Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery

Nenad Medvidovic	Alexander Egyed	Paul Gruenbacher
Computer Science Department	*Teknowledge Corporation*	*Sys. Eng. & Automation*
University of Southern California	*4640 Admiralty Way*	*Johannes Kepler University*
Los Angeles, CA 90089, USA	*Marina Del Rey, CA 90292, USA*	*4040 Linz, Austria*
neno@usc.edu	aegyed@acm.org	gruenbacher@acm.org

ABSTRACT

Ideally, a software project commences with *requirements gathering and specification*, reaches its major milestone with system *implementation and delivery*, and then continues, possibly indefinitely, into an *operation and maintenance* phase. The software system's *architecture* is in many ways the linchpin of this process: it is supposed to be an effective reification of the system's requirements and to be faithfully reflected in the system's implementation. Furthermore, the architecture is meant to guide system evolution, while also being updated in the process. However, in reality developers frequently deviate from the architecture, causing *architectural erosion*, a phenomenon in which the initial architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold. In this paper, we present an approach intended to address the problem of architectural erosion by combining three complementary activities. Our approach assumes that a given system's *requirements* and *implementation* are available, while the architecturally-relevant information either does not exist, is incomplete, or is unreliable. We combine techniques for *architectural discovery* from system requirements and *architectural recovery* from system implementations; we then leverage architectural styles to identify and *reconcile* any mismatches between the discovered and recovered architectural models. While promising, the approach presented in the paper is a work in progress and we discuss a number of remaining research challenges.

1 INTRODUCTION

Ideally, software systems are developed via a progression starting from *requirements* through *architecture* to *implementation*, regardless of the lifecycle model employed. Any changes to those systems during their, possibly indefinite, lifespans should then follow the same progression: a change in the requirements is reified in the architecture and, subsequently, the implementation. However, frequently neither the initial development process nor the system's evolution and maintenance follow such a path for reasons that include developer sloppiness; requirements that are immediately implemented due to (the perception of) short deadlines; architectural decisions that are violated to achieve non-functional qualities (e.g., improve performance, satisfy real-time constraints, reduce application memory footprint); off-the-shelf (OTS) functionality that is directly incorporated into the system's implementation; and the existence of legacy code that is perceived to prevent careful system architecting.

For these reasons, architectural artifacts are often out of sync with the system's requirements and its implementation, and we say that the architecture is *eroded* [27]. There are many potential problems associated with architectural erosion: difficulties in assessing how well the current implementation satisfies the current requirements; inability to trace a specific requirement to implementation artifacts; lack of understanding the complex effects of changing a requirement; and inadequate system maintainability and evolvability. The incorrect perception of the architecture may lead to incorrect architecture-level and, subsequently, implementation-level decisions in response to new or changing requirements.

To deal with the problem of architectural erosion, researchers and practitioners have typically engaged in *architectural recovery* [2,10,14,15,18,22,28,31,32], where the system's architecture is extracted from its source code. However, existing architectural recovery approaches fail to account for several pertinent issues. They rely primarily on implementation information, leveraging requirements in a limited fashion, if at all. Since the implementation may have violated certain system requirements, they will, in effect, recover incorrect architectures in such cases. In addition, architecturally-relevant decisions are frequently obscured by the implementation. This may be the result of justified implementation-level decisions, such as eliminating processing bottlenecks, removing duplicate modules for efficiency, OTS reuse, and so on. Architectural decisions might also be ignored without justification, due to a missing system-wide view, developer sloppiness, misguided "creativity" in implementing the desired functionality, and so on. Another problem with existing approaches to architectural recovery is their relative heavy weight, a by-product of the lack of reliance on information already present in the system's requirements. Perhaps most importantly, the existing architectural recovery approaches exhibit no understanding of the importance and role of *architectural styles* in developing large-scale, complex software systems. An architectural style is a key design idiom that implicitly captures a large number of design decisions, the rationale behind them, effective compositions of architectural elements, and system qualities that will likely result from the style's use [8,22,29]. Without this knowledge, a system's architecture will present only a partial picture regardless of how faithfully its structural, compositional, behavioral, and/or interaction details are recovered.

Our research goal is to combine software requirements, implementations, and architectural styles in a light-weight and scalable manner to stem architectural erosion. Requirements serve as the basis for discovering a software system's architecture. Implementations serve as the basis for recovering the system's architecture. Because of their different inputs, discovery and recovery are likely to reveal different and possibly incomplete architectural models. Architectural styles can be used to
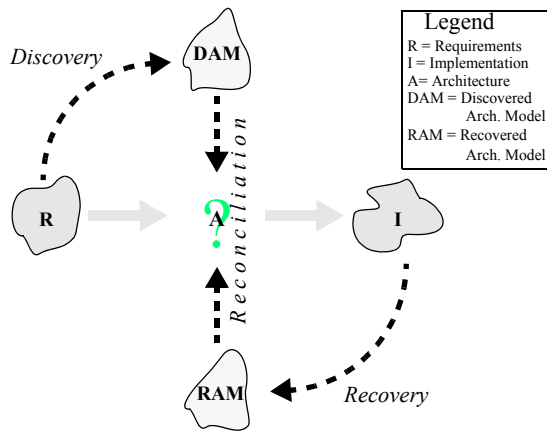
Figure 1. Conceptual view of the approach.

reconcile the two models and combine them into a coherent and more complete model of the software system's architecture. Our approach therefore consists of three interrelated activities as depicted in Figure 1:

1. a technique supporting the *discovery* of an architecture from system requirements;
2. a technique for *recovering* an architecture from system implementations; and
3. an architectural style characterization technique to identify and *reconcile* any mismatches between the discovered and recovered architectural models.

We assume that the existing information about an architecture either does not exist or is unreliable. We also assume that the system's requirements are known and that an inspectable implementation exists. We acknowledge that many modern software systems depend heavily on off-the-shelf libraries (e.g., GUI libraries) or middleware platforms (e.g., CORBA, DCOM). However, deriving architectural properties from such technologies is a challenging task and is thus outside the current scope of our work.

2 BACKGROUND

This work builds on three related areas: software architectures and architectural styles; software requirements, and specifically approaches for mapping requirements to architectural decisions; and architectural recovery.

2.1 Software Architectures and Styles

Software architecture is a level of design that "involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns" [29]. A goal of software architectures is to facilitate development of large-scale systems, preferably by integrating pre-existing building blocks of varying granularity, typically specified by different designers, implemented by different developers (possibly in different programming languages), with varying operating system requirements, and supporting different interaction protocols.

An *architectural style* [8,16,29] is a set of design rules that identify the kinds of building blocks that may be used to compose a system, together with the local or global constraints on the way the composition is done [29]. Styles codify the best design practices and successful system organizations [1,20]. Several architectural styles have been in use for a number of years, including client-server, pipe and filter, blackboard [29], C2 [30], and REST [9].

2.2 Architectural Discovery

Software requirements describe aspects of the problem to be solved and constraints on the solution. Requirements deal with stakeholder goals, options, agreements, issues, and conditions to capture the desired system features and properties. Requirements may be simple or complex, precise or ambiguous, stated concisely or elaborated carefully. Although informal requirements described in natural language often lead to ambiguities and inconsistency, they are frequently used in practice and are thus of special interest in our research.

The relationship between the requirements and architecture for a desired system is not readily obvious. Several existing techniques provide suggestions for addressing the problem. For example, the QUASAR approach [4] relates desired system features (e.g., "The system must be secure.") to solution fragments that effect those features (e.g., "Employ an encryption scheme."). The objective of QUASAR is to allow reuse and compose solution fragments across systems with similar desired features. However, this work has only recently begun addressing the relationship of desired features and software architectures. ATAM [17], a technique that supports the evaluation of architectural decision alternatives in light of non-functional requirements, has a similar limitation. Twin Peaks [25] attempts to overcome the separation of requirements specification and design activities by intertwining them. However, unlike our approach, Twin Peaks does not take into account the implementation. Brandozzi and Perry [3] have recently coined the term "architecture prescription language" for their extension of the KAOS goal specification language [19] to include architectural dimensions. Their approach has the same limitations as our architectural discovery technique: they are unable to suggest a complete architectural configuration based on the information extracted from the requirements, and they currently make no use of non-functional requirements in modeling the discovered architecture. This is why we have decided to couple architectural discovery, recovery, and styles.

Finally, a key issue in transforming requirements into architecture and other software models is traceability. Researchers have recognized the difficulties in capturing development decisions across software models [11]. In response to this, Gotel and Finkelstein [12] suggest a formal approach for ensuring the traceability of requirements during development.

2.3 Architectural Recovery

A number of existing approaches focus on recovering a software architecture from source code. ARM [14] is an

approach to architectural reconstruction distinguishing between the conceptual architecture and the actual architecture derived from source code. ARM applies design patterns and pattern recognition to compare the two architectures. Unlike our architectural recovery approach, ARM assumes the availability of system designers to formulate the conceptual architecture. Similarly to our recovery approach, software reflexion models [24] treat a system's architecture from two perspectives: the idealized, high-level view and the low-level view derived from source code. Reflexion models support incremental architectural recovery to analyze whether varying sets of relationships hold between the idealized and actual architectures. However, reflexion models do not make direct use of architectural concepts such as styles and connectors.

MORALE [28] is an approach for evolving legacy software systems developed with procedural languages. COREM [10] is an approach that converts procedural into object-oriented systems via four steps: design recovery, application modeling, object mapping, and source code adaptation. Neither of these approaches provides a means for determining whether the implemented systems completely and correctly satisfy their original requirements, or whether the requirements themselves are complete and consistent.

Recently, a series of studies has been undertaken to recover the architectures of several open-source applications [2,15]. The approach taken in these studies has been to come up with a conceptual architecture from a system's documentation and use it as the basis for understanding the system's implementation. The system documentation is assumed to be correct when, in fact, both the documentation (e.g., requirements) and implementation may be partially incorrect, incomplete, or internally inconsistent. As with all of the above approaches, architectural style information is not leveraged during recovery.

3   EXAMPLE APPLICATION

To illustrate the discussed concepts we use ShareDraw, an application implemented in Visual C++. ShareDraw is an extension to the DrawCli application, which is provided as part of the Microsoft Foundation Classes (MFC) release. DrawCli allows users to manipulate 2-D graphical objects (lines, ovals, polygons). ShareDraw extends DrawCli into a
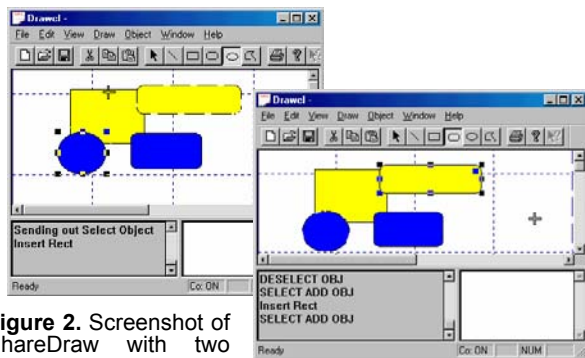

Figure 2. Screenshot of ShareDraw with two clients shown.

distributed application that adds collaborative drawing and chatting facilities, as depicted in Figure 2.

The architecture of ShareDraw was not available to us. Similarly, DrawCli's requirements were not available. However, given the highly interactive nature of the application, we can easily extract many of the functional requirements from the application's observed behavior. The requirements for the extension of DrawCli into ShareDraw were available. Several informally stated requirements describing some commonly performed operations are as follows:

> $Reqt_1$: ShareDraw should allow the user to save drawings for later retrieval.
>
> $Reqt_2$: ShareDraw should make object manipulation operations easily accessible to the user.
>
> $Reqt_3$: ShareDraw should allow the user to group and simultaneously manipulate multiple drawing objects.
>
> $Reqt_4$: ShareDraw should allow the user to instantly view the actions of all other users.

4   THE APPROACH

The goal of our research is to develop a generally applicable, style-centered approach for integrating architectural discovery and recovery techniques, and reconciling the identified differences. Our approach will comprise three separate, but complementary techniques, as depicted in Figure 1:

1. an architectural style-based technique for architectural discovery from software requirements,
2. an architectural style-based technique for architectural recovery from software implementations, and
3. a technique that leverages styles to reconcile the results of discovery and recovery.

4.1   Architectural Discovery

Elaborating system requirements into a viable software architecture satisfying those requirements is often based on intuition [25]. Software engineers face some critical challenges in performing this task [13]:

- Requirements are frequently captured informally in a natural language, while software architectures are usually specified formally [21].
- Non-functional system requirements are hard to capture in an architectural model [21].
- Mapping requirements into architectures and maintaining their inter-consistency is complicated since a single requirement may address multiple architectural concerns and vice versa.
- Large-scale systems have to satisfy hundreds, possibly thousands of requirements, making it difficult to identify and refine the architecturally relevant information contained in the requirements.

To address these challenges we developed CBSP [13], a light-weight technique to distill from the system requirements the key architectural elements and the dependencies among them. The result of the technique is an intermediate model between the requirements and architecture that con-

tains the essence of architectural information embedded in the requirements. This model is referred to as the discovered architectural model, or DAM. The CBSP approach creates DAM in a structured process using conflict resolution to address ambiguities in the requirements. The process consists of three main activities, detailed in [13]:

1. classify architecturally relevant requirements,
2. identify and resolve classification inconsistencies, and
3. refine/restate architecturally relevant requirements.

In this section we detail the DAM model itself.

4.1.1 Discovered Architectural Model

The basic idea behind our approach to architectural discovery is that any software requirement may explicitly or implicitly contain information relevant to the software system's architecture. It is frequently very hard to surface this information, as different stakeholders will perceive the same requirement in very different ways. CBSP captures this information in the intermediate DAM model. DAM is structured around a simple set of general architectural concerns derived from existing software architecture research [21,27,29]:

- *Components* provide application-specific functionality. They may be *data* or *processing* components [27].
- *Connectors* facilitate and govern all interactions among the components.
- *Configuration* of a system or a particular subsystem describes the relationships and organization among multiple (possibly all) components in the system.
- *Properties* describe the non-functional characteristics of individual components and connectors, or the entire configuration.

Thus, each derived DAM element explicates an architectural concern and represents an early architectural decision for the system. For example, a requirement such as

> *Reqt: The system should provide an interface to a Web browser.*

can be recast into a DAM processing component element and a DAM connector element

> $Comp_p$: *A Web browser should be used as a component in the system.*
> $Conn$: *A connector should be provided to ensure interoperability with third-party components.*

Because of the complexity of the relationship between requirements and architecture, DAM gives a software architect leeway in selecting the most appropriate *refinement* or, at times, *generalization* of one or more requirements. Examples of both refinement and generalization are given below.

There are seven possible DAM dimensions discussed below and illustrated with simple examples from the Share-Draw application. The seven dimensions involve the basic architectural constructs and, at the same time, reflect the simplicity of our approach.

(1-2) $Comp_p$ and $Comp_d$ are model elements that describe or involve an individual processing or data component in an architecture, respectively. For example

> *Reqt: The system should allow the user to directly manipulate graphical objects.*

may be refined into DAM elements describing both processing components and data components

> $Comp_p$: *Graphical object manipulation component.*
> $Comp_d$: *Data for abstract depiction of graphical object.*

(3) *Conn* are model elements that describe or imply a connector. For example

> *Reqt: Manipulated graphical objects must be stored on the file system.*

may be refined into

> *Conn: Connector enabling interaction between UI and file system components.*

(4) *Conf* are model elements that describe system-wide features or features pertinent to a large subset of the system's components and connectors. For example

> *Reqt:Allow independent customization of application look-and-feel and graphical object manipulation tools.*

may be refined into

> *Conf: Strict separation of graphical object manipulation, visualization, and storage components.*

(5) $Prop_{Comp}$ are model elements that describe or imply data or processing component properties, such as reliability, portability, incrementality, scalability, adaptability, and evolvability. For example

> *Reqt: The user should be able to view the effects of his actions with minimal perceived latency.*

may be refined into

> $Prop_{Comp}$: *Graphical object manipulation component should be efficient, supporting incremental updates.*

(6) $Prop_{Conn}$ are model elements that describe or imply connector properties. For example

> *Reqt: The system should support loading of graphical manipulation tools at runtime.*

may be refined into

> $Prop_{Conn}$: *Robust connectors should be provided to facilitate runtime component addition and removal.*

(7) $Prop_{Conf}$ are model elements that describe or imply system (or subsystem) properties. For example

> *Reqt:The system must support collaborative editing of graphical objects.*

may be transformed into

> $Prop_{Conf}$: *The system should be distributable.*

Note that, e.g., the $Prop_{Conn}$ example (5) involved refining a general requirement into a more specific DAM element. On the other hand, the $Prop_{Conf}$ example (6) involved the generalization of a specific requirement into a more general DAM artifact. In fact, in both cases multiple DAM artifacts may be produced as part of a single requirement. We are currently studying this issue with the goal of providing practical guidelines to architects engaging in this task.

4.1.2 Summary and Open Issues

At this point, we have an intermediate model, DAM. DAM classifies the key architectural concerns into seven categories: data components, processing components, con-

nectors, configurations, component properties, connector properties, and (sub)system properties. DAM is still stated in a requirements-like notation, such that it can be verified against the intentions of the system's non-architect stakeholders (e.g., customers). DAM reinterprets architecturally relevant requirements; no requirements are actually changed aside from clarifications that arise during the discovery process. Finally, DAM classifies and describes the system's architecturally relevant information in a way that makes it much easier to derive an architecture, and, subsequently, implementation from it would be from "raw" requirements.

However, a remaining problem is that the DAM elements provide a very low-level view of the architecturally relevant system requirements (recall the above examples). It may not be straightforward to map some aspects of DAM (e.g., configuration information, properties) into an effective architecture that will realize them. For example, in our experience architectural discovery is often unable to infer all interdependencies between architectural elements. This directly motivates the need to introduce additional information into the picture, as further discussed below.

4.2 Architectural Recovery

Architectural recovery complements architectural discovery by highlighting the major structural characteristics of the implemented system: data and processing components, connectors, and configuration. The result of architectural recovery is a recovered architectural model, or RAM. In this section we discuss the process of generating RAM. Later we will show how this information can be coupled with DAM to arrive at a more complete architectural model. We use UML to represent the recovered architecture.

4.2.1 Recovered Architectural Model

Our proposed architectural recovery technique will consist of the following four simple activities.
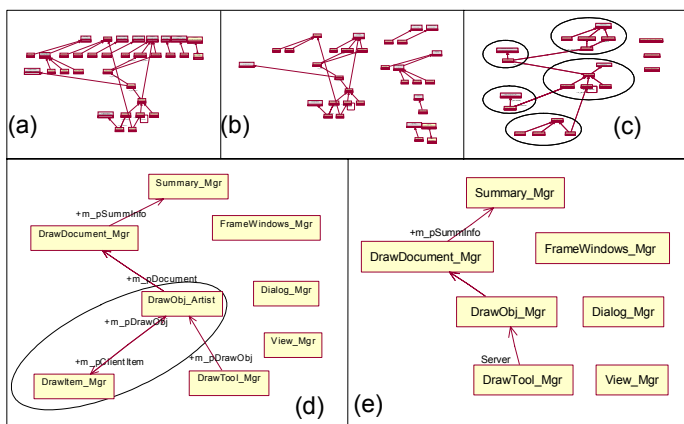
Figure 3. Identifying components from a UML class diagram. At this magnification, the top three diagrams are shown only for illustration, to convey the scope of the task.

Generate class diagrams. Numerous tools are available to infer class diagrams from source code automatically; the engineer need not even look at the system's source code to accomplish this step. Figure 3a shows the class diagram of ShareDraw's client subsystem, automatically generated by Rational Rose®.

Group related classes. Typically, a large number of implementation classes are required to implement individual architectural components and connectors. Classes can be grouped based on different criteria and/or architectural concerns. Multiple architects may participate in this process and, consequently, disagreements and mismatches may arise. The diagrams in Figure 3b-e show one possible such grouping of ShareDraw's classes, obtained by applying the three simple rules adopted from our Focus technique [6]:

- Classes isolated from the rest of the diagram comprise one grouping (Figure 3b).
- Classes that are related by generalization (i.e., inheritance) comprise additional groupings, as do classes related by aggregation and composition (Figure 3c).
- Finally, classes with two-way associations are grouped together since they denote tight coupling (Figure 3d).

Package groups of classes into architectural elements. Clusters of classes identified in the previous stage are packaged together into processing components, connectors, or their relationships (links). These elements can be further aggregated into even larger elements. Using this process, ShareDraw's client implementation is abstracted into seven components and three inter-component links (Figure 3e), as well as two remote procedure call (RPC) connectors. The connectors are not shown in Figure 3 since UML and Rational Rose® provide no mechanisms for distinguishing connectors from components. Figure 3 also does not show data components as introduced by Perry and Wolf [27] and discussed in Section 4.1. Data components may be extracted from the processing components' states and interfaces based on varying desired criteria (e.g., all class variables, all public method parameters, or both).

Determine partial system configuration. The relationships among the components identified in the preceding steps reflect the system's configuration. The configuration information may be incomplete in cases where the components do not interact in easily detectable ways (e.g., access to shared implementation substrate classes, implicit invocation, distributed interaction, and so on). Figure 3e shows only a partial configuration of ShareDraw's client: the topological relationship of the *FrameWindows_Mgr*, *Dialog_Mgr*, and *View_Mgr* components with the remaining components has not been identified in this process; in addition, the diagram does not identify the connectors for reasons discussed above.

4.2.2 Summary and Open Issues

The described architectural recovery technique is very simple and scalable, relying only on structural manipula-

tion of the system's implementation. The outcome of the technique is the RAM model, i.e., the collection of existing system's major processing and data components, its connectors, as well as a partial architectural configuration. RAM is intended to map to the structural aspects of DAM proposed in Section 4.1.

The result of the recovery step is not a complete "as is" architecture of the system. Several pieces of information are still missing. As discussed above, the architectural configuration information will likely be incomplete. In addition, similarly to a great majority of the existing recovery techniques (e.g., [2,10,14,15,18,28,31,32]), our proposed approach does not take into account non-functional properties. This shortcoming suggests the next step of our approach: by coupling the information represented in RAM and DAM with architectural style information, we can mitigate this problem and present a more complete picture of the architecture, as discussed below.

4.3 Reconciling Discovery and Recovery

The above two techniques provide related, though disconnected models of the system's architecture, as depicted in Figure 1. The requirements are refined and rephrased into DAM elements along the seven dimensions representing the key architectural concerns: data and processing components, connectors, configurations, component properties, connector properties, and (sub)system properties. The implementation is abstracted into four types of RAM elements: data and processing components, connectors, and (partial) configurations. This section discusses how the two

models can be "matched up" to derive a more complete architecture based on their combined information.

4.3.1 Determining Appropriate Architectural Styles

As discussed earlier, architectural styles [8,16,29] provide rules that exploit recurring structural and interaction patterns across a class of applications. Styles constrain architectural models syntactically and semantically. In order to select the appropriate style(s) for the given application, we propose to classify existing architectural styles across a set of commonly recurring dimensions. Our goal is to provide the foundation of a classification that is rich enough to allow us to effectively represent and select styles based on the given DAM and RAM models.

Our preliminary study of architectural styles [22] has identified the following seven dimensions as a good candidate set for effectively describing styles.

1. the types of *data* exchanged between style elements;
2. the *structure* of the elements allowed in a style;
3. the allowed *topologies* of architectural elements;
4. the allowed *behavior* of a style element;
5. the types of supported *interactions* between style elements and their allowed specializations;
6. the key *non-functional properties* especially enabled by the style; and.
7. the style's *domain* of applicability.

Table 1 depicts the result of an exercise in which we mapped four commonly occurring styles using this framework. This experience has indicated several challenges that

Table 1: Characterization of Four Architectural Styles

	Data	Structure	Topology	Behavior	Interaction	Properties	Domain
C2	Discrete events	Separable components	Limited component dependencies	Exposed via named services only	Asynchronous coordination	Distributability	GUI Systems
	Data tuples	Explicit connectors	Partially ordered connectivity-based "top" and "bottom" relations	Data queueing and buffering by connectors	Implicit invocation	Heterogeneity	
					Event-based interaction	Composability	
			Dynamic creation of connections	Multi-tasking mechanisms such as threads	Direction-oriented events propagated to topology-based recipients	Dynamicity	
Client-server	Parameterized request	Independent servers	Many-to-many connections among clients and servers	Listening server	Server location	Distributability	Distributed Systems
				Connections setup and teardown	Remote connection and communication protocol		
		Specialized clients		Buffering and queueing of requests	Implicit server invocation	Security	
	Typed response		Dynamic creation of connections	Multi-tasking mechanisms such as threads	Data marshalling and unmarshalling	Evolvability	
		Distributed protocol stacks			Client call synchronization	Heterogeneity	
				Exposed via named services only	Request-response protocol		
Pipe-and-filter	Streams of typed records	Explicit pipes and filters	Stream between a pipe and a filter	Stream transformation state machine	Synchronization between filter reads and writes	Heterogeneity	Dataflow systems
		Input and output ports on filters	No two sources or sinks connected to the same port instance		Propagation of stream contents to sinks	Reusability	
		Sources and sinks on pipes		Data buffering by pipes		Composability	
Push-based	Channel notification	Independent producers	Producers connected only to distributors	Content filtering in distributors	Distributor location	Distributability	Distributed systems
		Explicit distributors		Buffering and queueing by distributors	Remote connection and communication protocol	Scalability	
	Subscription request	Channel access/subscribers	Many-to-many channels among receivers and distributors	Subscription setup	Data marshalling and unmarshalling	Robustness	
		Receiver user interface		Content storage/expiration	Distribution policy	Security	
					Implicit invocation		
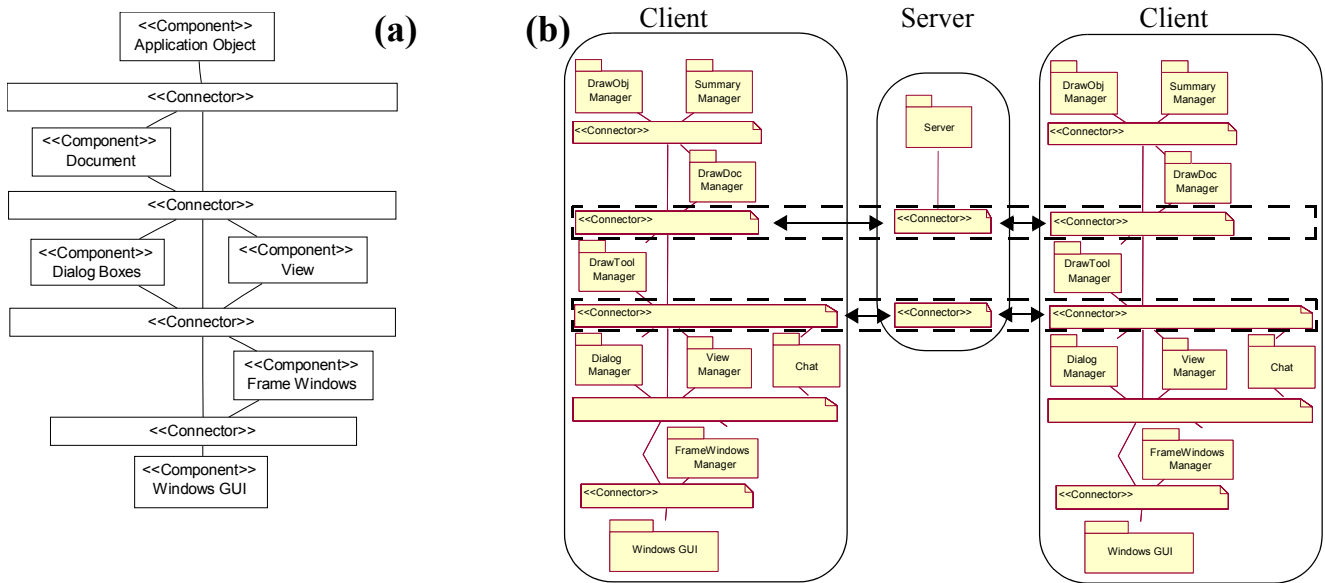
Figure 4. *(a)* "As intended" architecture of the ShareDraw client subsystem. A number of details have been elided for brevity. *(b)* Final architecture of the ShareDraw application, shown with two client subsystems (each roughly equivalent to the Draw-Cli application). The two highlighted connectors use RPC to communicate across processes.

we will need to address. First, we will need to carefully specify a large, representative if not complete, set of existing architectural styles. This process will help us test our hypothesis that the seven dimensions are sufficient to uniquely and richly describe a style. Second, we will need to characterize each style in a manner that will simplify the task of relating the information contained in DAM and RAM models to the information contained in Table 1. Third, we will need to address situations in which multiple styles are highlighted in this process as plausible candidates. A related issue is dealing with situations in which multiple styles are most appropriate to use *in tandem* for a given problem.

In fact, we indeed selected two styles in our ShareDraw example application: client-server to handle the distributed, coarse-grained aspects of the application, and C2 for its ability to compose the GUI-intensive application components within each client and server. This choice was aided by several factors, including our familiarity with these two styles, the fact that we had used them together in the past, the relatively small number of styles we had considered (e.g., Table 1 only includes four styles), and some domain properties (distribution and GUI aspects) that clearly mapped to these two styles. We envision this to be a much greater challenge in a more general setting. Our future work will include identifying conditions and situations under which specific combinations of styles are (dis)allowed. This is a non-trivial problem that deserves particular attention.

4.3.2 Integrating DAM and RAM

Once we have determined the suitable architectural style(s), we can integrate the, still separate, DAM and RAM models into a single integrated model. There are three possible approaches to accomplishing this step:

1. Apply the style information to DAM to derive an "as intended" architecture, and then "map" the information from RAM onto this architecture.
2. Apply the style information to RAM to derive an "as implemented" architecture, and then "map" the information from DAM onto this architecture.
3. Integrate DAM and RAM into an "as extracted" architecture, and then apply the style information to the integrated model.

We are currently investigating the respective benefits and drawbacks of the three approaches.

The "as intended" architecture of the ShareDraw application, obtained by integrating DAM and architectural style information as discussed above, is given in Figure 4a. The complete architecture, obtained by mapping the information contained in RAM to the "as intended" architecture, is shown in Figure 4b. As mentioned above, the final Share-Draw architecture combines the client-server and C2 styles.

Irrespective of the chosen integration approach, integrating DAM and RAM requires knowledge about how their elements interrelate. Although both DAM and RAM present architectural perspectives, they may be inconsistent, e.g., in the element names or level of architectural detail. The two models will thus need to be reconciled. Various interesting reconciliation scenarios can be envisioned. For example, a single RAM element may map to multiple DAM elements, and vice versa. It is also possible that no obvious relationship can be established between an element in one of the models and the other model's elements. We will carefully study these scenarios.

5 CONCLUSION

This work described in this paper is motivated by the observation that architecturally-relevant information is

readily available in a system's requirements and its implementation, although not always in an obvious form. This information can then be uncovered and used to help stem architectural erosion. The information captured in a system's requirements is high-level, possibly imprecise, but rich in human stakeholders' insights and rationale; this information often *suggests* the suitable architectural style(s) for the system. On the other hand, the information contained in a system's implementation is low-level, precise, and rich in detail; this information *reflects* the style(s) applied in the system's construction. We postulate that neither of the two sources of information should be considered complete or correct by itself. Instead, we propose that they be combined using the three presented techniques: architectural discovery, recovery, and reconciliation.

The work described in this paper is on-going. We have already identified several open issues that will frame our future research. In addition, we envision that the combination of the three techniques will likely result in a self-adjusting process in which the architecture is already known to be incorrect and/or incomplete, but, in addition, neither the requirements nor the implementation need be assumed correct or complete. Furthermore, the proposed approach will result in clearly specified and maintainable traceability links across the requirements, architecture, and implementation. We plan to adopt existing techniques (e.g., [7,26]) to capture and manage the traceability links.

6 REFERENCES

[1] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[2] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. *ICSE'99*, Los Angeles, CA, May 1999.

[3] M. Brandozzi and D. E. Perry. Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. *ICSE 2001 STRAW Workshop*, Toronto, May 2001.

[4] H. de Bruin, H. van Vliet, and Z. Baida. Documenting and Analyzing a Context-Sensitive Design Space. *WICSA-3,* August 2002, Montreal.

[5] A. Dardenne, S. Fickas, A. van Lamsweerde. Goal-Directed Concept Acquisition in Requirement Elicitation. *6th Int. Workshop on Software Specification and Design*, Oct. 1993.

[6] L. Ding and N. Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. *WICSA-2*, Amsterdam, August 2001.

[7] A. Egyed. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*. Accepted, to appear.

[8] R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UC Irvine, 2000.

[9] R. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ICSE 2000*, Limerick, June 2000.

[10] H. Gall, R. Klosch, and R. Mittermeir. Object-Oriented Re-Architecting. *ESEC-5*, Berlin, Sep. 1995.

[11] L. R. Gieszl. Traceability for Integration. *2nd International Conference on Systems Integration*, pp. 220-228, 1992.

[12] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. *1st International Conference on Rqts. Eng.*, pp. 94-101, 1994.

[13] P. Gruenbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures: The CBSP Approach. *5th IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, August 2001.

[14] G. Y. Guo, J. M. Atlee, and R. Kazman. A Software Architecture Reconstruction Method. *WICSA-1*, San Antonio, Feb. 1999.

[15] A. E. Hassan and R. C. Holt. A Reference Architecture for Web Servers. In *Working Conference on Reverse Engineering*, Brisbane, Australia, Nov. 2000.

[16] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. *ESEC/FSE'99*, Toulouse, Sep. 1999.

[17] R. Kazman, et al. Experience with Performing Architecture Tradeoff Analysis. *ICSE'99*.

[18] R. Kazman and J. Carriere. View Extraction and View Fusion in Architectural Understanding. In 5th *International Conference on Software Reuse*, Canada, June 1998.

[19] A. van Lamsweerde, R. Darimont, E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering, *IEEE Transactions on Software Engineering*, vol. 24, No. 11, Nov 1998.

[20] A. MacDonald and D. Carrington, Guiding Object-Oriented Design. *TOOLS'98,* Melbourne, Nov. 1998.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93 (January 2000).

[22] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural Style Requirements for Self-Healing Systems. *1st Workshop on Self-Healing Systems*, Charleston, Nov. 2002.

[23] G. Mullery. CORE: A Method for Controlled Requirements Specification. *ICSE4*, Munich, Germany, Sept. 1979.

[24] G. Murphy, D. Notkin and K. J. Sullivan, Software Reflection Models: Bridging the Gap between Design and Implementation, *IEEE Transactions on Software Engineering, vol. 27,* no. 4, April 2001, pp. 364-380.

[25] B. Nuseibeh, Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115-117, March 2001.

[26] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, Oct. 1994.

[27] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[28] S. Rugaber. A Tool Suite for Evolving Legacy Software. In *ICSM'99*, Oxford, England, Aug/Sep 1999.

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

[30] R.N. Taylor, N. Medvidovic, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6), June 1996.

[31] V. Tzerpos and R. C. Holt. A Hybrid Process for Recovering Software Architecture. In *CASCON'96*, Toronto, Nov. 1996.

[32] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, Jan. 1995.