Using Software Evolution to Focus Architectural Recovery

Nenad Medvidovic and Vladimir Jakobac
Computer Science Department
University of Southern California
Los Angeles, CA 90098-0781 USA
{neno,jakobac}@usc.edu

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 technical 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, "as documented" architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold. *Architectural recovery* is a process frequently used to cope with architectural erosion whereby the current, "as implemented" architecture of a software system is extracted from the system's implementation. In this paper we propose a light-weight approach to architectural recovery, called *Focus*, which has three unique facets. First, Focus uses a system's evolution requirements to isolate and incrementally recover only the fragment of the system's architecture affected by the evolution. In this manner, Focus allows engineers to direct their primary attention to the part of the system that is immediately impacted by the desired change; subsequent changes will incrementally uncover additional parts of the system's architecture. Secondly, in addition to software components, which are the usual target of existing recovery approaches, Focus also recovers the key architectural notions of software connector and architectural style. Finally, Focus does not only recover a system's architecture, but may in fact rearchitect the system. We have applied and evaluated Focus in the context of several off-the-shelf applications and architectural styles to date. We discuss its key strengths and point out several open issues that will frame our future work.

Keywords

Software architecture, architecture recovery, architecture erosion, evolution, architectural style, Focus

1 INTRODUCTION

Software lifecycle models (e.g., waterfall, spiral) and development methods (e.g., functional decomposition, object-oriented) differ in the details of how developers arrive at a software system. However, to one extent or another, all of them assume that 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 [29,36]: it is supposed to be an effective reification of the system's technical requirements and to be faithfully reflected in the system's implementation. Furthermore, the architecture is meant to guide system *evolution* [1,21], while also being updated in the process.

However, this is an idealized picture. During the past decade alone, software development has undergone dramatic changes. Object-orientation (OO) has become the dominant development methodology, resulting in

widely-embraced modeling notations (e.g., UML [3]), programming languages, or OOPLs, (e.g., Java [38]), distributed computing technologies (e.g., CORBA [28]), and reusable libraries of software components (e.g., Microsoft Foundation Classes, or MFC [23]). These advances have been accompanied by an increased frequency with which software is evolved ("development and evolution on Internet time"). Software evolution requires in-depth understanding of an application, its complexity, its overall architecture, its major components, and their interactions and dependencies [2]. Many of these requirements are often ignored in present-day development however, with a frequent willingness to compromise the quality and longevity of a system in order to decrease its time-to-market. This attitude, coupled with the complexity of the involved systems and the sloppiness with which the changes to them are often documented, directly contributes to numerous recorded cases of *architectural erosion* [4,5,8,29]. Architectural erosion denotes a major departure from the initial architecture's intent and conceptual integrity, caused by its frequent modifications; erosion also refers to the discrepancies between the architecture "as documented" and "as implemented." Evolving a system with an eroded architecture poses tremendous challenges to engineers and results in a real danger that the modifications intended to provide new functionality will be implemented incorrectly and those intended to remove a particular problem in the existing system will cause other, unforeseen problems.

To deal with the problem of architectural erosion, researchers and practitioners have typically engaged in *architectural recovery* [4,11,13,15,16,32,41,43], a process whereby the system's architecture is extracted from its implementation. However, existing architectural recovery approaches fail to account for several pertinent issues. One such issue is that a system's implementation will frequently obscure or violate architecturally-relevant decisions, either as a result of justified implementation-time decisions (e.g., eliminating processing bottlenecks or removing duplicate modules for efficiency) or without justification (e.g., developer sloppiness or misguided "creativity" in implementing the desired functionality). Another problem with existing approaches to architectural recovery is their goal of *completely* recovering a system's architecture, frequently resulting in heavy-weight techniques. Perhaps most importantly, the existing architectural recovery approaches exhibit little 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 [10,24,35,36]. 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.

For these reasons, we propose *Focus*, an approach to *evolving* OO systems that, as a by-product, also *recovers* their actual *architectures*. The architectures are recovered *incrementally*: only those parts of an

application affected by a given change are modified and their architecturally relevant characteristics extensively studied and documented (hence the name "Focus"); the recovery of additional subsystems' architectures will occur only as new modifications that pertain to those subsystems are required. With each new modification, the task of recovering the architecture of the relevant subsystem and enacting the change becomes easier since a larger portion of the overall system's architecture is known and correctly documented.

A unique aspect of Focus is that it recovers all three basic architectural building blocks: data components, processing components, and connectors [29]. Another unique aspect of Focus is its recognition that, while *implementing* a system using an OOPL may be the best solution, OO is not necessarily the ideal *style* for understanding, capturing, and modifying the system's *architecture* [29,36]. Instead, in Focus, the architecture is captured using the style deemed most appropriate to the characteristics of the application and its domain. Thus, for example, even though a distributed application is implemented in Java or C++, the architectural style of that application may be client-server [36]. In other words, Focus differentiates between a system's *logical* and *physical* architectures [19].[1] In fact, Focus allows an engineer to incrementally *rearchitect* the system during its evolution. This aspect of Focus in particular differentiates it from a number of related approaches that only consider a system's physical architecture directly recovered from its source code [16,32,43].

The Focus approach is based on the assumption that little or no *reliable* documentation exists for the system being modified. In some cases, existing understanding of similar systems (e.g., a documented "reference" architecture for applications built in a particular domain [15]) can be used to aid architectural recovery and evolution in Focus. Additionally, we make a minimal set of requirements:

1) The details of the desired modification are known.
2) *User-level* properties of the application are known.
3) The basic architectural characteristics of the underlying implementation platform (or *substrate*) are understood. An example of this requirement is familiarity with the class hierarchy of the AWT graphics toolkit [6] used in a Java application.

We believe these requirements to be reasonable. The first requirement is not specific to Focus, but is relevant to any software modification task. In the case of a large class of applications, the second requirement can be fulfilled relatively simply, by using the application and observing its behavior. The third requirement may present a somewhat greater challenge to a developer who is unfamiliar with the given implementation

[1] The logical and physical views of an architecture are often also referred to as *conceptual* and *implementation* architectures, respectively.

platform. However, it is necessitated by the reliance of present-day software development on an increasing body of libraries, frameworks, and middleware solutions.

Our ultimate goal is to evaluate Focus and assess the extent of its applicability across applications of varying sizes, belonging to arbitrary domains, and employing many different architectural styles. However, for practical reasons we have had to narrow down the scope of our evaluation to date. As already discussed, we are currently working with applications developed in OOPLs. We have conducted five case studies on small-to-medium sized systems, not counting their implementation substrates (e.g., MFC); two additional case studies have been conducted by external teams [37]. We have relied primarily on (combinations of) three architectural styles in our case studies: client-server, pipe and filter, and C2 [39]. Clearly, further experience is needed before we can fully and definitively assess Focus. However, our evidence to date indicates that Focus is an effective approach to architectural recovery in the explored settings, and suggests broader applicability.

The remainder of the paper is organized as follows. The details of the Focus approach are discussed and illustrated with an example in Section 2. Three additional case studies are outlined in Section 3. We evaluate Focus based on our experience to date and discuss several open issues in Section 4. Section 5 presents related work, while Section 6 concludes the paper.

2 THE FOCUS APPROACH

The Focus approach is driven by evolution requirements and applied iteratively. Each iteration is composed of two interrelated steps: *architectural recovery* and *system evolution*. Using this approach, the architecture of the original system is partially recovered, evolved to address new requirements, and enriched with detail in an incremental fashion.

To illustrate the details of Focus, we will use an example application. The application, DrawCli, is implemented in Visual C++ and provided as part of the MFC release. DrawCli provides an editor that allows users to manipulate 2-D graphical objects (lines, rectangles, polygons, etc.). The specific evolution requirement we used as the basis for applying Focus to this system is to extend DrawCli into a collaborative application that allows concurrent drawing and editing of graphical objects by multiple, distributed users. The new system, called ShareDraw, must also provide some level of group awareness, such as recording one user's actions (e.g., movement, object selection) on another user's screen. A subsequent evolution requirement was to extend ShareDraw with a "chat" facility to rapidly discuss issues during collaborative editing.

The details of the application, recovery of its architecture, and its evolution are discussed below. We use UML [3] as the notation for expressing different activities and artifacts in Focus. For exposition purposes, we discuss the two major steps of Focus (recovery and evolution) separately. However, we should reiterate that the two steps are in fact interrelated and applied repeatedly in an iterative fashion. Additionally, although the two modifications (support for collaborative editing and for chatting) were achieved in two successive applications of the Focus approach, in the interest of brevity we will discuss their details together whenever appropriate.

2.1 Architectural Recovery

The architectural recovery step is composed of six activities, as illustrated in Figure 1. These activities are divided into two categories: (1) *logical* and (2) *physical architecture recovery*. The former starts with an *idealized*, high-level model of the architecture inferred from the selected architectural style, the application's user-level behavior, and its implementation substrate; it focuses on specific parts of the architecture affected by the required change and tries to *refine* them by integrating more concrete details. The latter starts with the source code and tries to *abstract* it to get the *actual* components and their interactions in the implementation, in order to inform and influence the architectural refinement activities. It is important to point out that the goal of this process is not to arrive at a software system's architecture that is fully "correct" in relation to the implemented system. Such a property would be very difficult to verify given the likely complexity of the system in question and the lack of architecturally relevant documentation. Rather, our goal is to arrive at an architecture that "makes sense" for the system in question. In fact, we assume that the resulting architectural model will have inconsistencies in relation to the implementation, i.e., that it will be only *partially* "correct." Any such inconsistencies will be discovered and eliminated *incrementally* as different portions of the system are studied in depth in order to effect different evolution requirements (see Section 2.2).

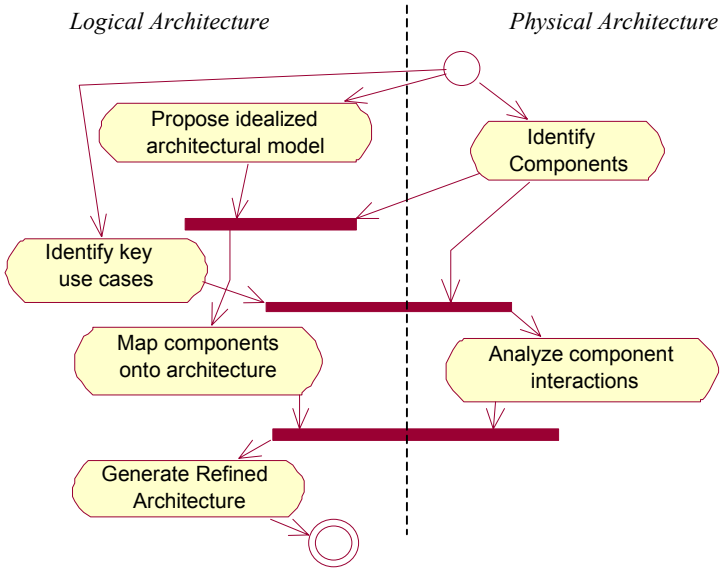

Figure 1. The architectural recovery step of Focus.

The following is a detailed explanation of the activities depicted in Figure 1. Note that the activities and their dependencies form a partially ordered graph. Therefore, as highlighted in activity headings, the order in which we discuss the activities does not always imply dependency, but rather follows a logical progression.

A – Identify Components (independent activity)

The first step in generating the initial logical architecture for a system consists of identifying the coarse grained components from the source code. Focus identifies both the processing components and, optionally, the data components in the system. Processing components are present in all existing architectural modeling approaches and languages [21]. On the other hand, data components are considered an integral part of software architectures [29], but are absent from a majority of the modeling languages.

Identify Processing Components – This step can be further divided into three stages:

(1) Generate class diagrams. No class attributes and methods need to be identified at this stage. However, three kinds of static relationships among classes should be captured in the diagram: association, generalization, and aggregation. A number of tools are available to infer class diagrams from the source code automatically. Figure 2a shows the class diagram of DrawCli, generated automatically by Rational Rose®.

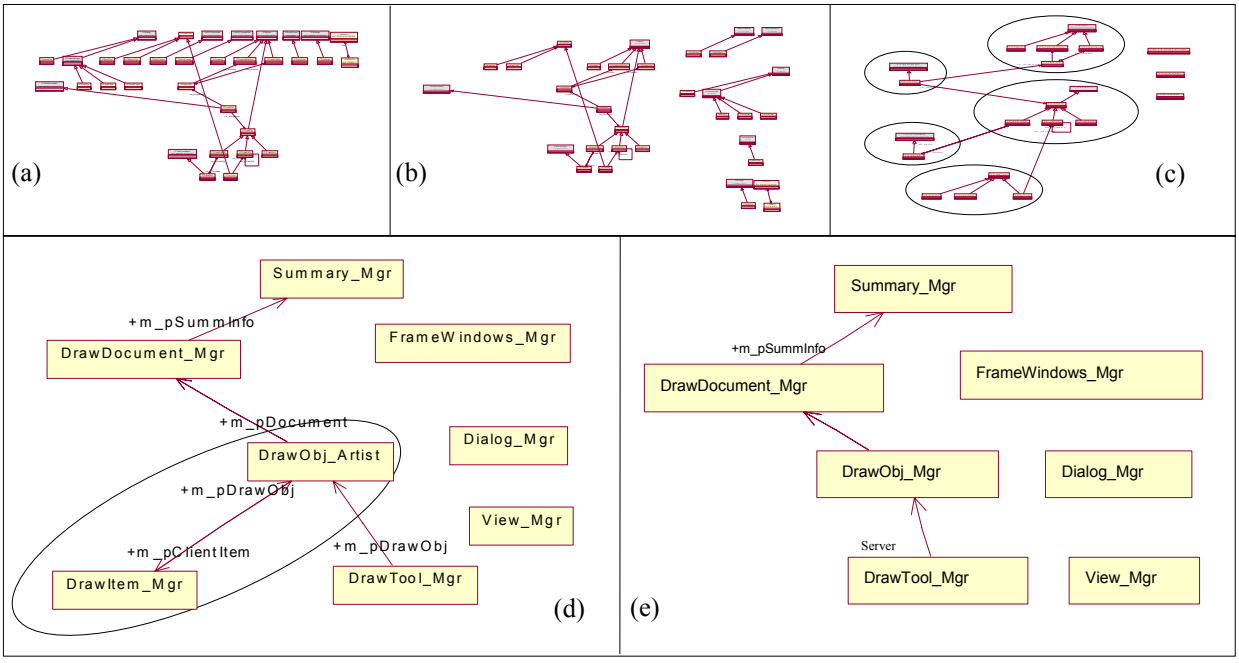

Figure 2. Identifying components from a class diagram. At this magnification, the top three diagrams are shown only for illustration, to convey the task's scope. Their complexity and considerations of the paper's length prevent us from enlarging and discussing them in more detail.

(2) *Group related classes.* The class diagram is further simplified by grouping related classes using ten rules, the first five of which are completely automatable. An engineer may choose to apply only a subset of these rules. However, the rules, or any subset thereof, are intended to be applied in the order in which they are presented. *(i)* Classes isolated from the rest of the diagram comprise one grouping (Figure 2b). Classes that are related by *(ii)* generalization (i.e., inheritance) comprise additional groupings, as do classes related by *(iii)* aggregation and *(iv)* composition (Figure 2c). Note that it is possible for multiple application-level classes to inherit from a single base class that belongs to the implementation substrate (e.g., MFC). In that case, the base class is replicated to facilitate mutually exclusive groupings. Finally, *(v)* classes with two-way associations are grouped together since they denote tight coupling (Figure 2d). Note that, at the end of this process, the only relationship spanning class clusters is association. The relations between classes belonging to different groups are still preserved and displayed. However, this is now done at the level of relations between entire groups, so that, for example, multiple associations are displayed as one.

The application of the above rules to DrawCli's implementation resulted in clusters of classes that could be abstracted into seven components (Figure 2e). However, these five rules alone are not always sufficient to determine components, especially in more complex systems. For this reason, we introduce additional rules. *(vi)* Classes that have large numbers of incoming and outgoing relationships with other classes are recognized as *domain* classes [30]: their high level of interdependency with other classes implies a significant role in the system; this role is derived from the nature of the problem domain. Determining which classes are domain classes is inherently subjective and requires some level of human judgment. Based on our experience with Focus, we have found the following heuristic to be very effective: domain classes are likely those classes whose number of dependencies with other classes in the system is 2-3 times greater than average; after the application of the first five clustering rules, most classes have 1-2 dependencies on other classes; therefore, classes with 4 or more dependencies are candidate domain classes.

The remaining rules are based on domain classes and further help in reducing the complexity of a system's class diagram. *(vii)* If class *C* has a single dependency, and that dependency originates from a domain class, then *C* may be abstracted into the domain class. *(viii)* Every class belonging to a circular dependency path with a domain class along the path can be abstracted into that domain class (Figure 3a). [2] In addition, *(ix)* if there is a path whose start node and end node are both referenced by the same domain class, then all the classes from that path can be abstracted into a new cluster of classes (Figure 3b). We call this rule *open circular dependency*. Finally, *(x)* if there are two different paths with the same end node, and whose start

[2] A path is defined as a set of uni-directionally connected classes, so that by following the direction of the dependency, the last class in the path can be reached starting from the first class. A minimal path contains two classes and a dependency between them.

nodes are referenced by the same domain class, then all the nodes from the two paths can be abstracted into a new cluster of classes (Figure 3c). We call this rule *transitive dependency*.

(3) *Package classes into components*. Clusters of classes identified in the previous stage are packaged into abstract components. These components can be further grouped, using the above rules, to form even larger components.

The above discussion concentrates on the first iteration of the task of identifying components. During subsequent iterations, a subset of these components will be refined to satisfy the given evolution requirement. The information captured during this stage is used to ensure that the refinement of the architecture is consistent with the implementation throughout the evolution process.

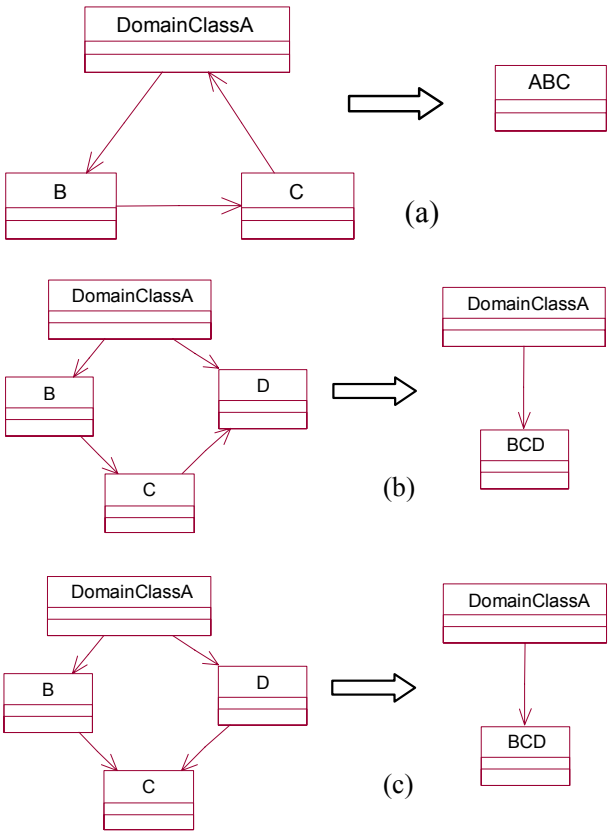

Figure 3. Example of clustering based on domain classes

Identify Data Components – The next step is to identify the data components that the processing components packaged in above manner will exchange. The obvious choice for the data components are the parameters of the processing components' public methods. However, since each processing component may comprise many implementation classes, the corresponding numbers of methods and their parameters may be very large. In the cases of components with clearly identified dependencies on other components (e.g., *DrawObj_Mgr* interacts with *DrawTool_Mgr* and *DrawDocument_Mgr* in Figure 2e), Focus considers only the data exchanged among them. However, this will clearly not suffice in the cases of components that have no identified dependencies on other components (e.g., *FrameWindows_Mgr*). Three possibilities suggest themselves:

- Only include data from components that have identified dependencies. This option is easiest, but may elide important information.

- Include the parameters from all of a given component's public methods. This option is most complete, but may obscure architecturally relevant data with a lot of data relevant only at the implementation level.
- Include the parameters from only those public methods that are not already accessed by the component's internal classes. This option is potentially most accurate, but may omit some data that are architecturally relevant (e.g., data accessed both internally and externally).

None of these three possibilities can be considered universally applicable, so Focus leaves the decision to the engineers' discretion. Recall from the above discussion that a fourth option, adopted by a number of architectural approaches, is to ignore the data components altogether. While our discussion in the remainder of the paper does not extensively center on data components, they play an important role in Focus (e.g., as evidenced by Figures 5 and 10).

B *– Propose Idealized Architectural Model* (independent activity)

During the initial phase of architectural recovery, the selection of an appropriate architectural style is critical. Since Focus assumes that the style used during the original development of the application is not known, the style can be selected based on several criteria, including the experience of the engineers involved in the recovery process, the characteristics of the implementation (e.g., a distributed application may imply a network-based style [10]) the characteristics of the application domain (e.g., a GUI-based application may imply the MVC [18] or C2 styles [39]), and, if available, the reference architecture for applications in the given domain (e.g., the reference architecture for Web servers uses the pipe and filter style [15]). Based on that style, the engineer's understanding of the application's architectural properties (e.g., program logic, development platform) and user-level properties is captured in an architectural model. The components identified in the idealized architecture need not match those extracted in activity *A* above (recall from Figure 1 that activities *A* and *B* are indeed independent). Also, there is no burden on the engineers to produce a "correct" architecture, only an architecture that "makes sense" for the application. The subsequent steps of Focus will identify and rectify any problems with the idealized architecture.

Our use of styles and architectural models in this situation differs in at least three ways from their typical use. First, while styles may be used in the *initial* formulation of a system's architecture, only a few of the approaches we have come across (e.g., [14]) make use of them in the process of architectural recovery. Secondly, as foreshadowed above, Focus allows for the style selected during this activity to be different from the style used in the application's initial development. This significantly lessens the burden of guessing the "correct" style. Finally, the architectural model at which one arrives as part of this activity is *idealized*; it is

understood that parts of it may characterize the application incorrectly. The model only sets a target for future adjustment and evolution (e.g., see activity *C*).

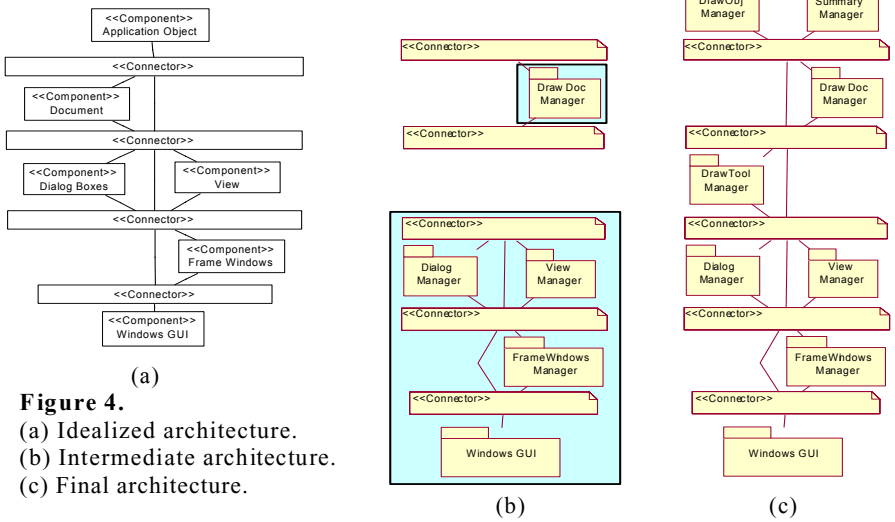

Figure 4.
(a) Idealized architecture.
(b) Intermediate architecture.
(c) Final architecture.

Figure 4a represents the initial idealized architectural model for the DrawCli system described in the C2 style [39]. C2 is a style intended for applications with a significant GUI aspect and has been successfully applied in the development of numerous applications to date. We briefly outline several of C2's key style rules. In C2, an application's computational elements (*components*) communicate by exchanging messages via explicit services in charge of component interactions (*connectors*). The *topology* of a C2-style architecture constrains the possible interactions in which components may engage. For example, the *Dialog Boxes* and *View* components in Figure 4a may not directly communicate with each other.

C *– Map Identified Components onto Idealized Architecture* (depends on activities A and B)

Mapping the components identified as part of activity *A* onto the idealized architecture of activity *B* will yield a set of intermediate architectural fragments (Figure 4b). These fragments only include those components (e.g., *Dialog Manager*) that clearly fit the idealized architecture. The determination of which components fit the idealized architecture can be made by the engineer, who is assumed to have participated in activities *A* and *B* and can thus perform simple component name matching (e.g., *Dialog_Mgr* in Figure 2e and *DialogBoxes* in Figure 4a) and, when necessary, name mismatch resolution. The engineer may be aided in this process by a simple tool that performs string comparison and suggests candidate matches. This task can be rendered more precise if data components accompany the processing components (e.g., as part of the processing components' interfaces).

This activity resulted in two such fragments of the DrawCli architecture, shown in the two shaded areas of Figure 4b. The remaining components identified in activity A (*Draw Obj Manager*, *Summary Manager*, and *Draw Tool Manager*) will be integrated into the complete architecture shown in Figure 4c in later steps.

D *– Identify Key Use Cases* (independent activity)

Focus expresses the major application requirements using UML use case diagrams. Use cases are derived by observing the user-level behavior of the application and from the statement of the desired application modification. When evolving an application, Focus identifies three categories of use cases: those unaffected by the change (e.g., *Print Diagram* in DrawCli); those corresponding to the new requirements (e.g., *Chat*); and those denoting existing functionality that must be modified (e.g., *Open File*). Identifying and prioritizing the latter two categories of use cases sets the *focus* for recovering additional architectural detail and evolving the application. We do not show the DrawCli use cases here for brevity; a prioritized view of them may be found at [37].

E *– Analyze Component Interactions* (depends on activities A and D)

To integrate all the identified components into the architecture, in addition to the components' static relationships (Figure 2), their interactions (i.e., control flow) must be analyzed as well. We use UML sequence diagrams to this end. We rely on two simplifying factors during this activity. First, we describe the control flow at the level of components (Figure 2e) rather than the much more numerous classes (Figure 2a). Secondly, we only generate the sequence diagrams corresponding to the key use cases identified in activity *D* above. During each iteration of Focus, the remaining use cases with the highest priority are selected. Figure 5 shows an example sequence diagram for drawing a rectangle.

We should note that Focus identifies processing and data components, but does not attempt to explicitly recover connectors. The reason for this is that connectors in systems implemented in OOPLs, at which we have specifically targeted Focus to date, are dispersed in the source code [21,36]. In essence, this activity identifies the (implicit) connectors in an application and allows their mapping to the (explicit) connectors suggested by the idealized architectural model (recall activity *B*). This issue is further discussed below, in the evolution step of Focus.

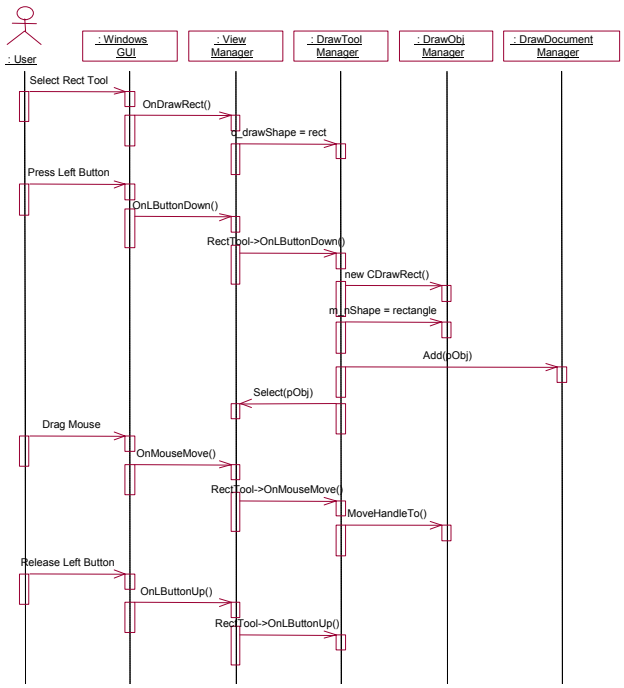

Figure 5. Sequence diagram for drawing a rectangle.

F *– Generate Refined Architecture* (depends on activities C and E)

Once the key component interactions are determined, we can proceed with completing the intermediate architecture created by activity *C*. We leverage the sequence diagrams to identify the dependencies between the remaining components (from Figure 2e) and those in the intermediate architecture (shaded areas of Figure 4b). We also leverage the sequence diagrams to identify any inconsistencies introduced into the architecture created during activity *B*. The inconsistencies that can be identified in this step fall into three categories:

- Components (processing or data) recovered from the implementation or specified in the idealized architecture, but not both.
- Components (processing or data) of different granularities in the two architectures.
- Interaction links recovered from the implementation or specified in the idealized architecture, but not both.

Note that this step will not eliminate all such inconsistencies, but only those relevant to the desired modifications of the application. The architecture resulting from this step is shown in Figure 4c.

2.2 System Evolution

Once the application architecture is recovered, we apply the system evolution step of Focus to modify the application in a manner that satisfies the new requirements. This step is also carried out in an incremental way, such that the modifications deemed most important are carried out first. As shown in Figure 6, system evolution is composed of the following five major activities.

A *– Propose Idealized Architecture Evolution* (independent activity)

Before the detailed changes are carried out, a high-level architecture evolution plan is made. In the DrawCli example, we propose to evolve the original system architecture into a distributed client-server architecture, where the internal architectures of both the clients and the server adhere to the principles of the C2 style. We also decide which components

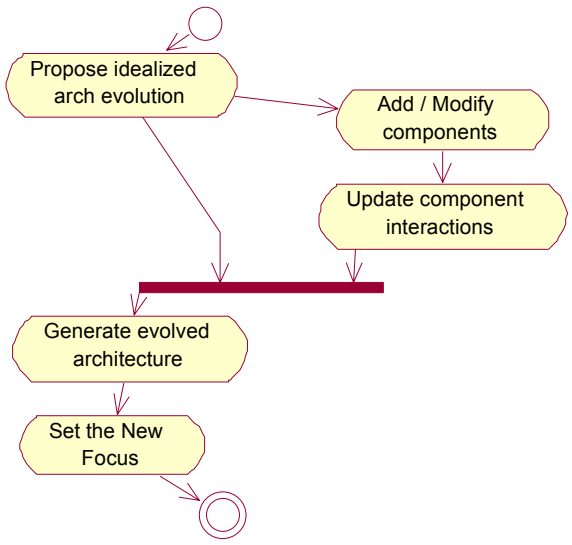

Figure 6. The system evolution step in Focus

from the newly-created clients and server must communicate across the client-server boundaries. In our case studies, we have found our reliance on explicit software connectors to be very useful in evolving an application. In this case, we exploit C2's connectors in order to distribute the application: any connector in the DrawCli architecture (Figure 4c) can be "sliced" into multiple horizontal or vertical sections; each connector section is placed in a different address space and ensures the proper communication flows across the entire connector [7]. Figure 7 shows two such connectors in DrawCli's evolved architecture, each of which is "sliced" and distributed across the three subsystems. The two connectors are produced by the two successive modifications to DrawCli (adding support for collaborative editing and adding a chat facility).

B *– Add/Modify Components* (depends on activity A)

Based on the evolution plan, the engineers must decide whether to add new or modify existing components, or both. For example, in the case of DrawCli, components enabling the chat facility and a server-side component are added to the system as shown in Figure 7; additionally, the interaction aspects of several existing components, highlighted in Figure 7, are modified. These modifications are discussed below.

C *– Update Component Interactions* (depends on activity B)

As components are added or modified, the control flow among them must be updated. In our example, when one user is drawing a rectangle, to make his actions visible to others, new messages should be generated and sent to the server. The server will then broadcast the messages to other clients. This interaction is modeled via a sequence diagram derived from the one shown in Figure 5. The modified sequence diagram is elided for brevity; it may be found at [37].

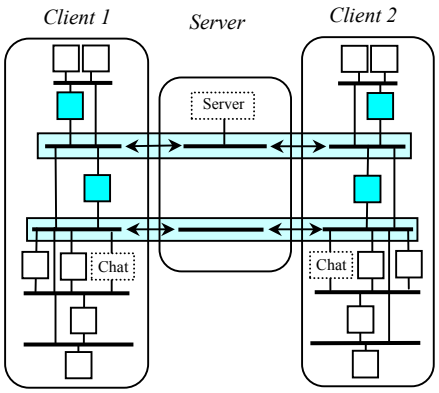

Figure 7. Evolved architecture of DrawCli. With the exception of the added *Chat* component, each client's architecture is identical to the architecture shown in Figure 4c.

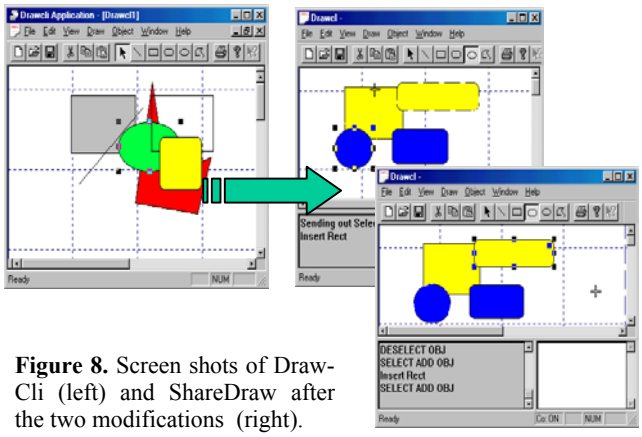

Figure 8. Screen shots of DrawCli (left) and ShareDraw after the two modifications (right).

D – *Generate Evolved Architecture* (depends on activities A and C)

All the above changes to the original system should be integrated into the original architecture (recall Figure 4c). The architecture thus generated will serve as the basis for the detailed design and implementation of the changes. The evolved architecture of DrawCli is shown in Figure 7.

One relevant implementation issue in the DrawCli example is updating component interactions to achieve the distributed drawing capability. This change to the application does not actually require individual components to be modified. Since the new communication among the components must be relayed across the client-server boundaries, we employ connectors to act as interaction intermediaries. DrawCli was originally not implemented using explicit connectors; instead, DrawCli's components (i.e., classes) communicate via method calls. Recall that the connectors depicted in Figures 3 and 6 are part of DrawCli's *idealized*, C2-style architecture. In order to evolve the actual application in the envisioned manner, we introduce explicit connectors in DrawCli's implementation. Specifically, we have introduced special-purpose OO classes into DrawCli's implementation, using the technique outlined in [7], to implement connectors based on Windows sockets; in turn, we have used those connector classes to achieve the topology suggested by Figure 7.

E – *Set the New Focus* (depends on activity D)

If the architecture generated in the preceding activity is still not detailed enough to enable the implementation of the required changes, a new iteration of the two major steps of Focus is required. The components affected by the changes (highlighted components in Figure 7) become the focus of attention during the subsequent iterations; the rest of the architecture will remain unaffected by the changes and is thus ignored in this process. Each subsequent iteration is intended to provide additional, lower-level detail of the impacted components (e.g., their internal structure as indicated in Figure 2) and connectors (e.g., their distribution as indicated in Figure 7). Figure 8 shows the screen shots of both DrawCli and the evolved system, *ShareDraw*.

In summary, the system evolution step of Focus is based on the recovered architecture; at the same time, its results are fed back to (the next iteration of) the architecture recovery step. Therefore, the recovery of the original system's architecture and the evolution of that architecture are achieved in an incremental, *focused* manner.

3 CASE STUDIES

In this section, we discuss three additional case studies involving architectural recovery and evolution using the Focus method. Information on several other case studies, including those performed independently, may be found at [37].

3.1 Jigsaw

Jigsaw is a Java-based Web server released by the W3C. It is an open-source project designed as an experimental platform. In this case study, the intent was to extend the Jigsaw server in such a way that, if a user types an incorrect URL into a Web browser's address field, the server will try to interpret the greatest possible portion of the URL string and return the HTML page with the contents of the most nested subdirectory whose name appears correctly in the given URL. For example, rather than returning an error message that the page cannot be found at `http://www.usc.edu:8080/work/reearch` (note that "reearch" has been misspelled), with the incorporated change the server would return the page containing all the available links inside the "work" directory on the server. This way, the user can more easily and meaningfully navigate to the desired document.

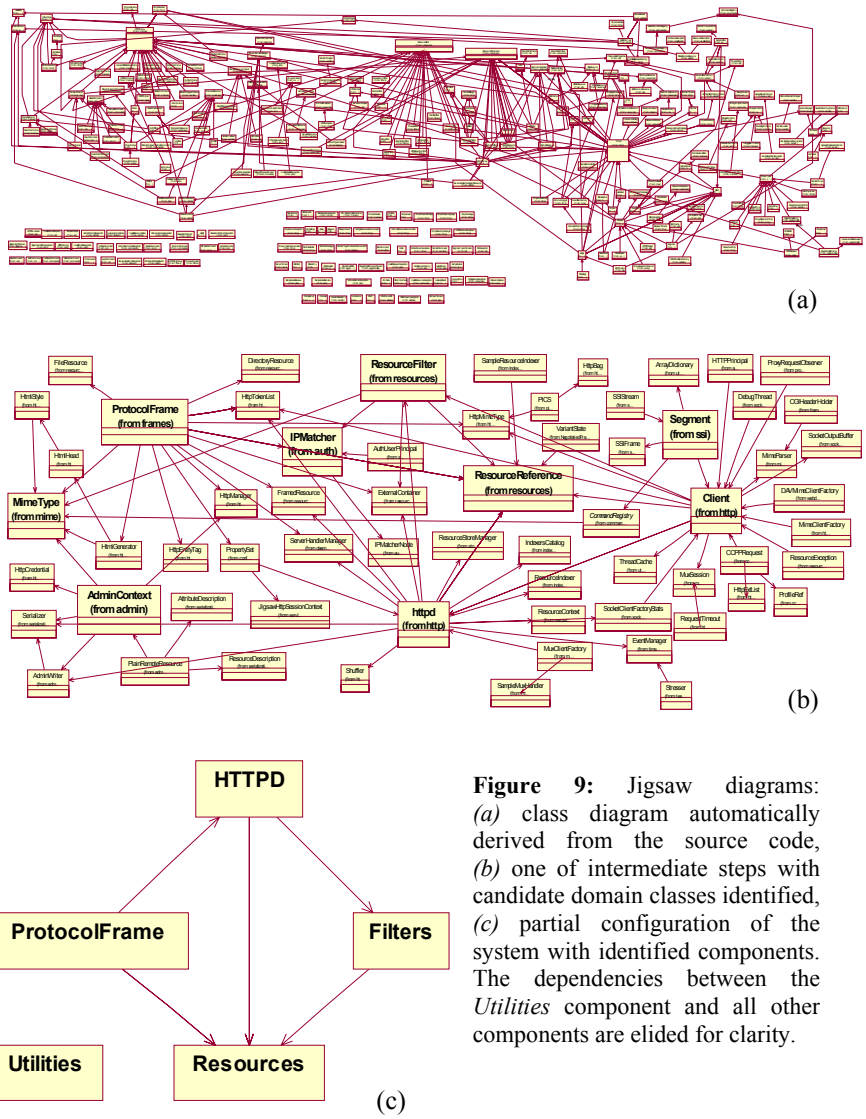

Figure 9: Jigsaw diagrams: *(a)* class diagram automatically derived from the source code, *(b)* one of intermediate steps with candidate domain classes identified, *(c)* partial configuration of the system with identified components. The dependencies between the *Utilities* component and all other components are elided for clarity.

The Jigsaw class diagram consisted of around 300 classes generated automatically by Rational Rose (Figure 9a). We assume that every system of this size has some common functions and data structures that are frequently used by the rest of the system and could be therefore grouped together, defining the *Utilities* component. This assumption goes in hand with similar hypotheses from other recovery approaches. For example, the *support library* pattern [42] describes systems with a number of procedures that are accessed by the majority of subsystems. In our case, the classes that belong to the *Utilities* component are those that do

not participate in any association (i.e., structural) relationships, but are only members of dependency (i.e., uses) relationships. Since it is performed automatically, this kind of classification might not be correct for all identified classes. Furthermore, there might be some other classes with existing structural relationships that would best fit within the *Utilities* component. If some correction of this kind is indeed required, it will be easily detected during Focus's evolution step.

To arrive at the idealized architectural model (recall activity *B* from Section 2.1), we used the reference architecture for Web servers, described in [15]. A Web server has a process that runs continuously on the machine, receives requests from Web browsers, spawns threads, and sends replies back to clients. It also has a module whose role is to distinguish between requests for static pages that can be served directly (e.g., HTML or image files), and requests that must be processed by an application server. The latter result in dynamically generated results of programs executed on the server's computer (e.g., a request to execute a Perl/CGI script that accesses the mySQL database).

Figure 9b depicts a diagram obtained after applying the *domain class identification* rule (rule *vi*) of the component identification step (activity *A* from Section 2.1). In total, there were nine classes with at least four incoming and outgoing dependencies and they were consequently recognized as candidate domain classes: *ResourceReference, httpd, ProtocolFrame, ResourceFilter, MimeType, AdminContext, IPMatcher, Segment,* and *Client*. As discussed in Section 2.1, this activity is independent of any other activity in Focus, meaning that the identified candidate domain classes are independent of the reference model chosen in activity *B*. By applying Focus's remaining component identification rules, these nine candidates were abstracted into five domain classes. For example, according to the *circular path* rule applied to the path containing the *Client, httpd,* and *SocketClientFactoryStats* classes (Figure 9b), the *Client* class was abstracted into the *httpd* class. Note that this way of abstraction was chosen arbitrarily: abstracting the *httpd* component into the *Client* would have made no difference in the subsequent steps of the process and would have provided an architecturally equivalent final result.

The application of the Focus recovery step resulted in a representation shown in Figure 9c. Note that, in accordance with the idealized architectural model, the dependencies between components represent the pipe connectors in the pipe and filter style. It should also be noted that this representation is not the only possible architecture of the system. One reason for this lies in the fact that Focus's architectural recovery is a semi-automated process and, therefore, prone to different engineers' decisions.

An analysis of the evolution requirement and the recovered architecture revealed that among the components identified in Figure 9c, there are two where this requirement can be met: *HTTPD*, as the front-end server

component from which replies are propagated to the client and internal representations of requests are generated [15], and *Resources*, as the component that is expected to provide all necessary logic for reply generation. At this point we needed a more detailed understanding of the request-reply use case scenario. Therefore, we focused on the flow of events between objects inside these components using a sequence diagram. The sequence diagram in Figure 10 only shows interactions of objects of interest while a request is being processed. The client's request is received by the *HTTPD* component; it is propagated through the *SocketClient* to *httpd*, and then sent to the *Resources* component for processing. If the *HTTPD* receives the "NOT FOUND" reply, another request is generated inside the *SocketClient*'s *perform* method, discarding the last occurrence of the slash ("/") character in the URL and the part of the URL to the right of it. The process is repeated until the non-null result from the server is received or the directory root of the server is reached.

On the other hand, request processing inside the *Resources* component boils down to the communication between four objects: *DirectoryResource, ContainerResource, FramedResource,* and *LookupResult*. *ContainerResource* handles URL lookup step by step: the resource is either located, or the new pathname is passed on for further lookup. When the non-existing resource name is encountered, the *target* field of the *LookupResult* object is protected from being set to null: the new target becomes the reference to the current directory. As a result, an HTML page listing the contents of the current directory is generated and sent to the client.

Both proposed solutions were implemented and evaluated. In both cases the resulting diagram remained the same since the diagram depicts only the structural organization of the computational components. However, notice that, for example, the sequence diagram shown in Figure 10 does in fact include the involved data components (the parameters of the displayed interactions). Because of its simpler use-case scenario that did not require communication (i.e., the exchange of data components) across computational components, we believe the second approach to be preferable. In it, the changes introduced due to the evolution requirement did not affect any of existing component interactions (activity *C* in the evolution part of

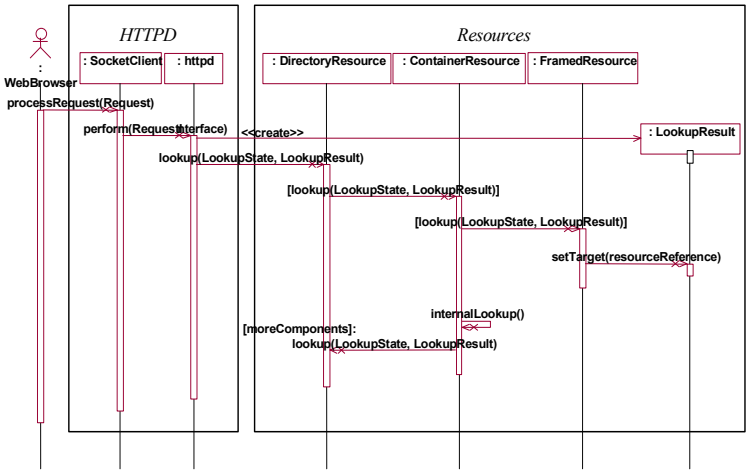

Figure 10: Sequence diagram for the request processing use case.

Focus in Section 2.2) because they were performed in a single component.

3.2 WordPad

WordPad is a standard word processing application included with various versions of *Windows*®. It is also provided in the *Visual C++* package as a sample application based on MFC. WordPad supports several "rich format editing" features, including the use of different text styles, fonts, colors, and point sizes; paragraph alignment, tabs, margins, and indentation; and the ability to read, write, and convert among Word 6.0, Rich Text Format (RTF), and ASCII text file formats.

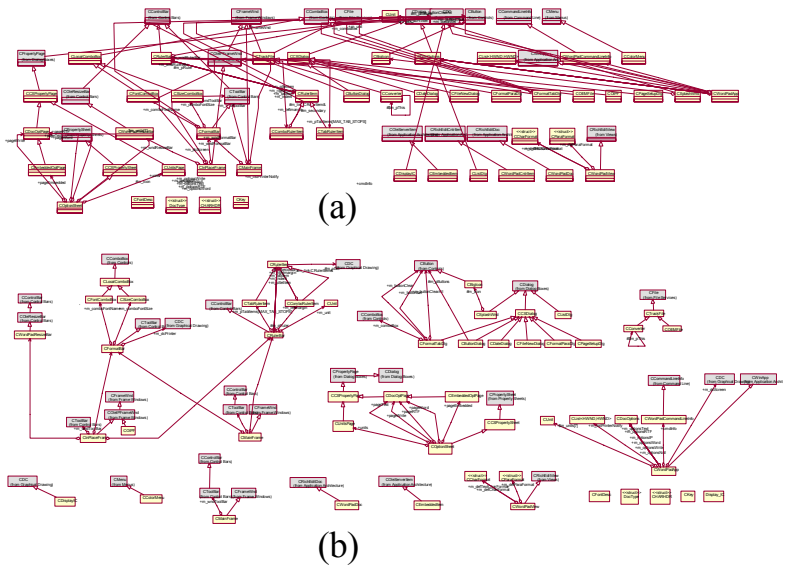

Figure 11. Class diagrams for WordPad: *(a)* automatically derived from the source code and *(b)* grouped in the process of using Focus into an initial set of components.

In this case study, the intent was to extend WordPad into a full-fledged collaborative authoring editor that supports fine-grain concurrent editing of shared documents by multiple, distributed users. By fine-grain, we mean that users can even edit the same word concurrently should they so choose.[3] Similarly to DrawCli, we also decided to integrate the *Chat* component into the evolved application, both as a useful utility in any computer-supported collaborative effort and as a point of comparison to our experience in applying Focus to DrawCli. Though shown in the evolved architecture of WordPad in Figure 12b (component labeled *C*), and in the screenshot of the resulting application in Figure 13, the details of this extension are not discussed here for brevity. However, we will further discuss this extension in Section 4, as part of our assessment of Focus.

WordPad's key functionality is based on three powerful MFC classes: *CRichEditView*, *CRichEditDoc*, and *CRichEditCntrItem*. Compared to a similar system developed from scratch, this rendered the actual size of the application-level code reasonably small: under 20,000 SLOC distributed over 61 source files. Still, with over 50 classes and their many dependencies, WordPad's class diagram is quite complicated (see Figure

[3] One may argue against the utility of such fine-grain control over collaborative document editing. However, determining the proper mechanism(s) for accomplishing this task is clearly outside the scope of this work.

11a), and does not provide many hints for modifying the system to satisfy the new requirements.

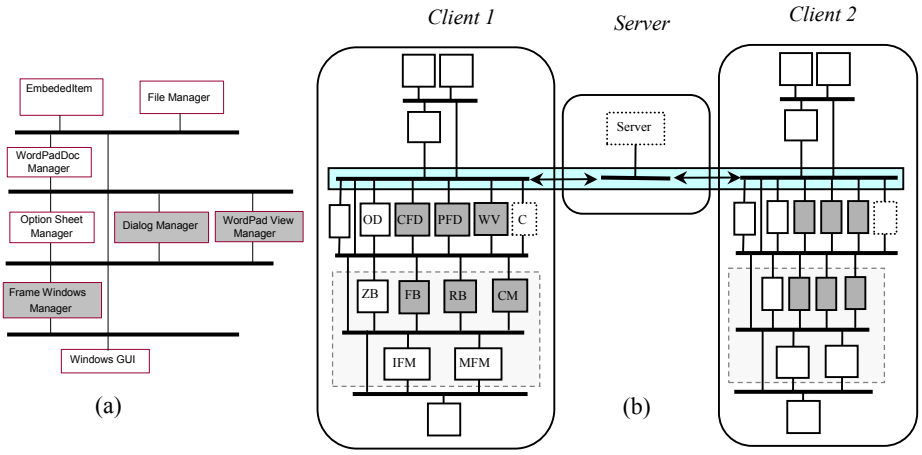

Figure 12. *(a)* Recovered and *(b)* evolved architectures of WordPad.

In applying the Focus approach, the WordPad classes were initially *grouped* into 16 low-level components (shown in Figure 11b). They are then further grouped based on the criteria discussed in Section 2 and, eventually, *packaged* into seven higher-level components. Those components were mapped onto our idealized C2-style architecture for WordPad with the help of several key use cases and their corresponding sequence diagrams. The outcome of this process—the refined WordPad architecture—is shown in Figure 12a.

An analysis of the evolution requirements and the recovered architecture revealed that only the three components highlighted in Figure 12a (*Dialog Manager*, *WordPad View Manager*, and *Frame Windows Manager*) would be affected by the desired change. Additionally, the analysis suggested that, similarly to the DrawCli example, the resulting application be built in the client-server style. For that reason, the *Server* component was introduced into the new system. Again, explicit connectors were exploited to easily facilitate distribution. Figure 12b shows the resulting architecture of the new system after two iterations of the Focus approach.

The *WordPad View Manager* component from Figure 12a is altered to depict other clients' modifications to a document, and is shown as the shaded *WV* component in Figure 12b. The *Dialog Manager* component of Figure 12a is refined into three more detailed components in Figure 12b: *Character Format Dialog* (*CFD*), *Paragraph Format Dialog* (*PFD*) and *Other Dialogs* (*OD*). This refinement is suggested by the class

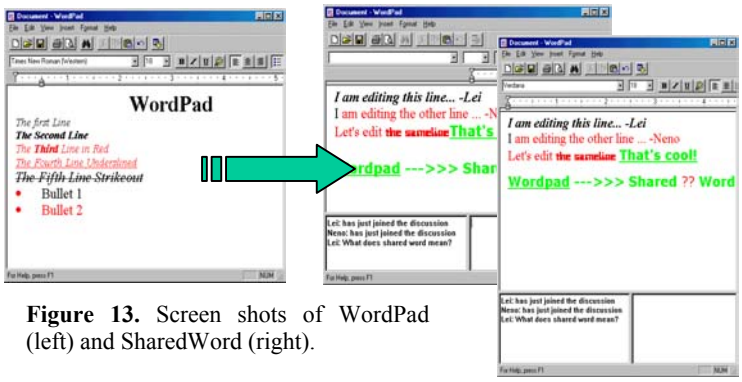

Figure 13. Screen shots of WordPad (left) and SharedWord (right).

grouping performed in the architecture recovery step of Focus. Components *CFD* and *PFD* are modified (and thus highlighted in the diagram) to inform other clients of local character and paragraph formatting changes to the document, while *OD* only affects local display and remains unchanged.

Similarly, in each client's subarchitecture in Figure 12b, the shaded area encompassing six components represents the detailed internal architecture of the *Frame Windows Manager* component from Figure 12a: *Resize Bar* (*ZB*), *Format Bar* (*FB*), *Ruler Bar* (*RB*), *Color Menu* (*CM*), *In-Place Frame Manager* (*IFM*) and *Main Frame Manager* (*MFM*). Among them, components *ZB*, *IFM*, and *MFM* remain unchanged.

With the help of the evolved architecture shown in Figure 12b, the implementation efforts are focused on the highlighted components and connectors, and leave other parts of the system unaffected. Figure 13 shows the screen shots of both the original WordPad and the new collaborative authoring editor, *SharedWord*.

3.3 Petri Net Simulator

The two case studies discussed above were based on MFC and, in that sense, it could be argued that they comprise a product family. In order to demonstrate that Focus is more broadly applicable, we conducted a third case study that dealt with a different implementation platform. In this case study, we modified an existing *Petri net simulation tool*, such that places in the Petri net are depicted by polygons whose number of sides equals the number of tokens inside each place (see Figure 14). Since places with zero, one, or two tokens cannot be represented by polygons, for the purpose of this exercise they are depicted by an empty circle, a filled circle, and a line, respectively. Every time a transition is fired, the shapes of all the places connected to that transition potentially change.

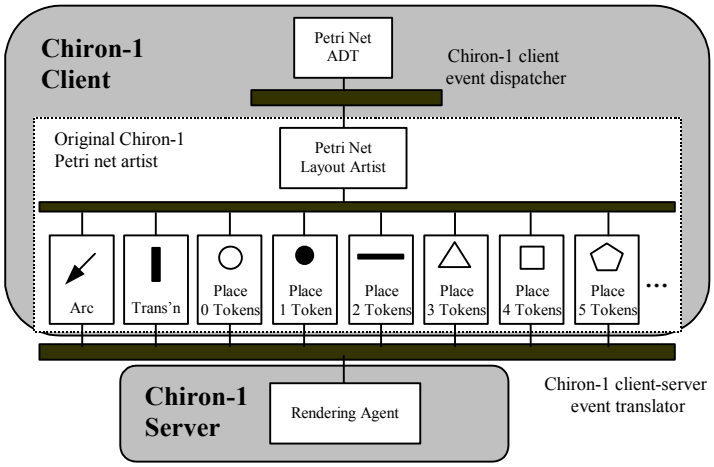

Figure 14. Recovery and evolution of the Petri net tool architecture.

The original Petri net tool was built in Ada using the Chiron-1 GUI management system as its implementation substrate [40]. Analysis of the tool's architecture revealed that it adhered to Chiron-1's strict client-server separation, where the *client*, implemented in Ada, provided the application's logic and GUI, while the *server*, implemented in C++, was in charge of rendering the application on the screen. Furthermore, the client consisted of two very large components, each of which addressed multiple application needs: an *ADT*, which

maintained the state of a Petri net and provided the logic for its execution, and an *artist*, which was tasked with maintaining all aspects of the Petri net's depiction. The ADT and artist communicated via events through Chiron-1's *event dispatcher*.

At this level of abstraction, the architecture of the Petri net tool was easier to recover than was the case with DrawCli or WordPad: the architecture is relatively monolithic and consists of only three coarse-grained components and two event buses (see Figure 14), so that a corresponding class diagram derived by Focus is very simple. However, this presented an added challenge during the evolution step of Focus since any modifications were likely to affect a large portion of the original application.

In order to achieve the specific modification described above and depicted in Figure 14, we redesigned the original artist by separating the layout of a Petri net from its presentation. In addition, the presentation of places with different numbers of tokens is entrusted to separate components (i.e., Ada packages). We reused the Chiron-1 event dispatcher to implement the connector between the Petri net layout and presentation layer components. The *Petri Net Layout Artist* shown in Figure 14 maintains the coordinates of places, transitions, and arcs, addresses issues of adjacency, and maintains logical associations with *Petri Net ADT* objects. At the same time, it has no knowledge of the artists in the presentation layer or the actual look of the Petri net. When a place is added, deleted, or repositioned, or its number of tokens changes due to a transition firing, the *Layout Artist* broadcasts the appropriate event notifications via the connector below it and only the artist maintaining the presentation of places with the specified number of tokens responds to them.

In order to achieve this modification, only the highlighted portion of the original application was modified. Most all of the new *Layout Artist*'s functionality was reused from the original artist. On the other hand, we had to alter and extend the functionality used for depicting Petri net places in the original application for each presentation-layer artist.

4 EVALUATION

Focus has shown a lot of promise in our studies conducted to date. Although it does not preclude the use of sophisticated tools, Focus *requires* only minimal tool support. In turn, it simplifies the human-intensive activities. For example, components are identified from source code in the architectural recovery step using ten simple rules (isolation, generalization, aggregation, composition, two-way association, domain class identification, single incoming dependency, circular path, open circular dependency, and transitive dependency), rendering this task feasible even for very large systems (e.g., the Jigsaw architecture was recovered in 10 person-hours). The approach similarly restricts the number of considered component interactions (and gener-

ated sequence diagrams) by focusing on those relevant to the desired application modification and by prioritizing use cases.

As a result of our focused approach to architectural recovery and evolution, we have had to consider only a small fraction of an application for each desired modification. Focus abstracts away a system's low-level details and leverages architectural style constraints in order to "zero in" on the parts of the system that need to be changed. In particular, after the components affected by a change have been identified, the task is limited only to the implementation-level classes that are clustered inside (activity A in Section 2.1) and mapped to (activity C in Section 2.1) those components. For example, addressing the new requirement in Jigsaw that was discussed in Section 3.1 required working with only 4% of Jigsaw's total source code when the change was introduced inside the *Resources* component, or 5% of the source code when it was inside the *HTTPD*. Similarly, the conversion of DrawCli to ShareDraw required working with only 15% of DrawCli's source code; the subsequent addition of the chat facility required focusing on 10% of the code. Note that this does not mean that we had to modify 25% of DrawCli. Rather, the two changes were contained within this portion of the original application. Focus helped us to isolate the exact portion affected by the changes and, in the process, to identify, document, and, by explicitly applying architectural style rules, evolve its architecture.

Another benefit of Focus is incurred when applying multiple changes to an application. In particular, much of the architectural recovery task is conducted only during the initial modification and its results are reused thereafter. Thus, for example, the additions of the chat facility to DrawCli and WordPad were accomplished in a fraction of the time it took to evolve the two into collaborative applications. One reason is that all 11 activities of Focus discussed in Section 2 are carried out during the first modification. During a subsequent modification, it is unnecessary to repeat activities A, B, and C of the architectural recovery step; furthermore, activities D and E are restricted to identifying only the use cases and component interactions specific to the new modification. As a result, the first modification to DrawCli was completed in 35 person-hours, while the second required less than 10.

While our initial results are promising, a number of relevant issues remain unexplored and warrant further study. These are discussed in the remainder of this section.

Selecting Architectural Style(s) – As discussed above, the selection of the appropriate style(s) is necessarily subjective because of a lack of necessary information. If, as in the case of the Jigsaw example, a reference architecture exists and its style is known, then style selection is easy and repeatable. If this is not the case, our experience has been that selecting the style will still be a reasonably straightforward exercise, simply

because typically a small number of styles will fit the application, and the goal of Focus is not to guess the "correct" style, just one that is appropriate, i.e., "close enough." If the style is a particularly bad fit for the application, we expect that the problem will be caught in activity C of focus's Step 1 (discussed in Section 2.1). In order to verify this hypothesis, we intend to run case studies in which an inappropriate style is deliberately selected for an architecture, and study its effects.

In two of the case studies described in this paper, we have demonstrated the simultaneous use of two styles in an application (client-server and C2). As originally speculated in [39], the chosen styles have proven to be complementary in many ways. Furthermore, we have selected applications for which there exists a well-suited overall style (in our case, C2). Neither of these assumptions will always hold. We need to conduct additional studies before we can assess the ability of Focus to address both (1) additional, less similar styles co-existing in a single application, and (2) multiple candidate styles for an application, none of which is an ideal fit. Furthermore, we intend to study the role of styles in extracting dynamic information about a system [17,34]. For example, in the Jigsaw case study, the reference architecture, adopted from [15], followed the pipe-and-filter style, providing information about the data and control flow between the different components. On the other hand, a partial configuration, resulting from the activity *A* of the Focus architecture recovery step, provides only a static representation of the investigated system.

Identifying Connectors – One unique aspect of Focus is that it identifies component interactions from the system's implementation and supports their mapping to the software connectors [22] suggested by the idealized architectural model. The fact that there is no typically dedicated code for connectors in software system implementations complicates this process. However, a simplifying factor in this task may be the observation made in a recent "state-of-the-art" report on middleware technologies [9], which states that modern software systems usually rely only on a small number of recurring interaction (i.e., connector) facilities. In the interest of ensuring feasibility of the proposed task, one should focus on such a limited subset of connectors, extending it only when needed.

Automatability – Ideally, a software architecture recovery approach would be amenable to extensive, or even full, automation, whereby a collection of software tools would derive a faithful architectural representation from a system's implementation with minimal human intervention. However, our experience has indicated that this would be an unrealistic expectation in an approach that extensively leverages a number of critical, but not yet fully understood or standardized architectural notions, such as connectors, styles, rationale, and so on. While the source code manipulation (e.g., clustering) facilities may indeed be automated (see below), a software architect will have to rely extensively on her experiences, intuitions, stylistic heuristics, and so on, since, by its very nature, the implementation will obscure certain architecturally relevant decisions and

notions. Therefore, we believe that, rather than trying to fully automate architectural recovery, the goal should be to support the capture, communication, and traceability of major architectural decisions during the recovery process. Our on-going work on extending the Rational Rose environment is trying to address these very needs.

Limitations of Source Code Analysis Tools – Several of Focus's clustering rules require identification of binary relationships between classes (e.g., associations, aggregations, and compositions). In turn, this requires that source code recovery tools be able to distinguish one relationship from another and represent them in a proper way. However, the binary class relationships at the design level do not always have unique representations at the implementation level [12]. For example, UML aggregation and composition relationships cannot be captured in Java, and therefore the Rational Rose Java reverse engineering tool shows only associations in a class diagram. A couple of solutions to the problem of the discontinuity between modeling and implementation for binary class relationships are given in [12]. Available tools are also often unable to discover a relationship that is implemented indirectly, e.g., by using instances of container classes (e.g., Vector, Map, List) to store objects of some other application class. This is further complicated by introducing user-defined container classes.

Implementation Platform – Two of the four case studies discussed in this paper dealt with applications implemented using MFC, thus sharing certain architectural traits. Additionally, the developer performing these studies had some previous experience with MFC, further easing our architecture recovery efforts. In like manner, the engineers performing the Jigsaw case study also had previous experience with its programming environment (Java). The fourth case study (Petri net simulator), however, has a very different implementation substrate (Chiron-1). While the overall approach still proved effective, our initial lack of familiarity with Chiron-1 required us to study it extensively before we could proceed with rearchitecting the application. The difference between the two experiences indicates that the implementation platform and the engineers' familiarity with it impact the time and effort required to apply the Focus method successfully. The true extent of that impact is currently unclear.

Domain Knowledge – Familiarity with a family of applications from a given domain, such as a product line or reference architecture, might simplify the discovery of a system's components. Another aspect of domain knowledge is familiarity with the environment and programming languages in which the target system has been developed. For example, the fact that Jigsaw is a server written for the Java Virtual Machine eliminated the need to search for a subsystem that enables communication with the underlying OS. The next level of domain knowledge use may be the use of comments and naming conventions (e.g., grouping based on similar names) for directories, packages, files, classes, interfaces, methods, parameters, and variables. However,

compared to a great number of existing architecture recovery approaches that use naming conventions as one of their clustering rules (e.g., [4,13,17,26,27]), Focus does not rely on this information. While such rules can be integrated into the architectural recovery step of Focus, we decided to avoid them because it is unclear to what extent one can expect developers in large, long lived, distributed, possibly open source projects (e.g., Jigsaw) to strictly follow a common and complete set of coding rules.

Concurrent Application Modifications – Focus currently (implicitly) assumes that multiple modifications to an application will be carried out in succession (e.g., as was done in the DrawCli and WordPad case studies). However, it is unclear whether a sequential approach is the optimal, or even realistic, way of accomplishing such a task. For example, it is likely that, once the subsystems affected by a change are identified, mutually exclusive changes may be pursued in parallel and their results (the recovered architectures and the modified implementations) merged after the fact. Deciding on what constitutes mutually exclusive modifications, merging multiple architectures and implementations that result in the process, and understanding the potential challenges of doing so remain open problems. Another, more challenging class of open problems stems from simultaneous system modifications that may not be mutually exclusive, e.g., in the case of open-source systems. Understanding such modifications, which are likely performed at different sites, and merging them into a unified system version is a formidable research problem in its own right.

Scalability – Our examples to date have dealt with applications of at most 100,000 SLOC, not counting their implementation substrates or meta-level resource configuration subsystems that were of no interest to our study (e.g., the MFC classes used by WordPad, the server-side *Rendering Agent* in Chiron-1, or Jigsaw's administration tool). As a partial consequence of the sizes of the involved applications, we have been able to complete each case study in less than 75 person-hours. While the encountered sizes are representative of many existing applications, further experiments are needed to assess the ability of the Focus method to scale up to larger systems. The key property of Focus that suggests that it will indeed be able to scale up to large systems is the fact that it zeroes-in on the particular subsystems affected by the evolution requirements. However, this may still result in having to understand and evolve substantial portions of a system in the cases of very broad evolution requirements (e.g., "make the system Web based") and very poorly structured systems. In the former case, we suspect that the broad evolution requirement can be recast into a number of more specific requirements, and Focus applied on each of them. The latter case is further discussed below.

Poor Modularity – Numerous researchers, including a recent study by Bowman et al. [4], have demonstrated that developer discipline is difficult to ensure in a large system built and maintained by many people over a long time; in such cases, the physical architecture of the system eventually reaches a high degree of connectivity among its components. We have not encountered cases of such severe architectural erosion in our studies

to date and it is an open question to what extent Focus would be able to address them adequately: if we were to simply use Focus to recover such a system's entire architecture (or a substantial subsystem's architecture), that would likely undermine Focus's key claimed benefits of incrementality and light weight.

5 RELATED WORK

Many aspects of Focus have been inspired by other work on architectural recovery and software evolution. A common approach to architectural recovery is *software clustering*. Based on the composition operations that are used for grouping source code entities such as classes or variables, existing software clustering techniques can be divided into two major categories. *Syntactic clustering* employs approaches that focus on static relationships among entities. Any functional property or meaning of entities is not taken into consideration. Entities are grouped together based on the existence of tight relationships between them. For example, if the result of source code analysis is that one task spawns another task, these two can be grouped together to represent (a part of) a single system component. More examples of operations of this kind include: variable and class references, procedure calls, use of packages, association and inheritance relationships among classes, and so on. In addition, these approaches often embody some kind of inter- and/or intra-component connectivity measures (e.g., [20]).

Semantic clustering, on the other hand, includes all aspects of a system's domain knowledge and information about the behavioral similarity of its entities. This means that all implementation constructs with similar functionality can be grouped together to form a software component. For example, all classes that provide support for checking the users' authenticity can be grouped together and form a single software component. Although this approach might provide more meaningful components, its main drawback is that it is domain specific, so that its rules cannot be easily applied to an arbitrary system.

Tzerpos and Holt give a similar categorization in [42] and indicate that the structure-based (i.e., syntactic) techniques are predominant among existing software clustering approaches and that the ones based on naming conventions are the most promising. Note that Focus uses a structure-based clustering approach without the use of naming convention for component identification. Naming conventions can also be used as a basis for behavior-based clustering. For example, two entities with non-similar names but with similar functionalities (implied by their names), such as *Authenticate.java* and *Authorize.java* can be grouped together into a *Security* component. Although the clustering-based approaches to architectural recovery presented in the remainder of this section are based on syntactic rules, some of them utilize semantic rules as well (e.g., support for creating abstract data type subsystems in Rigi). The section concludes with several examples of work related to Focus's evolution part.

Rigi [43] is a program-understanding tool that provides support for discovery and hierarchical representation of subsystem abstractions. Subsystem composition, based on artifacts that are extracted and then stored in an underlying repository, depends on the purpose, audience, and domain [26]. For example, for program understanding purposes, the established approach is low coupling and strong cohesion; alternatively, components can be identified by maintenance personnel based on their experience or qualifications. This also means that, unlike Focus, the composition criterion depends on the application that is being redocumented, the use of domain knowledge is unavoidable, and the recovery is sometimes done by persons who are familiar with the application (e.g., its developers). However, Rigi's users can also employ selection and search algorithms on the basis of system aspects such as graph connectivity, and component and dependency type [43].

Rigi [25] also has a concept of *omnipresent* nodes, which is seemingly similar to the notion of Focus's domain classes. However, the two are defined and used in very different ways. Rigi's omnipresent nodes are defined as nodes accessed by the majority of subsystems and are, therefore, considered to obscure the system's structure and are usually disregarded. They also capture the notion of helper classes, which are endpoints of uses relationships (i.e., dependencies) and, as such, are typically not shown in class diagrams. On the other hand, Focus's domain classes are defined based only on the number of structural relationships (i.e., references to objects), and, contrary to omnipresent nodes, also capture fan-out information. Moreover, domain classes with only fan-in relationships might serve as communication objects for other domain classes, and therefore cannot be disregarded in the process of clustering.

Bunch [20] is a clustering tool that lets the user evaluate the quality of an application's modularization, by providing source code graph connectivity metrics. The graph's nodes represent program units or modules, such as files or classes; edges between the nodes represent calls or relationships between those modules, such as function calls or inheritance. Similarly to the notion of Focus's *Utilities* component (recall Section 3.1), Bunch isolates all library modules in a single subsystem. Unlike Focus, where nodes with high fan-out values are seen as separate candidate components, Bunch treats them as drivers and groups them together in a single subsystem. Bunch thereby addresses the potential problem of the system's structure being obfuscated by the drivers, but it may also run the risk of misallocating system modules into the drivers subsystem.

ManSART [14,44] is a Software Architecture Recovery Tool that supports architectural style-based recovery. It recovers software architecture view representations that consist of a component-connector graph and associations to source code fragments. Components and connectors represent different source code abstractions in different views. The collection of different views provides hierarchies and abstractions that partially recover the overall design of the system. Architectural representation is obtained by combining (e.g., merg-

ing) different views or by finding connected subsets of a view. Special query language routines, called recognizers, are used to extract and analyze style information from an abstract syntax tree representation of the source code. Similarly to Focus, architectural styles in ManSART are used to represent an idealized view of the system. However, unlike Focus, where architectural styles are used to describe the constraints of the system's architecture, in ManSART the use of styles is limited to recognition of what components and connectors one can expect to find in the target system.

Similarly to the use of idealized architectures in Focus, Alborz [33] implements the notion of the architectural pattern that is incrementally developed by the user and can be referred to as a system's conceptual architecture. Each pattern generation step depends on the domain knowledge, system documents, and the results of the previous graph matching procedure steps [34]. However, unlike Focus, where the idealized architecture is represented using a component-connector topology, the conceptual architecture in Alborz is a pattern-graph with nodes that represent entities such as files, functions, and variables, and edges that represent *call* and *use* relationships. Component clustering in Alborz is based on the degree to which the entities in one component are related to the entities in another component [33]. This measure is obtained by applying data mining techniques on the graph representation of the system, where the system entities are denoted as nodes and data/control dependencies as edges.

Riva [31] identifies four main data categories that are involved and should be identified in the reverse architecting process: feature, architecture, design, and code. Furthermore, he provides suggestions for a unified exchange format that should be developed in order to cover all the phases of the process. Creating the architectural abstractions is, unlike Focus, done by looking at the documentation or by interviewing the developers. Similarly to Focus, the reference architecture might be used for the purpose of identifying architecturally relevant concepts.

Similarly to Focus, ARM [13] is an approach to architectural reconstruction that considers a system's architecture from two perspectives: the conceptual architecture and the actual architecture derived from source code. ARM applies design patterns and pattern recognition to compare the two architectures. To this end, ARM employs Dali, an environment providing several powerful tools for model extraction, analysis, fusion, and visualization. While we would certainly benefit from tool support such as that provided by Dali, a goal of Focus is to require only relatively simple static analysis tools to extract class diagrams from implementations. Additionally, unlike Focus, ARM is used to recover architectures, but not necessarily to aid in their evolution. Finally, ARM relies on system designers and documentation to arrive at a conceptual architecture, while Focus is applicable in situations where no such information exists.

Murphy et al.'s software reflexion models [27] also treat a system's architecture from two perspectives: the idealized, high-level view and the low-level view derived from source code. Moreover, reflexion models allow an incremental approach to architectural recovery: an engineer may analyze whether a desired, but not necessarily complete set of relationships holds between the idealized and actual architectures; the engineer may then repeat the process using a different set of relationships to gain a better understanding of the system. The key difference between this approach and Focus is that Focus is evolution-driven and also provides mechanisms for implementing new requirements. Although reflexion models have been used to aid system reengineering efforts, that is not their primary intended use. Furthermore, unlike reflexion models, Focus makes extensive use of explicit architectural styles and software connectors.

Tzerpos and Holt's ACDC algorithm identifies several patterns, which relate to subsystem structures that can be recognized while decomposing software systems [42]. Similarly to the concept of domain class discovery in Focus, the *central dispatcher* pattern describes resources that depend on a large number of other resources (e.g., a driver). Unlike Focus, where domain classes might be used during the component identification process, the central dispatcher resources are initially disregarded in the ACDC algorithm and reconsidered only in conjunction with already formed components.

Recently, a series of studies has been undertaken by Holt et al. to recover the architectures of several large-scale, open-source applications (e.g., Linux, Apache) [4,15]. Their approach is similar to ours in that they also come up with a conceptual architecture and use it as the basis of understanding a system's implementation. However, unlike Focus, their approach makes the assumption that at least some documentation will exist for the system. Furthermore, since the main objective of their work has been system understanding, the reverse architecting process is done thoroughly, with equal effort devoted to each part of a system. In Focus, the recovery effort is centered on the subsystem(s) believed to be affected by the desired evolution. Finally, Holt et al. extract relationships among subsystems (components in our case) via static function call analysis, whereas in Focus this is also done via requirement-driven use-case analysis.

With respect to system evolution, COREM [11] is a systematic approach to converting procedural software into OO systems via architectural transformations. It consists of four main steps: design recovery, application modeling, object mapping, and source-code adaptation. Though the steps used by Focus and COREM are similar, their objectives differ: COREM's objective is to change an entire legacy system into the more maintainable OO form, rather than try to incrementally satisfy new requirements.

Finally, similarly to Focus, MORALE [32] is used to adapt existing software systems to conform to new requirements. However, unlike Focus, MORALE mainly facilitates the evolution of legacy software systems

developed with procedural languages. Furthermore, MORALE is not intended for incremental recovery and evolution.

6 CONCLUSION

This paper discussed Focus, an approach for recovering and evolving architectures of undocumented OO applications. The approach has proven light-weight and efficient, showing the promise of enabling rapid, yet dependable evolution of a large class of existing applications. The philosophy behind Focus is that architectural recovery is not an end in itself, but is useful only as a means to achieving improved system maintenance and evolution. Focus thus reduces the scope and complexity of the architecture recovery step by focusing the engineers' attention on the system's components critical to the overall task (system evolution). The architecture of any subsystem that has not been impacted by the changes is not "correct," but is rather (deliberately) idealized; the actual architecture of each such subsystem will be recovered only as future changes that impact the subsystem are required. While Focus has been applied to date on several moderately sized third-party applications, its ability to center on the relevant subsystems suggests its possible applicability in large-scale settings.

In addition to its light weight, incrementality, and intentional approximation of a system's actual architecture, another novel aspect of Focus is that it employs the results of software architecture research that have emerged over the past decade. In particular, unlike most other documented architecture recovery approaches (e.g., [16, 27,41,43]), our approach directly exploits architectural styles and software connectors. Focus, therefore, not only serves to stem architectural erosion, but by incrementally fitting an application's *physical* (i.e., as-implemented) architecture to its postulated *logical* architecture in the selected style(s), it may fundamentally *rearchitect* the application.

Much work still remains to be done before the full extent of Focus's applicability can be assessed. It is likely that the open issues discussed in Section 4 represent only a subset of the relevant problems. We intend to use those issues as a starting point of our future research on Focus.

ACKNOWLEDGEMENTS

This paper is a significant revision and extension of the work published in the 2001 Working IFIP/IEEE Conference on Software Architectures, held in Amsterdam, the Netherlands, in August 2001. The authors wish to thank Lei Ding for contributing to the Focus approach and conducting the three MFC-based case studies. The authors also wish to acknowledge the instructor and students of USC's Spring 2003 CSCI 578 graduate course for completing additional case studies using Focus. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory or the U.S. Government.

REFERENCES

1. ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), 2002. http://www-2.cs.cmu.edu/~garlan/woss02/
2. Araki K., 1999. ed. Proc. International Workshop on the Principles of Software Evolution. Fukuoka City, Japan.
3. Booch G., Jacobson I., and Rumbaugh J. 1998. *The Unified Modeling Language User Guide.* Addison-Wesley.
4. Bowman I. T., Holt R. C., and Brewster N. V. 1999. Linux as a Case Study: Its Extracted Software Architecture. In *ICSE'99*, Los Angeles.
5. Bredemyer D. and Malan R. The Role of the Architect in Software Development. http://www.bredemeyer.com/pdf%20files/role.pdf
6. Chan P. and Lee R. 1996. The Java Class Libraries: An Annotated Reference. Addison-Wesley.
7. Dashofy E. M., Medvidovic N., and Taylor R. N. 1999. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *ICSE'99*, Los Angeles.
8. Eixelsberger W., Ogris M., Gall H., and Bellay B. 1998. Software architecture recovery of a program family. In *ICSE'98*, Kyoto, Japan.
9. Emmerich W. 2000. Software Engineering and Middleware: A Roadmap. In *The Future of SoftwareEngineering*, Anthony Finkelstein (Ed.), pp. 119-129, ACM Press.
10. Fielding R. 2000. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UC Irvine.
11. Gall H., Klosch R., and Mittermeir R. 1995. Object-Oriented Re-Architecting. In *ESEC-5*, Berlin.
12. Guéhéneuc Y.G., Albin-Amiot H., Douence R., and Cointe P. 2002. Bridging the Gap between Modeling and Programming Languages. TR 02/09/INFO, École des Mines de Nantes, France.
13. Guo G.Y., Atlee J.M., and Kazman R. 1999. A Software Architecture Reconstruction Method. In *WICSA-1*, San Antonio.
14. Harris D. R., Yeh A. S., and Reubenstein H. B. 1996. Extracting Architectural Features from Source Code. *Automated Software Engineering*, vol. 3. Kluwer Academic Publishers, Boston, pp. 109-138.
15. Hassan A. E. and Holt R. C. 2000. A Reference Architecture for Web Servers. In *Working Conference on Reverse Engineering*, Brisbane, Australia.
16. Kazman R. and Carriere J. 1998. View Extraction and View Fusion in Architectural Understanding. In *5th International Conference on Software Reuse*, Canada.
17. Kazman R., O'Brien L., and Verhoef C. 2003. Architecture Reconstruction Guidelines, 2nd Edition, TR CMU/SEI-2002-TR-034.
18. Krasner G. E. and Pope S. T. 1988. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80, *JOOP* 1(3), pp. 26-49
19. Kruchten P.B. 1995. The 4+1 View Model of Architecture. *IEEE Software*.
20. Mancoridis S., Mitchell B. S., Chen Y., and Gansner E. R. 1999. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*.
21. Medvidovic N. and Taylor R. N. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93.
22. Mehta N. R., Medvidovic N., and Phadke S. 2000. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 178-187, Limerick, Ireland.
23. Microsoft. Microsoft Foundation Class Library. http://msdn.microsoft.com/library/devprods/vs6/visualc/vcedit/vcrefwhatsnewmfcvisualc6.0.htm
24. Mikic-Rakic M., Mehta N. R. and Medvidovic N. 2002. Architectural Style Requirements for Self-Healing Systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, South Carolina.
25. Müller H. A., Orgun M. A., Tilley S. R., and Uhl J. S. 1993. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181--204
26. Müller H. A., Wong K., and Tilley S. R. 1994. Understanding Software Systems Using Reverse Engineering Technology, In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*.
27. Murphy G. C., Notkin D., and Sullivan K. 2001. Software Reflexion Models: Bridging the Gap Between Design and Implementation. *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364-380.
28. Object Management Group. Common Object Request Broker Architecture. http://www.corba.org
29. Perry D. E. and Wolf A. L. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT SEN*.
30. Quatrani T. 1998. *Visual Modeling with Rational Rose and UML*, Addison-Wesley.
31. Riva C. 2000. Reverse Architecting: Suggestions for an Exchange Format. Workshop on Standard Exchange Format, *International Conference on Software Engineering (ICSE 2000),* Limerick, Ireland.
32. Rugaber S. 1999. A Tool Suite for Evolving Legacy Software. In *ICSM'99*, Oxford, England.
33. Sartipi K. and Kontogiannis K. 2001. Component Clustering Based on Maximal Association**.** *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, pp. 103-114
34. Sartipi K. and Kontogiannis K. 2003. Pattern-based Software Architecture Recovery**.** *Proceedings of the Second ASERC Workshop on Software Architecture*, Alberta, Canada.
35. Shaw M. and Clements P. 1997. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97)*, Washington, DC.
36. Shaw M. and Garlan D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall.

37. Software Architecture Recovery with *Focus*. http://sunset.usc.edu/~neno/Focus/
38. Sun Microsystems. Java. http://sun.java.com/
39. Taylor R. N., Medvidovic N., Anderson K. M., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L. 1996. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Trans. on Software Engineering*, vol. 22, no. 6, pp. 390-406
40. Taylor R. N., Nies K. A., Bolcer G. A., MacFarlane C. A., Anderson K. M., and Johnson G. F. 1995. Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support. *ACM Trans. on CHI*, vol. 2, no. 2, pp. 105-144.
41. Tzerpos V. and Holt R. C. 1996. A Hybrid Process for Recovering Software Architecture. In *CASCON'96*, Toronto.
42. Tzerpos V. and Holt R. C. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pp 258-267.
43. Wong K., Tilley S., Muller H. A., and Storey M. D. 1995. Structural Redocumentation: A Case Study. *IEEE Software*, Jan. 1995, pp. 46-54.
44. Yeh A. S., Harris D. R., and Chase M. P. 1997. Manipulating Recovered Architecture Views, *Proc. Intl'l Conf. Software Eng.,* pp. 184-194.