Pushouts in Software Architecture Design

T. L. Riché
National Instruments
Austin, TX, USA
taylor.riche@ni.com

R. Gonçalves
Departamento de Informática
Universidade do Minho
Braga, Portugal
rgoncalves@di.uminho.pt

B. Marker and D. Batory
Dept. of Computer Science
University of Texas at Austin
Austin, TX, USA
{bamarker,batory}@cs.utexas.edu

ABSTRACT

A classical approach to program derivation is to progressively extend a simple specification and then incrementally refine it to an implementation. We claim this approach is hard or impractical when reverse engineering legacy software architectures. We present a case study that shows *optimizations* and *pushouts*—in addition to refinements and extensions—are essential for practical stepwise development of complex software architectures.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms

Design, Theory

Keywords

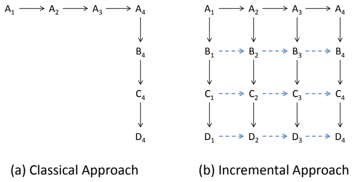
MDE, Software Architectures, Program Derivation, Pushouts

1. INTRODUCTION

At a recent NSF workshop on the Future of Software Engineering [45], there was consensus on the need for automated support for program evolution—for the obvious reason that programs can be very complicated structures whose safe and correct manipulation should yield a major improvement in the software lifecycle [24].

Workshop discussions revealed an interesting open problem. Classical approaches to program development define the spec of a simple program $\mathtt{A}_1$ and progressively extend it with additional functionality until the desired, fully-formed spec $\mathtt{A}_4$ is completed. Then $\mathtt{A}_4$ is incrementally refined to an implementation, arriving at program $\mathtt{D}_4$ (Figure 1a) [53]. A variation is after each extension, an implementation is produced by "replaying derivations" [6, 24, 38]. *After the workshop it was obvious (to us) that how one accomplishes derivation replay in the presence of extensions is not well understood.*

GPCE'12, September 26–27, 2012, Dresden, Germany.



(a) Classical Approach (b) Incremental Approach

Figure 1: Paths

We are exploring transformation-based derivations of pipe-and-filter architectures to reverse engineer designs of legacy architectures. Our goals are to reveal a process to derive architectural designs and to reconstruct these applications directly from our derivations.

We discovered that prior work on refinement under-appreciates two key ideas. First, *optimizations* are essential—one must erase modular boundaries in order to achieve designs that have non-functional properties (e.g. efficiency or fault-tolerance). Optimizations are not always part of refinement methodologies [27, 30, 38, 59].

Second, architectural designs can be very complicated. We found it impractical to refine fully-elaborated specifications—the refinements were too complex to understand and explain. Instead, we followed an incremental path (Figure 1b) that (a) refined a simple spec to its implementation at different levels of abstraction (designs $\mathtt{B}_1$, $\mathtt{C}_1$, $\mathtt{D}_1$), and then (b) incrementally extended these implementations using *pushouts* (completing a directed square □ from an incomplete square ⌜) to complete the commuting diagram of Figure 1b [47]. Each step (refinement, optimization, extension) was relatively easy to understand, allowing us to comprehend how a legacy application worked despite its complexity. Pushouts were the key for us to understand how to "replay derivations."

We present a case study of how we reverse-engineered the architecture of a complex, state-of-the-art asynchronous crash-fault tolerant server. We demonstrate the essential role of optimizations and pushouts—in addition to refinement and extension—by showing how we recover architectures of two versions of the system (both of which had no formal architectural description prior to our work): first a synchronous server and then an extension to an asynchronous server.

Stated differently, without optimizations and pushouts, we could not have reengineered legacy architectures nor could we have forward-engineered (reconstructed) them. Our conclusion is that optimizations and pushouts are essential for practical stepwise development of complex software architectures.

2. REFINEMENTS AND OPTIMIZATIONS

A *pipe-and-filter (PnF) architecture* is a directed multigraph of boxes and connectors that implements a system. A *box* is a component with input and output ports. A *connector* is a communication path for messages drawn in the direction of dataflow from an output port to one or more input ports [50].

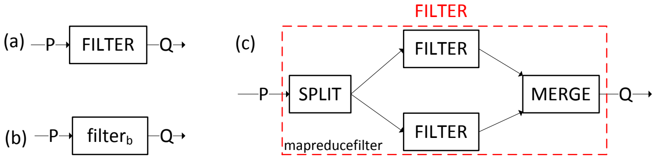


Figure 2: Pipe and Filter Architectures

A filter architecture is an example (Figure 2a). It consists of a single box FILTER that takes a stream of photographs P as input, examines each photograph p in P, and outputs p only if some criteria is satisfied. A shorter stream Q is produced. FILTER may have other parameters, such as a photograph type and filtering criteria. We elide these details.

An architectural *transformation* is a mapping (a multigraph rewrite) of an input architecture to an output architecture. We use three types of transformations: refinement, optimization, and extension. We discuss the first two now.

2.1 Refinement

Suppose there are multiple (but semantically equivalent) filtering boxes, $\mathtt{filter_a}$ and $\mathtt{filter_b}$, each with its own distinct performance characteristics. A transformation could replace the FILTER box of Figure 2a with the $\mathtt{filter_b}$ box resulting in Figure 2b. Another transformation replaces the FILTER box of Figure 2a with its map-reduce counterpart in Figure 2c, showing that an input stream can be split into substreams, each substream is filtered in parallel, and the output substreams are merged [16].

The collection of such transformations used in a domain forms a graph grammar: let A be an interface (a box that defines only the input/output ports and—at least informally—box semantics), and let $\mathtt{g_1}$, $\mathtt{g_2}$,... denote architectures (multigraphs) and/or primitive boxes that implement A:

$$\mathtt{A} \;:\; \mathtt{g_1} \mid \mathtt{g_2} \mid \ldots \;;$$

As an example, if FILTER of Figure 2a is a filter interface and everything else is an implementation, we have:

$$\begin{aligned}\mathtt{FILTER} \;:\; & \mathtt{filter_a} \mid \mathtt{filter_b} \\ & \mid \mathtt{mapreducefilter(\ SPLIT,\ FILTER,\ MERGE\ )} \;;\end{aligned}$$

where mapreducefilter(SPLIT, FILTER, MERGE) is the architecture of Figure 2c that is parameterized by implementations of the SPLIT, FILTER, and MERGE interfaces. Replacing an interface with an implementation is *refinement* [59].

Of course, the $\mathtt{g_i}$ multigraphs can reference other interfaces which have their own productions (implementations). The set of multigraphs that can be constructed by substituting implementations for interfaces defines the *language* or *domain* of systems that can be synthesized by refinement.

In general, a refinement has preconditions to satisfy before it can be applied. Consequently our grammars are not context-free. Recording such preconditions is indeed part of our work; we elide these details.

2.2 Optimization

Refinements alone are insufficient to build efficient systems. Consider the following grammar, where uppercase names are interfaces and lowercase names are components:

$$\begin{aligned}\mathtt{A} &: \mathtt{a\,B\,c} \mid \ldots \;; \\ \mathtt{B} &: \mathtt{b} \mid \ldots \;;\end{aligned}$$

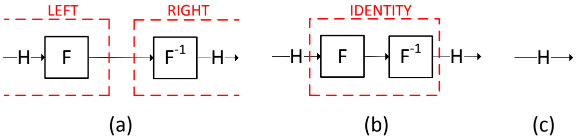


Figure 3: Architecture Optimizations

A sentence of this grammar is abc. Suppose composition bc implements interface Z:

$$\mathtt{Z} \;:\; \mathtt{bc} \mid \mathtt{q} \;;$$

Further, domain experts know that composition bc is inefficient and can be replaced by box q, which is faster. This is accomplished by *abstracting* sentence abc to aZ and then refining to a faster program by replacing Z with q to yield aq. Abstraction followed by refinement is the essence of architectural *optimization*. The pairing of bc arises because c comes from a refinement of A and b comes from a refinement of B. Dissolving modular boundaries exposes inefficiencies which can be removed by optimizations.

Figure 3a is an example. Modular boundaries are indicated by dashed lines. The LEFT box processes stream H by box F. The RIGHT box immediately processes its input by box $\mathtt{F^{-1}}$, the inverse of F. Figure 3b dissolves these boundaries, exposes the inefficiency, and reveals the IDENTITY abstraction of which (F followed by $\mathtt{F^{-1}}$) is an implementation. Figure 3c is the optimized architecture.

With optimizations, our grammars become a set of bi-directional rewrite rules (interface ⇔ implementation pairs) called a *Thue System* [8].

2.3 Application to Case Studies

The PnF architecture of our case study was *never* conceived in terms of transformations and was *not* built with the aid of software architectural models. However, the novelty, indeed genius, of its design can be expressed in terms of a sequence of transformations that were *implicitly used by their authors*. It was built over a five month period during which its authors used less rigid engineering approaches. The authors did what felt natural, but effectively used transformations. (Exposing these transformations and making them explicit is one of our contributions). There may be no a priori reason or justification for why the authors chose particular transformations, other than they were necessary for that system or that they introduced a novel algorithm or protocol.

Further, our explanations are no substitute for domain expertise; they are intended to complement, encode, and structure domain knowledge for others to follow. Expressing designs by transformations may be novel to domain experts, but the end result is rarely surprising to them. Further, the assumptions we use are standard fare for the domain. On the other hand, non-experts may find the assumptions or the choice of transformations unintuitive. In any case, our approach offers a simple and effective way to reveal architectural details of PnF applications incrementally and *be able to recreate these applications from these descriptions*.

3. THE SCFT ARCHITECTURE OF UPRIGHT

Our interest in PnF architectures originally stemmed from a reverse-engineering effort to understand Figure 4, which is a portion of the PnF architecture of *Upright*, a state-of-the-art *Asynchronous Crash-Fault Tolerant (ACFT)* server [14]. We could not explain or derive this architecture directly. Instead, we first derived a simpler architecture for a *Synchronous Crash-Fault Tolerant (SCFT)* server. In later sections, we extend our SCFT design to an ACFT design.

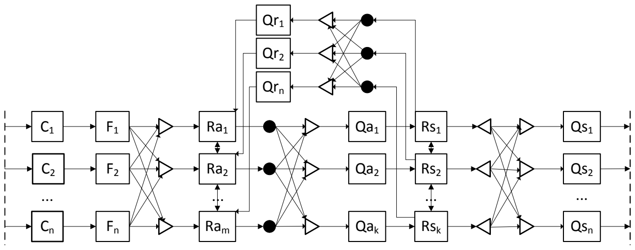


Figure 4: Asynchronous Crash Fault Tolerant Architecture

3.1 Basic Request Processing

Request-processing applications (RPAs) have multiple clients sending requests (a.k.a. *messages*) to a server. Client requests can read or write the server's internal state, which persists across requests. That servers have state is important: *crash-fault tolerance (CFT)* of non-stateful servers is trivial.[1]

RPAs have a cylinder topology representing the cyclic flow of request-response. We unroll the cylinder (Figure 5a) by breaking the seam along dotted lines. Figure 5b shows a typical architecture with clients $\mathtt{C}_1 \ldots \mathtt{C}_n$ and server `S`. Each client sends messages to the server. Messages from different clients are serialized into a single stream by ▷. The server receives each message, updates its state, and then sends its response. Responses are demultiplexed into multiple output streams, one per client, by ◁. Response messages wrap around the dotted lines. [2]

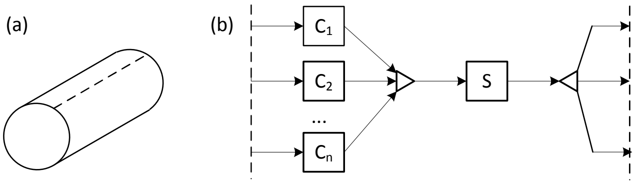


Figure 5: Unrolled Cylinder Topology

3.2 Synchronous CFT

Crash fault tolerance is the ability of a service to survive a number of failures. A *crash failure* occurs when a box stops processing messages—no messages pass through a failed box and a failed box cannot create new messages. The failure of network boxes—▷, ◁, ⊙ (reliable broadcast [43]), and • (broadcast)—are treated identically to software boxes. Failure is self-contained, meaning that failures do not propagate across box boundaries. [3, 4]

The technical objective of SCFT is to eliminate *Single Points of Failure (SPoF)* by replicating functionality [49]. An SPoF is the failure of a single box that causes the entire server abstraction to fail. Our initial design (Figure 5b) has three SPoFs: the serializer ▷, the server `S`, and the demultiplexer ◁. Client failures never cause a server abstraction to fail as SCFT servers should expect and gracefully handle non-responsive clients. We show in the following sections how Figure 4 is an extended, refined, and optimized implementation of Figure 5b that has no SPoFs and is recoverable.

3.2.1 List Refinement

The `List` transformation maps box `S` to an architecture where there is a single box `L` between the clients and server. `L` implements an ordered list of messages, collecting messages from clients and passing messages one at a time to the server. In effect `L` makes the network queue explicit, materializing a placeholder for subsequent refinements. Figure 6 shows the architecture after applying `List`.

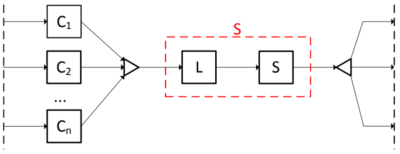


Figure 6: After the `List` Refinement

3.2.2 Replication Refinements

The next transformations, `Paxos` and `RepS`, replicate boxes to improve system availability, i.e. to make the server abstraction more resilient to crashes. See Figure 7.

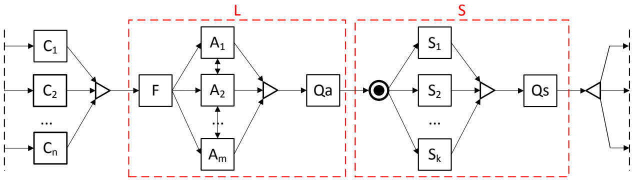


Figure 7: Replicated Agreement and Server Boxes

First consider the replicating-server refinement `RepS`. It replicates the server `S` `k`-times, indicated by boxes $\mathtt{S}_1 \ldots \mathtt{S}_\mathtt{k}$, and adds three new boxes: ⊙, ▷, and `Qs`. Box ⊙ reliably broadcasts an incoming message to each server replica: if one server replica receives a message, then all replicas receive the message—such a strong guarantee is necessary for correctness. Replicas receive the message, update their state, and send responses. ▷ 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 a single response to the client, *thus maintaining the abstraction of a single server* `S`.

Now consider the `Paxos` refinement of box `L` [36, 37]. When a client submits a message, a *forwarding box (F)* receives it and delivers the message to any or all of the `m` identical agreement boxes $\mathtt{A}_1 \ldots \mathtt{A}_\mathtt{m}$. Each `A` box implements an agreement protocol to guarantee a quorum-decided linear order in which client messages should be processed. `A` replicas communicate with each other to determine the next client message to process. Each `A` replica votes, sending

[1] Replicating a stateless application is easy: consistency across replicas is guaranteed by the fact that the replicas never change.

[2] CFT servers must uphold two properties: safety and liveness. Loosely, *safety* means the server is always in a "good" state, for varying definitions of "good", and *liveness* means that the server eventually makes progress [43]. Upright provides a serializable view of RPAs [46]; serialization proofs of Upright's protocols are given in [34]. Further, Upright's protocols uphold a liveness property called "eventual liveness" [14]. As consensus in an asynchronous system is impossible [20], the existence of occasional periods of a well-behaved network is assumed, allowing both SCFT and ACFT servers to make progress.

[3] We assume that each box executes on its own machine. Normally, multiple boxes are mapped to a single machine. The rule for machine failure is simple: if a machine fails, all boxes on that machine fail.

[4] All requests and boxes are assumed benign. Malicious behavior (or even benign behavior outside the accepted application protocol) is assumed to not occur. Work on *Byzantine Fault Tolerance (BFT)* relaxes these assumptions [13, 14, 35]. *In any case, the mapping of a vanilla client-server design to a SCFT design is unaffected: BFT designs extend SCFT designs.*

its “next” message to the quorum box Qa. Qa forwards a single message to the server once it receives identical messages from a sufficient number of replicas. In this way, Paxos *maintains the behavior of and interface to a single* L *box.*

The degree of A and S box replication—values m and k—depend on the number of faults f to tolerate and the agreement protocol. Common assignments set $\mathtt{m} = 2\mathtt{f} + 1$ and $\mathtt{k} = \mathtt{f} + 1$. [5]

The Paxos and RepS transformations commute as the order in which they are applied does not matter. However, both must be applied to guarantee that the system tolerates the failures of up to f server and f agreement boxes. Figure 7 is the result of applying Paxos and RepS to Figure 6.

3.2.3 Optimizations

Our original architecture of Figure 5b had three SPoFs; our current design has eight. (From left to right in Figure 7, they are the $\rhd$, F, $\rhd$, Qa, $\odot$, $\rhd$, Qs, and $\lhd$ boxes.) *Here is where refinement is insufficient to derive our target architecture; modular boundaries must be broken to remove SPoFs.*

Exactly as explained in Section 2.2, we dissolved boundaries in Figure 7 to produce Figure 8 and identify three new abstractions that circumscribe two or three boxes, all of which are SPoFs. For example the LEFT abstraction contains SPoF boxes $\rhd$ and F. MIDDLE contains three SPoF boxes ($\rhd$, Qa, and $\odot$), and the RIGHT abstraction contains three SPoF boxes ($\rhd$, Qs, and $\lhd$).

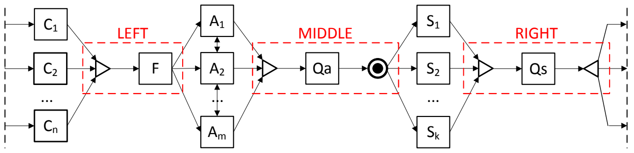


Figure 8: Abstractions with Multiple SPoFs

Here is how SPoFs were removed from Upright: Figure 9a shows that box G consumes n streams $\mathtt{I_1} \ldots \mathtt{I_n}$ to produce a single stream T which is input to box H. In turn, H outputs m streams $\mathtt{O_1} \ldots \mathtt{O_m}$. Let R be interface that G followed by H implements. A *rotation* swaps the positions of G and H (possibly replacing them with slightly different operations $\mathtt{H'}$ and $\mathtt{G'}$) and replicates them, providing yet another implementation of R.[6]

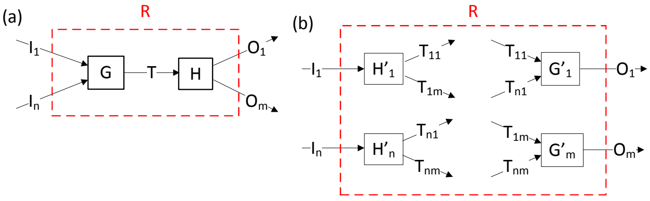


Figure 9: Rotation Optimization

To see a real example of a rotation, consider the LEFT abstraction of Figure 8, which we show again in Figure 10a comprising the sequence ($\rhd$, F). The $\mathtt{R_a}$ rotation swaps the order of these boxes (see Figure 10b). Box F is replicated once for each client and $\rhd$ is replicated once for each $\mathtt{A_i}$. That is, instead of serializing all requests and then forwarding, client requests are immediately forwarded and then serialized before each A replica. The property that each client request is sent to a subset of A replicas is preserved by $\mathtt{R_a}$. As expected, the interface of n input channels and m output channels is maintained. But now, F and $\rhd$ boxes *are no longer SPoFs* in Figure 10b.

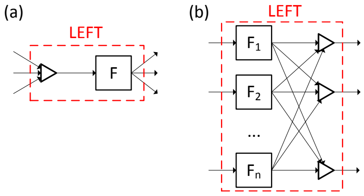


Figure 10: The $\mathtt{R_a}$ Rotation

The MIDDLE abstraction of Figure 8, replicated in Figure 11a, consists of the box sequence ($\rhd$, Qa, $\odot$). Transformation $\mathtt{R_b}$ is a pair of rotations that modify the order ($\rhd$, Qa, $\odot$) to ($\rhd$, $\odot$, Qa) (Figure 11b) and finally to ($\odot$, $\rhd$, Qa) (Figure 11c). That is, instead of taking a quorum of responses from A replicas and reliably broadcasting the result, the results of all A replicas are reliably broadcast and a quorum is taken at each server replica. The property that a quorum-decided request from replicated A boxes is delivered to all server replicas is preserved by $\mathtt{R_b}$.

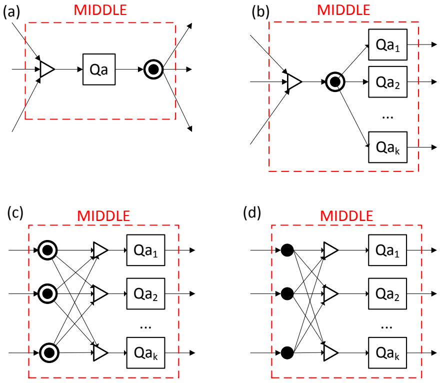


Figure 11: The $\mathtt{R_b}$ Optimization

$\mathtt{R_b}$ includes one more optimization that is well-known to domain experts. Reliable broadcast is very expensive.[7] By replicating $\odot$ at each of the $\mathtt{A_i}$ boxes, we can take advantage of the fact that quorums are taken at each server and replace the reliable broadcast box $\odot$ with the normal (unreliable) network broadcast box $\bullet$ that is simple to implement. The abstraction in Figure 11d now contains no SPoFs, and also represents a standard and efficient way to implement a *reliable crossbar* [14].

Similarly, transformation $\mathtt{R_c}$ is a pair of rotations that is applied to the RIGHT abstraction in Figure 8, which maps the box sequence ($\rhd$, Qs, $\lhd$) to ($\rhd$, $\lhd$, Qs) and then to ($\lhd$, $\rhd$, Qs), thereby eliminating all SPoFs. Figure 12 is the result of applying $\mathtt{R_a}$–$\mathtt{R_c}$ to Figure 8. It is the PnF architecture of Upright’s SCFT design.

[5] Yin et al. explain the 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) [60]. Since a typical server requires more computational resources than agreement, thus needing a more powerful and expensive machine, using fewer server replicas in an architecture is desirable.

[6] Readers will recognize this as commuting diagram $\mathtt{G} \cdot \mathtt{H} = \mathtt{H'} \cdot \mathtt{G'}$. The sequential composition of G and H is replaced with a cross-product of $\mathtt{H'}$ and $\mathtt{G'}$: the end result is two different implementations of the same abstraction.

[7] Experts often use an agreement cluster to implement reliable broadcast.

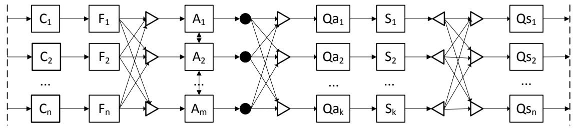


Figure 12: Synchronous Crash Fault Tolerant Architecture

3.2.4 Recap

We used the architecture of Figure 12 to reimplement Upright's SCFT design. More details about model validation are postponed until Section 5.6. Our next task is to extend the SCFT architecture to an ACFT architecture (i.e. mapping from Figure 12 to Figure 4).

4. PUSHOUTS AND EXTENSIONS

Think of a refinement as a vertical mapping from an initial architecture A to a resultant architecture B. We also need transformations that extend architectures with more functionality. Extensions are horizontal mappings, such as mapping architecture A to architecture E (Figure 13).

$$\begin{tikzcd} A \arrow[r, "\text{extension}"] \arrow[d, "\text{refinement}"'] & E \arrow[d, dashed] \\ B \arrow[r, dashed] & F \arrow[phantom, d, "\text{pushout}", near start] \\ & \end{tikzcd}$$

Figure 13: Pushout

An architectural *extension* augments an architecture by adding new functionality and ports to existing boxes, generalizing connectors, and adding new boxes and connectors. In software product lines, extensions are called *features* [29]. A common special case of extension is *box extension*, where new capabilities and ports are added to a box. 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 [3, 5, 31]. Connectors can also be extended (e.g. messages of type M are transmitted; an extension transmits a subtype T of M). We write $\mathtt{X} \rightsquigarrow \mathtt{Y}$ to mean that X is *extended to* Y.

Figure 14 illustrates architectural extension $\alpha \rightsquigarrow \beta$. Figure 14$\alpha$ shows architecture α with a Sort and WebServer box. The WebServer takes sorted tuples and creates a webpage of sorted results. Figure 14β shows an extended architecture β that permits sort keys to be altered at run time. Sort and WebServer are extended with new ports ($\mathtt{Sort} \rightsquigarrow \mathtt{ESort}$ and $\mathtt{WebServer} \rightsquigarrow \mathtt{EWebServer}$) and a feedback connector labeled newKey is added. A newKey message changes the key that ESort uses to sort incoming tuples (e.g. switching from last names to ID numbers in a patient database or artists to album titles in a music player).

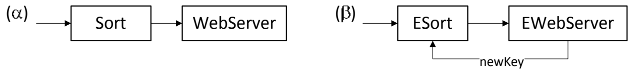


Figure 14: Architecture Extension $\alpha \rightsquigarrow \beta$

Refinement is different than extension. Refinement preserves semantics; it maps an interface to a streaming architecture that implements that interface. The interface is not altered and new connectors external to it are not added. In contrast, extension enhances the semantics of an architecture by elaborating existing boxes with new ports, new functionality, and new boxes and connectors.[8]

A *pushout* is a completion of the commuting diagram of Figure 13. Given refinement or optimization $\mathtt{A} \rightarrow \mathtt{B}$ and extension $\mathtt{A} \rightsquigarrow \mathtt{E}$, the goal is to determine architecture F so that $\mathtt{E} \rightarrow \mathtt{F}$ is a refinement or optimization and $\mathtt{B} \rightsquigarrow \mathtt{F}$ is an extension.[9]

We use pushouts to map Upright's SCFT architecture to its corresponding ACFT architecture in the next section.

5. THE ACFT ARCHITECTURE OF UPRIGHT

Failure is permanent in an SCFT architecture. If a box fails, it no longer responds and will never respond again. An SCFT architecture supports the failure of f boxes. When the limit is exceeded, the entire system is in a failed state, the one-server abstraction is violated, and safety and liveness are no longer guaranteed.

In this section, we explain how domain experts relaxed restrictions on liveness. An ACFT architecture guarantees liveness even with an occasional poorly behaved network or temporary box failure (and always guarantees safety), assuming the network eventually enters a sufficiently long well-behaved period. That is, liveness is guaranteed *even with occasional network asynchrony*.

In effect ACFT limits the situations where a client sees an unresponsive server abstraction. The mechanisms that support asynchrony also allow the architecture to mask some failures by enabling a box to be restarted and catch up. A box that fails and recovers is considered correct, albeit slow. The other boxes will continue to make progress; a recovering box may not be able to catch up immediately. If other boxes fail, the remaining servers may have to wait for the recovering servers to catch up (as required by quorum boxes), at which point the system can resume servicing client requests [14]. Note: boxes may still crash and enter a state where they will never again process requests; asynchrony support cannot mask all failures.

5.1 Road Map

Figure 15 shows a road map (commuting diagram) of where we are and where we are going. We began with the topmost left node (labeled Figure 5b) and incrementally refined and optimized downward until we reached the bottommost left node (Figure 12). In the next sections, we use extensions $\mathtt{Async}_0 \ldots \mathtt{Async}_3$ to map each of the architectures on the left to corresponding architectures on the right. The dashed vertical arrows in Figure 15 are extended refinements and optimizations ($\mathtt{List} \rightsquigarrow \mathtt{RList}$, $\mathtt{Paxos} \rightsquigarrow \mathtt{RPaxos}$, etc.) that complete the commuting diagram.

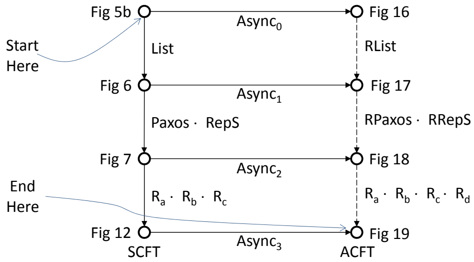


Figure 15: Roadmap in Our ACFT Designs

[8] Interestingly, well-known tools for data flow architectures, like LabVIEW [44], Weaves [23], Fractal [10], and Click [32] support refinement, but *not* extension and *not* optimization. Our use of the terms refinement and extension is consistent with that used in formal methods [52].

[9] The objects of this category are designs of a system at different levels of abstraction and with different functionality.

5.2 Asynchronous Extension $\mathtt{Async}_0$

$\mathtt{Async}_0$ extends S (the interface of a non-recoverable server) to $\mathtt{RS}^\alpha$ (the interface of a recoverable asynchronous server), $\mathtt{S} \rightsquigarrow \mathtt{RS}^\alpha$. Recovering from asynchrony includes logging, checkpointing, and the ability to load a checkpoint and replay the log upon restart. Most modern request-processing servers implement this functionality, such as Hadoop [25] and Zookeeper [61]. Figure 16 is the initial PnF architecture of an ACFT server.

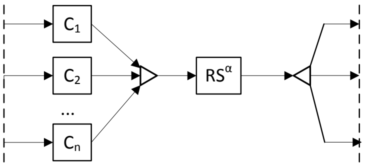


Figure 16: Initial ACFT Architecture

5.3 The $\mathtt{Async}_0$ and List Pushout

Recall that the List refinement introduces an L box in front of server S in Figure 6, where L is part of the greater abstraction of one correct server. For the entire abstraction to be recoverable, all of its stateful internal boxes must be recoverable. $\mathtt{Async}_1$ modifies the architecture of Figure 6 with two box extensions, $\mathtt{L} \rightsquigarrow \mathtt{RL}$ and $\mathtt{RS}^\alpha \rightsquigarrow \mathtt{RS}^\beta$, and adds a new connector: the recoverable list box needs information from the recoverable server to checkpoint the server's state. A "feedback" connector is used to transport this request. The result is Figure 17.

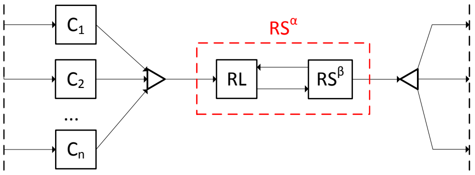


Figure 17: After Extension $\mathtt{Async}_1$ or Refinement RList

Admittedly, Figure 17 is evident only to domain experts. To non-experts, Figure 17 simply states that the recoverable list and server boxes are tightly coupled in Upright's design.

5.4 The $\mathtt{Async}_1$ and $\mathtt{Paxos}\cdot\mathtt{RepS}$ Pushout

Recall the Paxos and RepS refinements replicate agreement and server boxes. $\mathtt{Async}_2$ extends agreement and server boxes to support asynchronous recovery and adds new boxes and connectors to implement the "feedback" connector of Figure 17. The result is Figure 18.

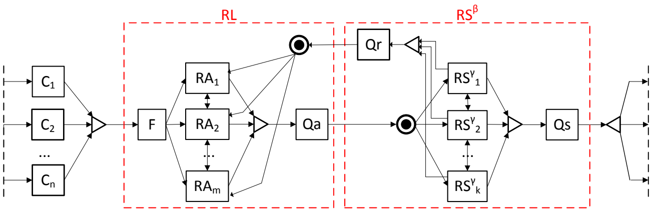


Figure 18: After Extension $\mathtt{Async}_2$ or Refinement $\mathtt{RPaxos}\cdot\mathtt{RRepS}$

Server Replication.

When the recoverable server $\mathtt{RS}^\beta$ of Figure 17 is replicated, it needs to communicate with other replicas. The reason lies in the recovery algorithm for replicated servers: when a replica crashes, other servers continue to make progress. When the failed server recovers—it may be behind—other client messages may have been processed in the interim. For the failed server to catch up, it must fetch checkpoint state from other replicas.

Extension $\mathtt{Async}_2$ maps each server box S of Figure 7 to a recoverable server box $\mathtt{RS}^\gamma$, $\mathtt{S} \rightsquigarrow \mathtt{RS}^\gamma$. The server box is extended to account for server recovery (as $\mathtt{RS}^\beta$, discussed above) and for the following situation. After processing a fixed number of client messages, the agreement box asks the server for its current checkpoint. The agreement box can only accept a checkpoint if it receives a quorum of matching checkpoints from server replicas.[10] Further, 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 Qr (a) takes quorums of checkpoint messages from servers and (b) immediately passes along recovery messages of servers that have fallen behind. In this way, *extension* $\mathtt{Async}_2$ *or refinement* RRepS *maintains the abstraction of a single recoverable server* $\mathtt{RS}^\beta$.

Agreement Replication.

Extension $\mathtt{Async}_2$ maps each agreement box A of Figure 7 to a recoverable agreement box RA, $\mathtt{A} \rightsquigarrow \mathtt{RA}$, that performs the same tasks as an A box, but in addition, responds to new messages, such as one for receiving the current checkpoint on which the RA replicas must agree.[11] Further, a reliable broadcast (⊙) is introduced that sends all incoming "Help" messages from the server abstraction to all agreement replicas $\mathtt{RA}_1 \ldots \mathtt{RA}_m$. In this way, *extension* $\mathtt{Async}_2$ *or refinement* RPaxos *maintains the behavior of and interface to a single* RL *box.*

5.5 The $\mathtt{Async}_2$ and $\mathtt{R_a}\cdot\mathtt{R_b}\cdot\mathtt{R_c}$ Pushout

Once again, we are at a point where refinement is inadequate to complete the AFCT design; optimizations are needed to complete the ACFT architecture. The transformation that maps Figure 18 to Figure 19 is simple: the same optimizations $(\mathtt{R_a}\cdot\mathtt{R_b}\cdot\mathtt{R_c})$ that were used in the SCFT architecture are replayed. Further, the additional boxes that were added in Figure 18 are all SPoFs. Transformation $\mathtt{R_d}$ is a pair of rotations that modify the order $(\triangleright, \mathtt{Qr}, \odot)$ to $(\triangleright, \odot, \mathtt{Qr})$ and then to $(\odot, \triangleright, \mathtt{Qr})$. Further, $\mathtt{R_d}$ replaces the reliable broadcast ⊙ with the normal broadcast •. Doing so removes SPoFs and preserves the property that all replica RA boxes receive quorum-decided messages from server replicas (or "Help!" messages from recovering servers). Optimization $\mathtt{R_a}\cdot\mathtt{R_b}\cdot\mathtt{R_c}\cdot\mathtt{R_d}$ of Figure 18, or equivalently extension $\mathtt{Async}_3$ of Figure 12, yields Figure 19 (Figure 4), Upright's ACFT server architecture.

5.6 Validation

We implemented both the SCFT and ACFT architectures by hand, incrementally as described above, using Python 3 atop a custom-built, light-weight streaming-application framework (code for this case study can be downloaded from [21]). Each box was a process; messages were transferred via sockets. We tested the initial archi-

[10] By receiving occasionally checkpoints from the server, the agreement nodes can implement recovery without the requirement of an infinite replay log. This optimization is obviously desirable.

[11] 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.

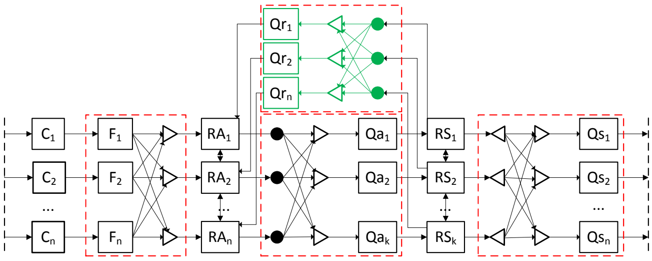


Figure 19: Asynchronous Crash Fault Tolerant Architecture

tecture and each subsequent architecture that was derived, thereby validating our transformations.

As future work we hope to collaborate with our ACFT colleagues to formally prove the correctness of each individual transformation, making this work an example of the *correct by construction* paradigm. As mentioned earlier, Clement et al. have a proof of the *composite* transformation for the ACFT architecture [14], but not proofs for each *individual* transformation. Although special cases of the ACFT proof reduce to proofs of specific compositions of transformations, it is unclear whether proofs of individual transformations are ultimately simpler than proofs of a single composite transformation.

Interestingly, rotations were unfamiliar to our ACFT colleagues. Their informal designs went directly from Figure 5b to Figure 19, avoiding our incremental construction of a reliable crossbar. This discovery suggests how our approach may explain complex and intuitive designs in a structured way.

5.7 Epilog

We continued to elaborate our ACFT design to Upright's full design for a *Byzantine Fault Tolerant (BFT)* server. This required architectural extensions and pushouts to the ACFT architectures (right-hand sides of Figure 15) to add batching, authentication, and machine boundaries.

In doing so, we saw a pattern that shed light on limitations of classical approaches. The first is simple: optimizations are not always included in refinement methodologies. Optimizations are essential in Upright's design.

Second, classical approaches are known to have difficulty scaling [48]. To see this, consider Figure 20. Traditionally, one starts with a simple architecture A_1 and progressively extends it to A_2, A_3, and A_4 before refinements $A_4 \to B_4$, $B_4 \to C_4$, and $C_4 \to D_4$ are applied to derive the final architecture D_4.

We observed that the sequence of extensions that are applied to an initial architecture are generally easy to understand—there is little or no detail at this level. (This is the $A_1 \Rightarrow \cdots \Rightarrow A_4$ path). In contrast, *the transformations that are applied to the fully-extended architecture A_4 integrate all of the refinements, optimizations, and extensions that preceded it.* For example, refinement $B_4 \to C_4$ maps the result of composite arrow $A_1 \Rightarrow B_4$ to the result of composite arrow $A_1 \Rightarrow C_4$. While each individual arrow of a composite can be grasped by uninitiated readers, monolithic composites are only understood by domain experts.

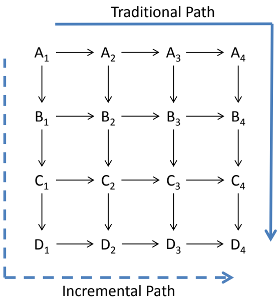


Figure 20: Paths

This observation has practical significance: had we followed the classical approach where a specification is fully elaborated before refinement takes place [52], *we could not have understood, let alone derived, Upright's architecture.* The required (composite) refinements were simply too complicated to understand and explain. *Pushouts were needed to synthesize complicated architectural designs. Their use allows us to understand how "derivation replay" is accomplished in the presence of extensions. We now know what to add and what to change in a derivation.*

Stated bluntly, classical approaches without the aid of pushouts and optimizations cannot describe practical systems.

6. RELATED WORK

The idea of using pushouts is not new in software development. Smith [51] used pushouts to automatically compute how to apply an abstract algebraic specification of an extension to an algebraic specification of an algorithm. Morphisms model how extensions are applied to abstract algebraic specifications, and tactics are used to find possible ways of applying the same extension to concrete algorithms. Smith starts with algebraic specifications, whereas we use pipe-and-filter architectures.

Amphion is a classic example of *Knowledge Base Software Engineering (KBSE)* [38]. It provides a DSL to write an abstract FOL specification (theorem) of the problem to solve. Amphion relies on a domain theory (a formalization of a domain and its implemented components in a library) and uses deductive synthesis to prove the theorem from which a program is extracted. If a theorem is changed, Amphion starts its entire process from scratch. This is possible if one has a rather complete knowledge base. In our case, the set of extensions (features) that can be applied to an architecture is open-ended. And even in the common case where one limits the set of extensions, there is the problem of how to populate the knowledge base with transformations that can derive extended programs. Pushouts explain how refinements of extended architectures can be created.

Pipe-and-filter architectures are a fundamental class of software designs [22]. Practical tools for building PnF architectures have a long history. LabVIEW [44], Weaves [23], and Fractal [10] are platforms for executing PnF architectures. Other component-based systems or languages follow a similar approach [22, 32, 54, 58]. Refinement is their sole abstraction; *user-defined optimizations, extensions, and pushouts are absent.*

Our work is an example of *Model Driven Engineering (MDE)*: we map high-level models of architectures to low-level models of architectures. We rely on *endogenous transformations*—transformations whose domain and co-domain are the same. Most of the MDE literature focuses on *exogenous transformations*—mappings whose domain and co-domain are different [40]. For example, each architectural style has its own metamodel. Prior work mapped architectures of one style to an architecture of another [1, 18, 26, 56]. Endogenous transformations have been used sparingly, *not for incremental architectural development as we do*, but to simulate architecture execution (e.g. adding and removing clients in a client-server system) [11, 41]. The few cases where endogenous transformations are composed to produce MDE designs [55, 57, 62] deal with simple transformations that encode extensions as model deltas, not refinements and optimizations that we present.

Banach et al. [4] propose the concept of *retrenchment* to improve the refinements ability to deal with the complexity of the real world, or the limitations of models closer to the implementation (e.g. the need to handle error situations). Retrenchment may be compared to extensions, as both allow to add new behavior to models/specifications. However, retrenchment deals with exogenous transformations, adding new behavior to specifications in one domain to a more concrete domain.

Burstall and Darlington [12] proposed a transformation system for recursive programs. Their folding and unfolding operations are similar to our abstraction and refinement transformations. They used equational rewrites, whereas we are using graph rewrites. Moreover, their approach did not account for extensions or pushouts, that were essential to our work.

There is a rich collection of papers on architecture refinement; we limit our discussions to key papers for lack of space. Traditional approaches start with an abstract architecture or specification and then apply refinements to progressively expose more hierarchical detail on how the abstract architecture is implemented. Some researchers have shown that their refinements can be verified, not violating any of the original design's properties [7, 26, 41, 42]. Our work differs from traditional refinement in three ways: 1) We start with a simple architecture and apply endogenous transformations to extend it and to expose implementation details. 2) We are unaware of prior work that uses architectural optimizations and pushouts in the development of PnF systems. And 3) the role of extension in SWD architectural design is under-appreciated. Broy seems to lay the mathematical foundations for extension [9], but we are unaware of a recent system that puts his ideas into practice.

Our work is an example of software architecture recovery [17, 33]. Classic research focuses on tools, data exchange formats, and metrics for extracting and clustering information from source (code, makefiles, documentation, etc.) and application execution traces to reconstruct an architecture [19, 28, 39]. Our approach is different: our decomposition is based on semantics and not metrics; our source is our understanding of a domain and that which we can gain from domain-experts, rather than from code and execution traces. Our work is more in line with architectural recovery using MDE, where a system is described by multiple viewpoints (metamodels) and their views (models) [19]. But even here our work is different, as we start with a well-known viewpoint (a pipe-and-filter architecture) and stress the role of transformations to derive an architecture's design.

Some of our refinements are similar to algorithmic skeletons [15], in particular the *map* skeleton [2]. Although some skeleton approaches support optimizations of skeleton compositions [2], they do not provide support for user-defined optimizations, nor extensions.

Finally, our work may eventually be an example of the *correct by construction* paradigm: if the initial architecture is correct, and its transformations are correct, the resulting architecture is correct. Clement et al. have a proof of the *composite* transformation for the ACFT architecture [14], but not proofs for each *individual* transformation.

7. CONCLUSIONS

Automated support for program derivation and program evolution is an important objective: it is widely believed that it will yield a major improvement in the software lifecycle [24]. We seek to reveal a process by which architectural designs can be defined by refinement, optimization, and extension, from which legacy applications can be reconstructed from our models.

In this paper, we explained how we reverse-engineered Upright, a complex state-of-the-art *asynchronous crash fault tolerant (ACFT)* server. We provided strong evidence for the following points: (1) that optimizations and pushouts—in addition to refinement and extension—are essential in software architectural recovery, and (2) that refining fully-specified systems can be difficult or impractical. We found it easier to refine a simple description of a system through a series of progressively more detailed implementations, and then extend each of these implementations using pushouts to "replay derivations" of prior refinements. The value in doing so is that we now know how to populate a knowledge base of transformations, transformations that can be grasped by uninitiated readers, while monolithic composite transformations are only understood by domain experts. To validate our study, we recreated Upright's ACFT and SCFT architectures following the derivation that we presented in this paper.

As of this writing, we are building a domain-independent MDE tool that aids the process described in this paper, and that supports refinements, extensions, and optimizations. This paper provides a foundation to codify domain-specific knowledge in a machine-manipulatable way, which is a necessary foundational step for building semi-automated tools for 1) reverse engineering software architectures from complex, expert-built applications, and 2) recreating applications using these models.

Acknowledgments: We gratefully acknowledge helpful feedback from E. Torlak (MIT), G. Heineman (WPI), G. Karsai (Vanderbilt), E. Hehner (Toronto), H. Vin (Tata Consulting), and A. Clement (Texas). Batory and Riché are supported by the NSF's Science of Design Project CCF 0724979 and NSF's Computer Systems Research Grant CNS 0509338. Marker was supported by sponsored by NSF grant CCF-0917167, a fellowship from Sandia National Laboratories, and an NSF Graduate Research Fellowship under grant DGE-1110007. Gonçalves is supported by Portuguese Science Foundation (FCT) grant SFRH/BD/47800/2008 and FCT project UTAustin/CA/0056/2008.

8. REFERENCES

[1] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Arch. into a Design. In *UML*, 1999.

[2] M. Aldinucci, M. Coppola, and M. Danalutto. Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff. In *CMPP*, 1998.

[3] S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[4] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Sci. Comput. Program.*, 67(2-3):301–329, 2007.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004.

[6] I. D. Baxter. Design Maintenance Systems. *CACM*, April 1992.

[7] J. P. Bernhard and B. Rumpe. Stepwise Refinement of Data Flow Architectures. Technical Report TUM-19746, TU München, 1997.

[8] R. V. Book. Confluent and other types of thue systems. *J. ACM*, 29:171–182, January 1982.

[9] M. Broy. Compositional Refinement of Interactive Systems. *JACM*, 44(6), 1992.

[10] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal Component Model. `http://fractal.ow2.org`, 2004.

[11] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. L. Lafuente. Graph-Based Design and Analysis of Dynamic Software Arch. In *LNCS*. Springer-Verlag, 2008.

[12] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[13] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT, Jan. 2001.

[14] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. L. Riché. UpRight Cluster Services. In *SOSP*, Oct. 2009.

[15] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1989.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, Dec. 2004.

[17] S. Ducasse, D. Pollet, and L. Poyet. A Process-Oriented Software Arch. Reconstruction Taxonomy. In *CSMR*, 2007.

[18] A. Egyed, N. Mehta, and N. Medvidovic. SW Connectors and Refinement in Family Arch. In *IWSAPF*, 2000.

[19] J. Favre. Cacophony: Metamodel-driven architecture recovery. In *WCRE*, 2004.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[21] Code Generator. `http://code.google.com/p/stepwise-ft/`.

[22] D. Garlan. Style-Based Refinement for Software Architecture. In *ISAW*, 1996.

[23] M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *ICSE*, 1991.

[24] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. *Kestrel Institute Technical Report KES.U.83.2*, 1983.

[25] Hadoop. `http://hadoop.apache.org/core/`.

[26] R. Heckel and S. Thöne. Behavior-Preserving Refinement Relations Between Dynamic Soft. Arch. In *WADT*, 2004.

[27] E. Hehner. Predicative Programming Part 1. *CACM*, 1984.

[28] R. Holt, A. Winter, and A. Schurr. GXL: Toward a Standard Exchange Format. In *Reverse Engineering*, Nov. 2000.

[29] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-021, 1990.

[30] I. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.

[31] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[32] 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.

[33] R. Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In *LNCS 5413*, 2009.

[34] R. Kotla. *xbft: Bzyantine Fault Tolerance with High Performance, Low Cost, and Aggressive Fault Isolation*. PhD thesis, The University of Texas at Austin, 2008.

[35] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, Oct. 2007.

[36] L. Lamport. The Part-Time Parliament. *ACM ToCS*, 16(2):133–169, 1998.

[37] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos Register. In *IEEE SRDS*, 2007.

[38] M. R. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *ISMIS*, 1994.

[39] O. Maqbool and H. Babri. Hierarchical Clustering for Software Arch. Recovery. *IEEE TSE*, pages 759–780, 2007.

[40] T. Mens and et al. A Taxonomy of Model Transformations. In *GraMoT*, 2005.

[41] D. L. Métayer. Describing Software Arch. Styles Using Graph Grammars. *IEEE TSE*, 24(7):521–533, 1998.

[42] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architectural Refinement. *IEEE TSE*, 21:356–372, 1995.

[43] S. Mullender, editor. *Distributed Systems*. Addision-Wesley, 2nd edition, 2003.

[44] National Instruments LabVIEW 2011. `http://www.ni.com/labview/`.

[45] Future of Software Engineering Research. `http://www.nitrd.gov/SUBCOMMITTEE%5Csdp%5Cfoser%5CFOSER%20December%202011.pdf`, 2011.

[46] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.

[47] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[48] M. Poppleton. Private Correspondence. Private Correspondence, 2010.

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

[50] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[51] D. Smith. Mechanizing the development of software. In *Proc. of the Marktoberdorf Int. Summer School*, 1999.

[52] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

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

[54] W. Thies. *Lang. and Compiler Support for Stream Programs*. PhD thesis, MIT, Feb. 2009.

[55] S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Develop.: A Case Study for Portlets. In *ICSE*, 2007.

[56] T. Tseng, J. Aldrich, D. Garlan, and B. Schmerl. Semantic Issues in Arch. Refinement. Technical report, CMU, 2004.

[57] M. Volter and I. Groher. Product Line Implementation using AO and MD Software Development. In *SPLC*, 2007.

[58] S. Wang, G. S. Avrunin, and L. A. Clarke. Arch. Building Blocks for Plug-and-Play System Design. In *CBSE*, 2006.

[59] N. Wirth. Program Development by Stepwise Refinement. *CACM*, 1971.

[60] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Servers. In *SOSP*, 2003.

[61] Zookeeper. `http://hadoop.apache.org/zookeeper`.

[62] S. Zschaler and et. al. VML*: A Family of Languages for Variability Management in Software Product Lines. In *SLE*, 2009.