 12, 19, 13, 23, 24, 25, 31]. As SWD of architectures is well-explored, we were surprised that prior results were inadequate to explain RCFT architectures as well as streaming architectures in other domains. The reasons are: (1) prior work focused on a bottom-up construction of systems: components and connectors were used to build progressively higher-level abstractions. In contrast, we deal with middleware mappings: we start with an executable architecture and incrementally map it to a parallel architecture with desired performance or availability properties; (2) architectural optimizations that break abstraction boundaries for purposes of efficiency and fault-tolerance are essential but absent in extant work; and (3) components may be extended with new capabilities, ports and relationships that were not present previously. This latter point is controversial as not all prior work advocates extensions; extensions are common in our approach. Without these additions, we could *not* describe, let alone verify, the systems that we were building.

We present our findings as a structured MDE approach to enhance streaming software architectures. We begin with an executable component-connector architecture as our initial model. We then transform this model by refinements (that expose hierarchical detail) and optimizations (that break encapsulation boundaries) to incrementally derive a parallel architecture. As each transformation is simple and so too is its proof of correctness, our approach is *correct by construction*, i.e., if the initial architecture is correct and its transformations are correct, the final architecture is correct.

To verify that our hand-written implementations preserve essential properties, we rely on testing. We define unit, module, and integration tests after each transformation and reuse these tests as the architecture is developed. In essence, we not only derive and verify parallel architectures incrementally, but also show how their implementations can be built and tested incrementally as well. This too we believe is new.

We present two non-trivial case studies to evaluate and demonstrate the generality of our work: (1) recoverable crash fault tolerant servers and (2) join parallelizations in relational database machines. In these studies, we reengineer prototypes created by experts to give them architectures that are designed in a principled manner. These studies illustrate the similarity of architectures in different domains, but they also reveal the fundamentals that underlie pragmatic SWD. Our work shows how non-experts (other software engineers like us) can participate and appreciate the challenges that experts face and the techniques they implicitly use. We begin with an overview of our approach.

2. Architectural Design Overview

An *architecture* is a directed graph of boxes and arrows. A box is a component; an arrow is a communication path for messages or tuples, pointing in the direction of dataflow.

A sort architecture is an elementary example. It consists of a single box *SORT* that takes a stream of tuples A as input and produces a sorted stream of tuples $sort(A)$ as output (Figure 1a). *SORT* works by reading stream A into memory, sorting its tuples, and outputing the sorted stream. *SORT* has other parameters, such as a sort key and a tuple comparison function. We elide these details without loss of generality.

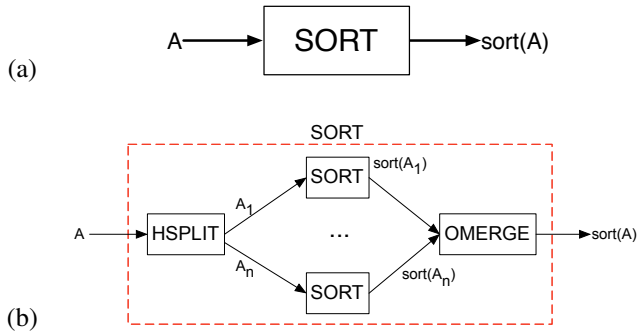

Figure 1. Sort Architecture

An *architectural transformation* or simply *transformation* is a mapping of an input architecture to an output architecture. Properties of relevance in the input architecture are preserved by the output architecture. We identify two kinds of transformations: refinements and optimizations. A transformation that exposes hierarchical implementation detail is a *refinement*. Figure 1b shows a classical database refinement of a *SORT* box that exposes how *SORT* can be parallelized: the input stream A is hash-partitioned on keys by the *HSPLIT* box, producing substreams $A_1 \ldots A_n$. All tuples in stream A_i hash to value $i \in 1 \ldots n$. Each substream is sorted on the same sort key, producing sorted streams $sort(A_1) \ldots sort(A_n)$, which are then combined by an ordered merge, producing stream $sort(A)$.

In general, boxes, arrows, or connected subgraphs can be transformed. The transformations we consider can be proven correct, but often they are simple enough that intuition suffices. (The correctness of Figure 1b is 'obvious' to database researchers, but we are unaware of a published formal proof). If the initial architecture is correct, and its transformations are correct, the resulting architecture is correct.

2.1 Testing

Most details of a system are not captured by its architecture. Although we might have a proof-of-correctness, say, for a *SORT* box, we might not know if the box's hand-crafted implementation is correct. Hence testing is needed.

Figure 2a depicts a *unit test* or *module test* of a *SORT* box as a harness that feeds input streams into *SORT* and verifies that its output is correct. When *SORT* is refined (Figure 2b), this unit test becomes an *integration test* that verifies the correctness of *SORT*'s parallel architecture. Additional unit tests are created for the *HSPLIT* and *OMERGE* boxes, so that these unit tests become integration tests when their boxes are refined. Note the unit test for the *SORT* box is also a *system test*, as it is a test for the entire parallel *SORT* architecture.

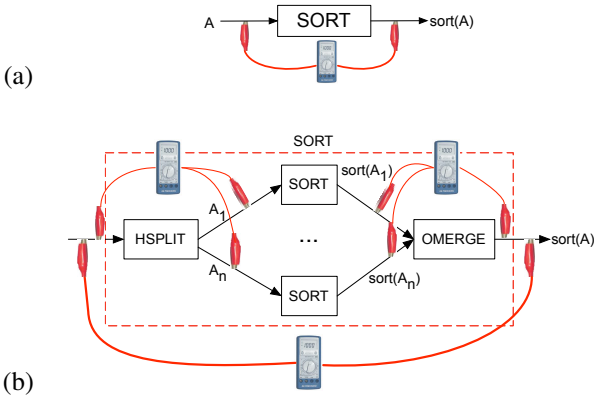

Figure 2. Test Harnesses

In general, tests are created at every level of abstraction. The validity of each test (i.e., the validity of the properties that it checks) must hold after every transformation. So as architectural details of an implementation are revealed, we verify its correctness by an accumulation of tests. Note that a test could be a unit test, integration test, or a system test depending on the level of abstraction at which it is used. In this sense, our approach has an appealing symmetry.

2.2 Optimizations

The second kind of transformation is an optimization. An *optimization* breaks encapsulations to realize non-functional properties (e.g., efficiency or fault-tolerance); it is an equivalence rewrite that does not alter design semantics. Consider the projection-sort architecture of Figure 3a. A stream of tuples A enters projection box *PROJ*, where *PROJ* removes unnecessary fields. The resulting stream A' is then sorted, yielding stream B.

Both *PROJ* and *SORT* boxes are refined individually to their parallel counterparts in Figure 3b. Both *HSPLIT* boxes partition streams using exactly the same hash function. Note that the *MERGE* box serializes substreams $A'_1 \ldots A'_n$ into stream A' and then *HSPLIT* reconstructs *these same substreams*. An optimization removes the *MERGE* and *HSPLIT* pair, as their composition is the identity map. Figure 3c shows the optimized projection-sort architecture.

Optimizations play havoc with tests, as boxes and connectors can disappear. For example, the connector between *MERGE* and *HSPLIT* in Figure 3b is absent in Figure 3c. To run tests after an optimization, it may be necessary to reintroduce eliminated boxes – *MERGE* is reintroduced to

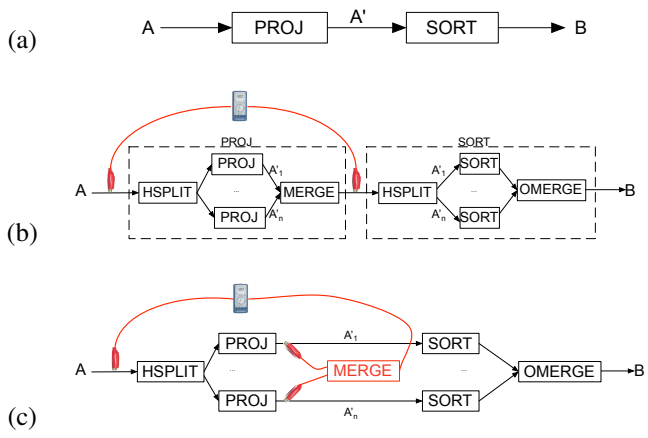

Figure 3. Optimization and Testing

run the *PROJ* unit test in Figure 3c. Reintroductions are temporary—they only belong to test harnesses.

The optimization of $PROJ - SORT$ example is an instance of a more general concept we call an *exchange*. Figure 4a shows substreams $A_1 \ldots A_n$ combined by a *MERGE* (which is part of one abstraction) and split into substreams $B_1 \ldots B_k$ by a *HSPLIT* (which is part of another abstraction). Unlike the $PROJ - SORT$ example, here the composition of *MERGE* and *HSPLIT* is *not* the identity map. An exchange reorders *stateless* computations. The key property of Figure 4a is each tuple of an A_i substream is hash-routed to a unique B_j substream. This property is preserved by exchanging (swapping) the composition order of *MERGE* and *HSPLIT* (Figure 4b): substream A_i is first *HSPLIT* into substreams $A_{i1} \ldots A_{ik}$ and then substreams $A_{1j} \ldots A_{nj}$ are combined by *MERGE* to form substream B_j for all $j \in 1 \ldots k$. Exchanges are essential in streaming architectures.

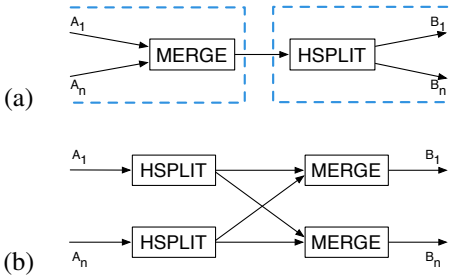

Figure 4. Exchanges

2.3 Extensions

Last but not least, it is possible for a transformation to add new capabilities and ports to a box. We say that a box X is *extended* to Y (written $X \rightsquigarrow Y$) to express this change. Extending a box is equivalent to adding one or more "features," which can be accomplished by preprocessors (e.g., `#ifdef` inclusion of extra code) or by more sophisticated means [2].

In the next sections, we show how these ideas can be used to explain, verify, build, and test designs created by domain experts from very different fields of software development.

3. Recoverable Crash-Fault-Tolerant Servers

Our work on streaming architectures stemmed from the desire to define and implement generic transformations to map a vanilla server to an RCFT server [26]. There are several such mappings in practice [14, 33]. In this section, we describe the most recently proposed mapping that is used in state-of-the-art fault-tolerant servers [9]. We begin with some client-server basics.

3.1 Basics

We consider request-processing applications with multiple clients sending requests to a server. Client requests can read or write the server's internal state, which persists across the processing of requests. For each request, the server receives the message, updates its state, and sends a response back to the client. That servers have state is important: RCFT of non-stateful servers is a *much* simpler problem.

RCFT servers have a cylinder topology representing the cyclic flow of request-response. We unroll the cylinder, as shown in Figure 5a, by breaking the seam along dotted lines. Figure 5b shows a typical architecture with clients $C_1 \ldots C_n$ and server S. $\triangleright$ denotes a serialization of requests into a single stream. $\triangleleft$ is a demultiplexer that splits an input stream into multiple output streams, one stream per client.[1]

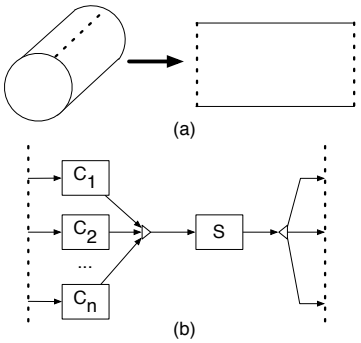

Figure 5. Unrolled Cylinder Topology

3.2 Crash Fault Tolerance

Crash fault tolerance is the ability for a service to survive a number of failures. A *failure* occurs when a box stops processing messages—no messages pass through a failed box and a failed box cannot create new messages. We assume that every box executes on a separate machine, later we relax this restriction. The rules for failure are simple: if a machine fails, all boxes on that machine fail. The failure of network boxes — $\triangleright$, $\triangleleft$, and $\bullet$ — affect a machine the same

[1] $\triangleright$, $\triangleleft$, and $\bullet$ (broadcast) are network components that model the behavior of the physical network connecting different boxes.

as pure software boxes. For example, a machine cannot process requests if its network card stops working. Failure is self-contained, meaning that failures do not propagate across machine boundaries. Further, we do not guarantee delivery of each individual network packet; retransmissions (either by the application, network protocol, or both) must be employed to deal with transient network packet loss. These are standard assumptions in CFT research.

The technical objective of CFT is to eliminate *Single Points of Failure (SPoF)* by replicating functionality [27]. An SPoF is the failure of a single box (machine) that causes the server abstraction to fail. Our initial design (Figure 5b) has three SPoFs: the server S, the serializer $\triangleright$, and demultiplexer $\triangleleft$ can fail, resulting in the failure of the server abstraction. In the following, we progressively reveal a way to eliminate SPoFs in this design.

3.2.1 Agreement Refinement

The first transformation CFT_0 adds a single node $A^{\perp}$ between the clients and server. The *agreement node* $A^{\perp}$ implements an ordered queue of messages, receiving messages from clients and passing messages one at a time to the server. $A^{\perp}$ adds nothing; it is a placeholder for subsequent refinements. Figure 6 shows the architecture after CFT_0.

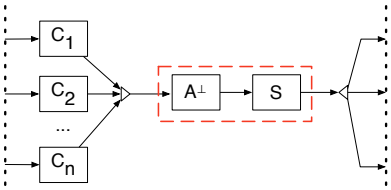

Figure 6. After the CFT_0 Transformation

3.2.2 Replication Refinements

The next transformations, CFT_{1a} and CFT_{1b}, replicate the agreement and server boxes to improve system availability, i.e., to make the server abstraction more resilient to crashes. See Figure 7.

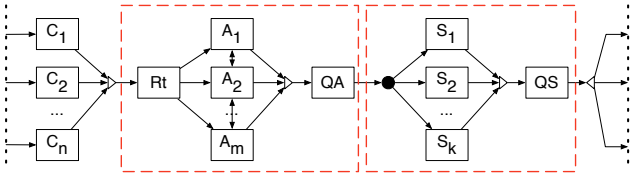

Figure 7. Fully Replicated $A^{\perp}$ and S Boxes

The replicating-agreement transformation CFT_{1a} does the following: (1) it extends $A^{\perp}$ to A (i.e., $A^{\perp} \rightsquigarrow A$) to implement an agreement protocol, (2) replicates A m-times as boxes $A_1 \ldots A_m$, and (3) adds a *routing* box, Rt, before and a serializer, $\triangleright$, and *quorum* box, QA, after the replicas. Rt forwards incoming client messages to a subset of A replicas, where the actual subset is determined by the agreement protocol. The replicas $A_1 \ldots A_m$ implement an *agreement protocol* that guarantees a quorum-decided linear order in which client messages should be processed [20, 21]. A replicas communicate with each other to determine what should be the next client message to process. Each A replica votes, sending its "next" message to the quorum box QA. QA forwards a message to the server if it receives identical messages from a sufficient number of replicas. CFT_{1a} maintains the behavior of and interface to $A^{\perp}$.

Now consider the replicating-server transformation CFT_{1b}. It replicates the server S k-times, indicated by boxes $S_1 \ldots S_k$, and adds three new boxes: $\bullet$, $\triangleright$, and QS. Box $\bullet$ broadcasts an incoming request to each server replica. Replicas receive the message, updates their state, and send responses. $\triangleright$ serializes all responses and box QS collects a quorum of identical responses. Once QS receives matching responses from a sufficient number of S replicas, QS transmits the response to the client, thus maintaining the abstraction of a single server.[2]

In summary, A is replicated m-times and S is replicated k-times. The specific values of m and k depend on the number of faults f to tolerate and the agreement protocol. Common assignments set $m = 2f + 1$ and $k = f + 1$. Yin et al. explain this difference in replica count by noticing that the quorums necessary to prove the protocols correct are larger for messages coming from the agreement portion (QA) than for messages coming from the execution (QS) [32]. Since the server, on average, requires more computational resources than agreement, and thus needs a more powerful—and more expensive—machine, using fewer server replicas in the architecture is a desirable property.

Transformations CFT_{1a} and CFT_{1b} are commutative as the order in which they are applied does not matter. However, both transformations must be applied to guarantee that the system will tolerate the failures of up to f server machines and up to f agreement machines. Figure 7 is the result of applying these transformations to Figure 6.

3.2.3 Exchanges

Our original architecture of Figure 5b had three SPoFs; our current design in Figure 7 has eight! Although there are more SPoFs, we show these are easier to remove.

In Figure 8 we dissolve existing abstraction boundaries and identify three new abstractions that contain two or three boxes, all of which are SPoFs. For example, the left-most abstraction contains SPoF boxes $\triangleright$ and Rt. The middle abstraction contains three SPoF boxes ($\triangleright$, QA, and $\bullet$). And the right-most abstraction contains three SPoF boxes ($\triangleright$, QS, and $\triangleleft$). In the following paragraphs, we explain how exchanges revise the implementations of each abstraction to implementations that have no SPoFs.

[2] Appreciate the need for an agreement protocol: a consistent order of requests is essential for correct behavior because if replicated servers process client requests in different orders, server replica states will diverge and responses from different servers for a single client request would be inconsistent. This inconsistency breaks the one-correct-server abstraction.

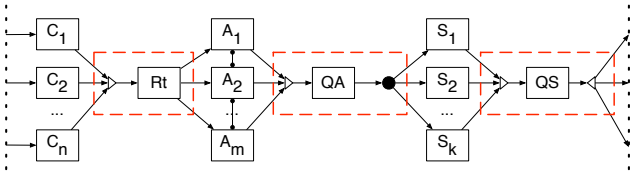

Figure 8. Abstractions with Multiple SPoFs

Consider the left-most abstraction of Figure 8, which we show again in Figure 9a and comprises the sequence $(\triangleright, Rt)$. The CFT_{2a} exchange swaps the order of these boxes (see Figure 9b). Box Rt is replicated once for each client and $\triangleright$ is replicated once for each A_i. Instead of serializing all requests and then routing, we route client requests immediately and serialize requests before each A replica. The property that each client request is sent to a subset of A replicas is preserved by CFT_{2a}. As expected, the interface of n input channels and m output channels is maintained. But now, Rt and $\triangleright$ boxes are no longer SPoFs in Figure 9b.

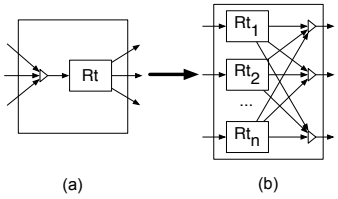

Figure 9. The CFT_{2a} Rewrite

Now consider the middle abstraction of Figure 8, which is replicated in Figure 10a and consists of the box sequence $(\triangleright, QA, \bullet)$. This sequence separates A replicas and S replicas in Figure 8. Transformation CFT_{2b} is a pair of exchanges that modify the order $(\triangleright, QA, \bullet)$ to $(\triangleright, \bullet, QA)$ and then to $(\bullet, \triangleright, QA)$. That is, instead of taking a quorum of responses from A replicas and broadcasting the result, we broadcast the results of all A replicas and take a quorum at each server replica. The property that a quorum-decided request from replicated A boxes is delivered to all server replicas is preserved by CFT_{2b}. But now, there are no SPoFs in Figure 10c.

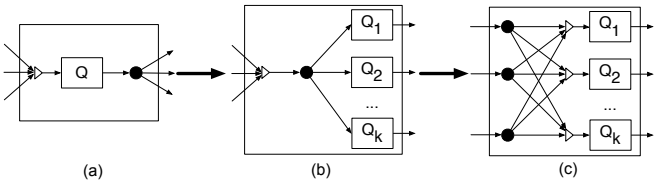

Figure 10. The rewrite CFT_{2b}.

Similarly, transformation CFT_{2c} is a pair of exchanges that is applied to the right-most abstraction in Figure 8, which is the box sequence $(\triangleright, QS, \triangleleft)$ that separates S replicas from client boxes. CFT_{2c} modifies the order $(\triangleright, QS, \triangleleft)$ to $(\triangleright, \triangleleft, QS)$ and then to $(\triangleleft, \triangleright, QS)$. That is, instead of taking a quorum of server responses and forwarding a single response to a client, server replicas send their responses to the client, and the client takes the quorum of responses. This preserves the property that quorum-decided responses from replica S boxes are received by a client box. Doing so no longer makes $\triangleright$, QS, and $\triangleleft$ SPoFs.

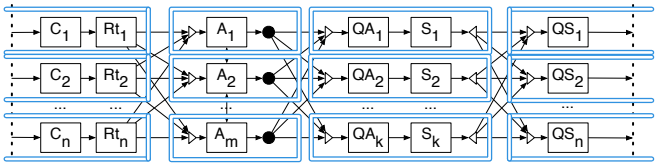

Figure 11. Exchanges and Machine Boundaries

Figure 11 shows the result of applying transformations CFT_{2a}-CFT_{2c} to Figure 7. Further, we relax an earlier assumption of limiting one box per machine to expose *machine boundaries*, i.e., enclosures that assign multiple boxes per machine, as indicated by double-walled rectangles in Figure 11. Note that boxes $\triangleright$, QS_i, C_i, and Rt_i execute on client machine i. As stated earlier, if a machine fails, all of its boxes fail. The server abstraction fails if and only if more than f server machines or f agreement machines fail.

3.3 Recovery

Up to this point, failure is permanent. If a machine or a box fails, it no longer responds and has no hope of ever responding again. After the CFT transformations, our architecture supports f failures of A replicas and f failures of S replicas. When either limit is exceeded, the entire system is in a failed state and we violate our one-correct-server abstraction.

In this section we describe a series of transformations that add failure recovery to our design. Failure recovery limits the situations where a client sees an unresponsive server abstraction. A machine that crashes but recovers can be considered correct, albeit slow. The other machines will continue to make progress, though the recovering machine may not be able to catch up immediately. If other machines crash, the remaining fast servers may have to wait for the recovering servers to catch up, at which point the system can continue to serve client requests.

3.3.1 Recoverable Server Extension

Transformation RC_0 extends a server box S without failure recovery to a server box RS with recovery ($S \rightsquigarrow RS$). Failure recovery includes both logging and checkpointing, and the ability to load a checkpoint and replay the log upon restart. Most modern request-processing servers implement this functionality (two open-source examples are Hadoop [14] and Zookeeper [33]). The extension $S \rightsquigarrow RS$ is not automatic; server programmers must manually extend their code to correctly implement recovery because checkpointing accesses application-specific data structures. Figure 12 shows the result of applying RC_0 to Figure 5b.

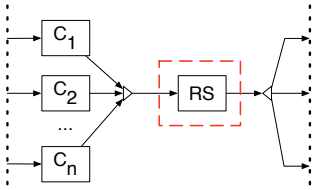

Figure 12. After Transformation RC_0

3.3.2 Recoverable Agreement Refinement

Recall that transformation CFT_0 introduces an $A^{\perp}$ box in front of S. $A^{\perp}$ is part of the greater abstraction of one correct server. For this entire abstraction to be recoverable, all of its stateful internal boxes must be recoverable, thus $A^{\perp} \rightsquigarrow RA$. Further, the recovery algorithm of the server RS is extended slightly, as it needs information from the RA box to fully recover. (Note: this "extra" information is not essential at this stage of design, but like the introduction of the A box, it is a placeholder for subsequent refinements to elaborate. Providing this "extra" information is part of $RS \rightsquigarrow RS'$).

We do not have to transform the other boxes in the architecture (such as $\triangleright$ or $\triangleleft$) as they are stateless; the recovery of a box that does not keep persistent state is a simple restart.

In summary, transformation RC_1 makes three changes: (1) $A^{\perp} \rightsquigarrow RA^{\perp}$, (2) $RS \rightsquigarrow RS'$, and (3) a new arrow connects RS' to RA to allow RS' to query RA for checkpoint information. Figure 13 shows the result of applying RC_1 to Figure 12.[3]

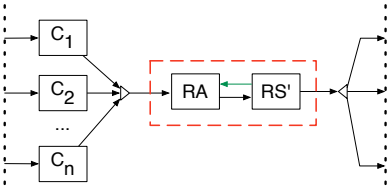

Figure 13. After Transformation RC_1

3.3.3 Replication Refinements

RC_{2a} and RC_{2b} are transformations that respectively add failure recovery to replicated agreement and server boxes. Although RC_{2a} and RC_{2b} are commutative, they are dependent on CFT_{1a} and CFT_{1b} respectively. Figure 14 shows the result of applying both transformations to Figure 13.

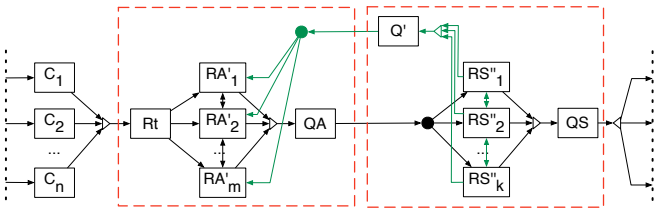

Figure 14. After the RC_{2a} and RC_{2b} Transformations

[3] We represent multiple communication channels in the same direction with just one arrow for brevity.

Agreement Replication. Once CFT_{1a} replicates agreement box RA, the transformation RC_{2a} extends replicated RA boxes to RA' by adding new message handlers such as one for receiving the current checkpoint on which the RA' replicas must agree.[4] Further, a broadcast ($\bullet$) is introduced that sends all incoming messages from the server to all agreement replicas $RA'_1 \ldots RA'_m$.

Server Replication. Once CFT_{1b} replicates server box RS', transformation RC_{2a} extends replicated RS' boxes to RS'' by adding inter-replica communication between the server replicas. The system can continue with multiple servers to make progress after another server crashes, however, the restarting server may be behind. Thus, it must fetch checkpoint state from the other server replicas to catch up.

After processing a fixed number of client messages, the agreement box asks the server for its current checkpoint. The agreement box can only accept this checkpoint if it receives a quorum of matching checkpoints from server replicas. When a server crashes and attempts to recover, it asks the agreement box for the latest checkpoint. This "Help, I need to recover!" message comes from just one server, and the agreement box does not wait for a quorum (as one will never come). A special quorum box Q' (a) takes quorums of checkpoint messages from servers and (b) immediately passes along recovery messages of failed servers.

3.3.4 Exchanges

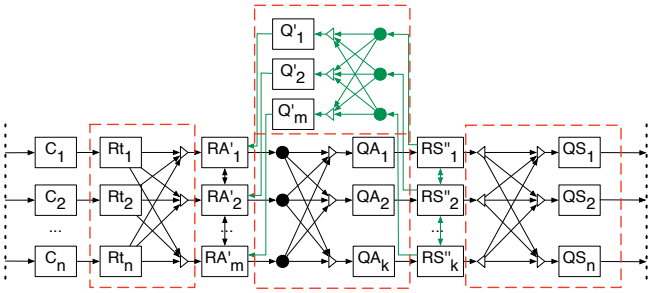

Figure 15. A Recoverable CFT Client-Server Architecture

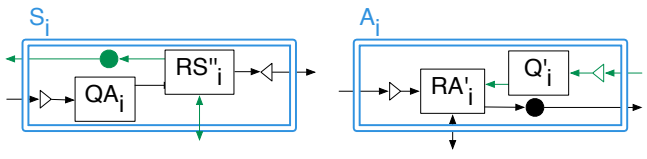

Figure 16. Machine Boundaries of RA' and RS'' Replicas

In addition to the $CFT_{2a} \ldots CFT_{2c}$ exchanges, we apply one more. Recall Figure 14, where there is a sequence of boxes $((\triangleright, Q', \bullet))$ that connect server replicas to agreement replicas. These three boxes are each SPoFs.

[4] As part of the transformation, the agreement protocol must now not only agree upon the next client request to transmit, but also on the checkpoint to save to stable storage.

Transformation RC_3 is a pair of exchanges that modify the order $(\triangleright, Q', \bullet)$ to $(\triangleright, \bullet, Q')$ and then to $(\bullet, \triangleright, Q')$. RC_3 removes $\bullet$, Q', and $\triangleright$ as SPoFs, and preserves the property that all replica RA' boxes receive quorum-decided messages from server replicas (or "Help!" messages from recovering servers). Figure 15 shows an RCFT server architecture.

Figure 16 shows the machine boundaries for RA' and RS'' replicas. Each machine must execute all of the boxes associated with its respective machine type.

3.4 Perspective

Implementation Experience. We built our RCFT design in the Lagniappe Programming Environment [26]. Lagniappe focuses on easing the development of streaming architectures for multi-$\star$ platforms[5] and is a component-based programming environment where programmers model their application dataflow while at the same time provide component implementations in an imperative language (e.g., C++). Lagniappe continues ideas proposed by others [18, 30], but instead explicitly focuses on applications with persistent state and with the challenges of running on multi-$\star$ systems.

Lagniappe is not a pure model-driven programming environment. The transformations it performs on application and platform models are hard-wired and stem from domain knowledge in properly extracting parallelism from streaming applications to achieve higher throughput and lower latency (i.e., increased performance). However, there are other reasons to transform an application to a parallel execution, specifically to increase availability by adding fault-tolerance. We intend to reimplement Lagniappe in a true MDE fashion to allow not only applications and platforms to be modeled, but also explicit transformations to be added as well. Figure 15 is the RCFT architecture that we implemented in Lagniappe.

Testing. With the expert implementation came a large and unstructured suite of tests for RCFT. Incremental development allowed us to reorganize these tests in a more meaningful manner. For example, we identified unit (module) tests for different components, such as agreement, checkpointing, recover, etc., which we then used as integration tests when these boxes were refined. This reorganization convinced us that we would have created a comparable set of tests using our process of incremental development.

For example, Figure 17a shows a basic test, Test1, that is a unit test of S, and determines if the response of S matches the expected output from C_1 sending a request. Test1 is a unit test of the server and is dependent on the actual server application. For example, a distributed file system would send back data at a location requested by the client. Test1 (and all similar tests of the server functionality) become integration tests as we refine the server abstraction to include agreement. We show this in Figures 17b and 17c.

[5] Multi-$\star$ platforms refer to any type of parallelism available in today's hardware: multi-threaded, multi-core, multi-processor, or even multi-machine.

Test2 is a unit test of $A^{\perp}$, shown in Figure 17b. Test2 tests that all agreement replicas agree upon the same next request. When transformation CFT_{1a} replicates and refines $A^{\perp}$ to $A_1 \ldots A_m$, Test2 moves from a unit test of $A^{\perp}$ to an integration test of this refinement of the agreement abstraction.

An important class of tests in any fault-tolerance setting is failure injection. Figure 17d shows a simple example. If we kill the machine on which server replica S_1 executes, Test1 must still execute correctly. This test setup—failure injection with the input stream from Test1—shows that the overall system can handle a failure, and thus the availability of the system is higher than in the system in Figure 17a. We use our system test Test1 to not only validate the correctness of the abstraction throughout the multiple refinements, but also to test *availability despite failures*. We also do more complex failure injection: once we transform the system to include recovery, we can rotate failures through the nodes to test both the overall availability and the recovery functionality.

4. Hash Joins in Database Machines

Gamma was (and perhaps still is) the most sophisticated relational database machine built in academics [11]. Most of the work on Gamma was in the late 1980s and early 1990s. We focus on Gamma's join parallelization, which is typical of modern relational database machine architectures. What is new in this section is our presentation of Gamma. Published descriptions are informal (cf. [11]); our presentation is a derivation from first principles. Readers should note that exactly the same principles used in developing RCFT architectures are used in developing parallel join architectures.

A *hash join* is an implementation of a relational equi-join; it takes two streams (A, B) of tuples as input and produces their equi-join $(A \bowtie B)$ as output. The basic hash join algorithm is simple: read all tuples of stream A into a main-memory hash table, where the join key of A tuples are hashed. Then read stream B, one tuple at a time. By hashing a B tuple's join key, one can quickly identify all A tuples that join with the B tuple. This algorithm has linear complexity in that each A and B tuple is read only once. Figure 18a shows the executable $HJOIN$ architecture that we start with.

4.1 Bloom Filtering Refinement

Joins are among the most expensive database operations. Gamma makes an ingenious use of Bloom filters [5] to reduce the number of tuples to join. It requires the use of two boxes: $BLOOM$ (to create a filter) and $BFILTER$ (to apply the filter). This refinement of $HJOIN$ is shown in Figure 18b.

The $BLOOM$ box takes a stream of tuples A as input and outputs exactly the same stream A along with a bitmap M. The algorithm first clears M. Each tuple of A is read, its join key is hashed, the corresponding bit (indicated by the hash) is set in M, and the A tuple is output. After all A tuples are read, M is output. M is the *Bloom filter*.

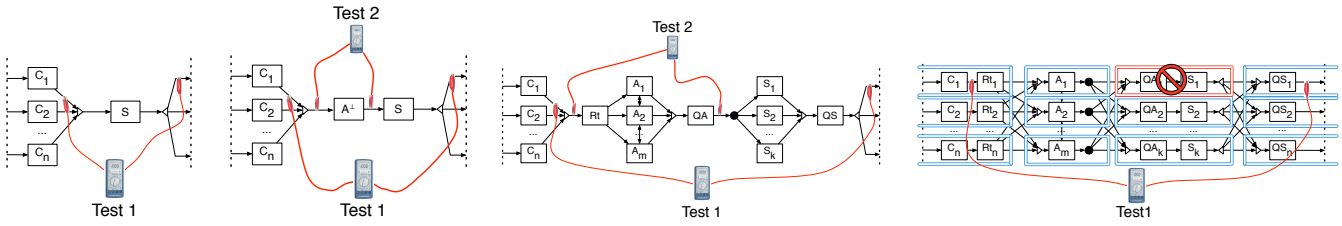

Figure 17. Testing the CFT system.

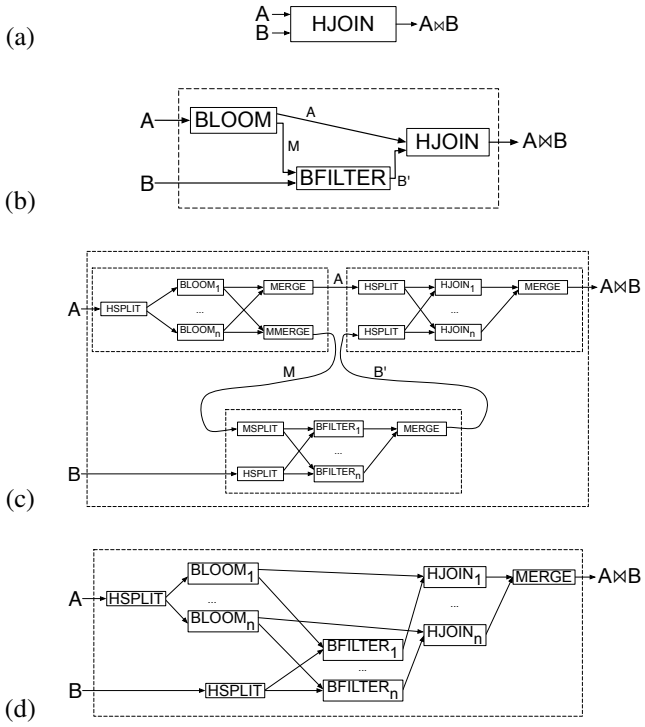

Figure 18. Hash Join Architecture

The *BFILTER* box takes a bitmap M and a stream of tuples B as input, and eliminates B tuples that cannot join with A tuples. The algorithm begins by reading M. B is read one tuple at a time; the B tuple's join key is hashed, and the corresponding bit in M is checked. If the bit is unset, the B tuple is discarded as it can not possibly join with any A tuple. Otherwise the B tuple is output. Stream B' is the result. Given the behaviors of the *BLOOM*, *BFILTER*, and *HJOIN* boxes, it is easy to prove the correctness of the Figure 18a to Figure 18b refinement.

4.2 Parallelizing Refinements

Next, the *BLOOM*, *BFILTER*, and *HJOIN* boxes are hierarchically refined by replacing each with their parallel versions (Figure 18c). A *BLOOM* box is parallelized by hash-splitting its input stream A into substreams $A_1 \ldots A_n$, creating a *BLOOM* filter $M_1 \ldots M_n$ for each substream, coalescing $A_1 \ldots A_n$ back into A, and merging bit maps $M_1 \ldots M_n$ into a single map M.

A *BFILTER* box is parallelized by hash-splitting its input stream B into substreams $B_1 \ldots B_n$, where the same hash function that splits stream A is used to split stream B. Map M is decomposed into submaps $M_1 \ldots M_n$ and substream B_i is filtered by M_i. The reduced substreams $B'_1 \ldots B'_n$ are coalesced into stream B'.

The parallelization of the *HJOIN* box is standard [1]: both input streams A, B are hash-split on their join keys using the same hashing function as before. Each stream A_i is joined with stream B_j $(i, j \in 1 \ldots n)$, yielding n^2 *HJOIN* boxes. Since an equi-join is computed, we know $A_i \bowtie B_j = \emptyset$ for all $i \neq j$ (as equal keys must hash to the same value). Thus, a single abstract *HJOIN* box is replaced by n *HJOIN* boxes instead of n^2 boxes (as all the other *HJOIN* boxes have provably null outputs). By merging the joins of $A_i \bowtie B_i$ $(i \in 1 \ldots n)$, $A \bowtie B$ is produced as output.

Figure 18c shows the result of applying all three parallelization refinements to Figure 18b. Each of the *BLOOM*, *BFILTER*, and *HJOIN* parallelization rewrites have simple proofs of correctness.

4.3 Optimizations

A primary goal of Gamma was to determine performance increases that could be gained by parallelization. The architecture of Figure 18c has three *serialization bottlenecks* which degrade performance. Consider the *MERGE* of substreams $A_1 \ldots A_n$ into A, followed by a *HSPLIT* to reconstruct $A_1 \ldots A_n$. *There is no need to materialize* A: the $MERGE - HSPLIT$ pair is the identity map: $A_i \rightarrow A_i$ $(i \in 1 \ldots n)$. The same applies for the $MERGE - HSPLIT$ pair for collapsing and reconstructing substreams $B'_1 \ldots B'_n$. The removal of $MERGE - HSPLIT$ pairs eliminates two serialization bottlenecks.

The third bottleneck combines maps $M_1 \ldots M_n$ into M and then decomposes M back into $M_1 \ldots M_n$. The $MMERGE - MSPLIT$ pair is the identity map: $M_i \rightarrow M_i$ $(i \in 1 \ldots n)$. This optimization removes the $MMERGE - MSPLIT$ boxes and reroutes the streams appropriately.[6]

[6] There are many ways in which *MMERGE* and *MSPLIT* can be realized. The simplest is this: M is a $n \times k$ bitmap. The join key of an A tuple is hashed twice: once to determine the row of M, the second to determine the column

Figure 18d shows the result of all three optimizations.

4.4 Perspective

Experience. Gamma has an elegant architecture. In presenting this material to graduate database students, we observed that it is easier to remember the *derivation* of Gamma's hash join architecture rather than to remember the final design of Figure 18d. Further, implementing and testing Gamma hash joins at each level of abstraction (as in Figure 18) is an interesting exercise; see [17] for more details. Unit tests are defined for each primitive box. When boxes are refined, unit tests become integration tests. Interesting cases occur when optimizations eliminate boxes and connectors. The harness in Figure 19 replays the unit test of a *BLOOM* box on the final hash join architecture (Figure 18d). *MMERGE* and *MERGE* boxes are reintroduced with appropriate connectors to reconstitute the output A stream and M map.

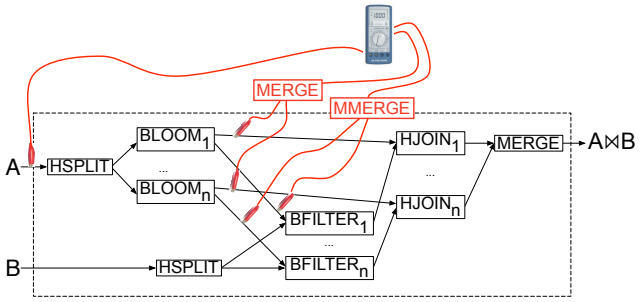

Figure 19. Unit Test of *BLOOM* after Optimization

Exchanges. Figure 18d is not the last word on Gamma's architecture. Exchanges are used to optimize the processing of cascading joins. Figure 20a shows an incomplete architecture where the output of one join cascades to the input to another. Figure 20b reveals partial internals of these *HJOIN* boxes, where the output of the first join is formed by merging substreams $C_1 \ldots C_n$ into stream C and then C is immediately hash-split into substreams $D_1 \ldots D_k$. This is yet another serialization bottleneck. Unlike the bottlenecks in Section 4.3, cascading joins use different join keys, so that the partitioning of C *before* its merge is different than the partitioning of C *after* the hash-split ($n \neq k$).

Exchanges are used to remove these serialization bottlenecks, where (*MERGE*, *HSPLIT*) pairs are swapped with (*HSPLIT*, *MERGE*) pairs (Figure 20c). That is, each C_i is hash-split into k substreams ($C_{i1} \ldots C_{ik}$) and sets of n substreams ($C_{1j} \ldots C_{nj}$) are merged into stream D_j ($j \in 1 \ldots k$). This exchange preserves the property that tuples of C whose hash-value is j are assigned to stream D_j, while eliminating a serialization bottleneck. A drawback of this exchange is bookkeeping: between the *HSPLIT* and *MERGE* boxes is an $n \times k$ matrix of substreams, which we denote by $[C]_{nk}$.

within the selected row. Thus, all tuples of substream A_i hash to row i of M. *MMERGE* combines $M_1 \ldots M_n$ into M by boolean disjunction. For each i, *MSPLIT* extracts row i from M and zeros out the rest of M_i.

Figure 20c exchanges are performed on *HJOIN* input and output streams. This generalizes Gamma's *HJOIN* architecture to Figure 20d: a single *HJOIN* box takes matrices $[A]_{ij}$ and $[B]_{kj}$ of substreams as input (stream A is hash-partitioned into $i \times j$ disjoint substreams and B into $k \times j$ substreams) and produces a matrix $[A \bowtie B]_{jn}$ of substreams as output ($A \bowtie B$ is hash-partitioned into $j \times n$ disjoint substreams). Exchanges arise in parallel database architectures generally, and Gamma's architecture in particular.

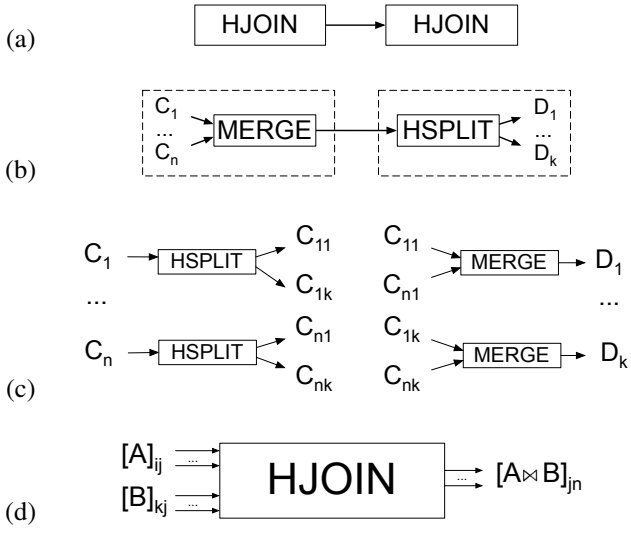

Figure 20. Exchange of *MERGE* and *HSPLIT*

5. Related Work

There is a rich collection of papers on software architecture refinement [4, 7, 8, 10, 12, 19, 13, 23, 24, 25, 31]. Although we build upon these pioneering works, the bridge is not straightforward. Here is our integrative perspective.

If programs R and P can be defined as predicates then refinement is implication [15, 16]. Program R is said to refine program P if $R \Rightarrow P$. A special case of refinement is equivalence $R \Leftrightarrow P$. Our exchange rewrites and most of the transformations used in our case studies are algebraic identities (equivalences) [23, 24].

More generally, however, programs are behaviors where actions, updates, and state changes are performed [6]; refinement is then called *behavioral substitutability*. Liskov's Substitution Principle states "Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T" [22]. Subtyping is a refinement relationship that scales to architectures [12]. *The tests that we define on an abstraction are intended to verify the key properties that we want subsequent refinements and optimizations of this abstraction to preserve.*

When transformations are not equivalences, new properties and relationships may appear among components that did not exist previously. For example, our first recovery transformation RC_1 (Section 3.3) adds a relationship from

component S to A that was not originally present (Figure 13). The addition of new relationships has an interesting consequence: an architecture's specification may change.

In the algebraic specification community, the terms *refinement* and *extension* have different meanings. Consider a 2-space, where points along the X-axis are specifications, and points along the Y-axis are implementations. A classical paradigm (e.g., Z [28]) elaborates a specification incrementally by extensions. Once completed, the specification is refined progressively into an implementation without spec alteration. Horizontal arrows in Figure 21a denote extensions; vertical arrows are refinements. Program P has specification S and implementation I in Figure 21a.

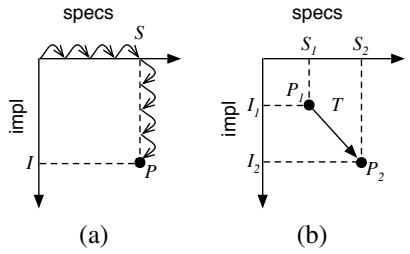

Figure 21. Extension and Refinement

Our approach is different. Transformations can simultaneously extend a specification *and* modify an implementation, a diagonal move through this 2-space. This follows modern practice: we start with an existing program (spec, implementation) and transform it to an improved (spec, implementation) – we do not start from scratch. In Figure 21b, program $P_1 = (S_1, I_1)$ is mapped to program $P_2 = (S_2, I_2)$ by transformation T. In effect, we begin with an *underspecified* design and progressively elaborate it [4, 7].

This observation has two important consequences, both of which usually distinguish our work from others. First, existing components can be "extended" to expose new ports or to have new functionality (e.g., recovery). As mentioned earlier, a primitive implementation technique to achieve this is to use preprocessors to include or exclude ports in builds; more advanced techniques to do this are known [2].

Second and more important is that a specification can have both functional and non-functional requirements [3]. *Functional requirements* express logical or behavioral relationships of components in a system. *Non-functional requirements* are properties—such as performance or availability—that are expected from an architecture. Recall our RCFT transformations. We start with a specification *CS* of a client-server relationship (the exact details of which are not important). When we apply the sequence of transformations to map *CS* to a CFT architecture with no single point of failure, we alter our the spec to $CS \wedge \neg SPoF$. When we apply the next set of transformations to add recovery, we alter our system's spec to $CS \wedge \neg SPoF \wedge Recovery$. *At no time do we alter the functional requirements of our system; our transformations progressively add non-functional requirements.*

The ability to alter specifications and implementations is essential: it allows *both* to be developed incrementally. As our development of RCFT servers shows, we focus on one concern (non-functional requirement) at a time and explain in simple steps how this concern is realized. It would be *much* more complicated to start with a complete specification and show how it is refined to an implementation. For example, compose our SPoF and Recovery transformations together, so that there are now three or four big steps/transformations to map *CS* to $CS \wedge \neg SPoF \wedge Recovery$. In all honestly, we could not have conceived such transformations: they would be too complicated to understand let alone explain. Decomposing a design (spec *and* implementation) into understandable steps is the essence of our work and of scalable stepwise development.

6. Conclusions

Experts intuitively construct non-obvious architectures that take time to understand. The details and inner workings are appreciated only with a considerable effort by non-experts. Our experience re-affirms this, but the contribution of this paper offers hope to others. By revealing details of architectural design incrementally, non-experts can document, appreciate, and contribute to domain-specific architecture development. This affords new opportunities to improve the *trustworthiness* of software. We can now verify architectures whose proofs of correctness we did not have or could not provide before. Further, we can show how architecture implementation and testing can be more systematic. And most important, we expose domain-independent (mathematical) principles that underly software design—principles that are simple enough that all developers and students can learn.

In this paper, we presented two non-trivial case studies to demonstrate our points: recoverable crash fault tolerant server architectures and architectures for parallelizing relational database hash joins. Although these domains are quite different, their architectures have straightforward and principled explanations. We implemented these architectures based on our models, whose details are presented elsewhere [9, 17, 26].

From another perspective, we note that complex designs are never created instantaneously; they come from simpler designs, and these designs from even simpler designs, recursively. By exposing the relationship between progressively more sophisticated designs enables us to understand and verify these designs, and build and test them better. Incremental development is not just "cute" (e.g., nice to have), but rather is essential to achieving these goals.

A next step in research is to develop MDE-based tools that allow designers to explore a design space interactively by defining and applying transformations. Produced designs would rely on standard MDE technology to synthesize implementations. In effect, architectural refinement provides a

another dimension of activities in MDE, allowing engineers to alter designs in a verifiable way.

Acknowledgments. We gratefully acknowledge helpful feedback from G. Karsai (Vanderbilt), E. Hehner (Toronto), H. Vin (Tata Consulting), C. Lengauer (Passau) on early drafts. Batory and Riché are supported by the NSF's Science of Design Project #CCF-0724979. Riché was additionally supported by NSF's Computer Systems Research Grant CNS-0509338.

References

[1] F. Baru. DB2 Parallel Edition. *IBM Systems Journal*, 34(2), 1995.

[2] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Stepwise Refinement. *IEEE TSE*, (30), June 2004.

[3] I. Baxter. Design Maintenance Systems. *CACM*, 35(4):73–89, 1992.

[4] J. P. Bernhard and B. Rumpe. Stepwise refinement of data flow architectures. Technical Report TUM-19746, Technische Universität München, 1997.

[5] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[7] M. Broy. Compositional refinement of interactive systems. *JACM*, (44), 1992.

[8] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal Component Model. Technical report, Object Web Consortium, http://fractal.objectweb.org/specification/index.html, 2004.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. L. Riché. UpRight Cluster Services. In *Proc. of 22nd SOSP*, Big Sky, MT, Oct. 2009.

[10] J. Cobleigh, L. Osterweil, A. Wise, and B. Lerner. Containment units: A hierarchically composable architecture for adaptive systems. In *Proc. of FSE*, pages 159–165, 2002.

[11] D. J. Dewitt, S. Ghandeharizadeh, D. Schneider, A. B. H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[12] D. Garlan. Style-Based Refinement for Software Architecture. In *Proc. of ISAW*, pages 72–75, 1996.

[13] M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *Proc. of ICSE*, pages 23–34, 1991.

[14] Hadoop. http://hadoop.apache.org/core/.

[15] E. Hehner. Predicative programming part 1. *CACM*, 1984.

[16] C. Hoare. Programs are Predicates. In *Royal Society of London on Mathematical Logic and Programming Languages*, pages 141–155. Prentice Hall, 1985.

[17] E. Kang, T. L. Riché, and D. Batory. Incremental Development of Streaming Architectures: A Case Study. In preparation, 2009.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[19] J. Kong, K. Zhang, J. Dong, and G. Song. A Graph Grammar Approach to Software Architecture Verification and Transformation. In *Proc. of ICSAC*, pages 492–497, 2003.

[20] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[21] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos Register. In *Proc. of IEEE SRDS*, pages 114–126, Washington, DC, USA, 2007. IEEE Computer Society.

[22] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM ToPLaS*, pages 16–6, 1994.

[23] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *ACM SIGSOFT*, pages 164–174, 1994.

[24] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architectural refinement. *IEEE TSE*, (21):356–372, 1995.

[25] National Instruments LabView 8. http://www.ni.com/labview/.

[26] T. L. Riché. *The Lagniappe Programming Environment*. PhD thesis, The University of Texas at Austin, Aug. 2008.

[27] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.

[28] J. M. Spivey. The Z Notation: A Reference Manual, 1998.

[29] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[30] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of Inter. Conf. on Compiler Construction*, Grenoble, France, Apr. 2002.

[31] S. Wang, G. S. Avrunin, and L. A. Clarke. Architectural Building Blocks for Plug-and-Play System Design. In *Proc. of SCBSE*, 2006.

[32] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, Oct. 2003.

[33] Zookeeper. http://hadoop.apache.org/zookeeper.