1. ABSTRACT
A documented, analyzed software architecture is a key ingredient in achieving quality in large software intensive systems. But the system that is implemented must conform to its architecture for the qualities of the design to carry over into the implementation. To ensure that systems conform to their architectures, and remain in conformance throughout their lifetimes, we need to reconstruct architecture from source artifacts. To do this properly, a wide variety of tools that provide both static and dynamic information are needed. Thus, we advocate a workbench approach to architecture reconstruction tools.

1.1 Keywords
Software architecture, software architecture reconstruction, reverse engineering

2. WHY RECONSTRUCT SOFTWARE ARCHITECTURES?
Evaluation of an architecture’s properties is critical to successful system development [1]. However, reasoning about a system’s intended architecture must be recognized as distinct from reasoning about its realized architecture. As design and eventually implementation of an architecture proceed, faithfulness to the principles of the intended architecture is not always easy to achieve. This is particularly true in cases where the intended architecture is not completely specified, documented or disseminated to all of the project members. In our experience, this is the rule and well specified, documented, disseminated and controlled architectures are the exception.

This problem is exacerbated during maintenance and evolutionary development, as architectural drift and erosion occur. However, if we wish to transfer our reasoning about the properties of a system’s intended architecture to the properties of the implemented system, we must understand to what degree the realized architecture conforms to the intended architecture.

Architectural conformance may only be measured if we have available two architectures to compare: the intended architecture and the architecture as it is realized in the implemented system. The former should be documented early in a system’s lifetime and maintained throughout. The latter, however, typically exists only in artifacts such as source code and makefiles and, occasionally, designs that are directly realized as code (through, for example, the use of an architecture description language). In addition, it is infrequent that an implementation language provides explicit mechanisms for the representation of architectural constructs. Therefore, facilities for the reconstruction of a software architecture from these artifacts is critical in measuring architectural conformance.

Beyond its importance for measuring architectural conformance, software architecture reconstruction also provides important leverage for the effective reuse of software assets. The ability to identify the architecture of an existing system that has successfully met its quality goals fosters reuse of the architecture in systems with similar goals; hence architectural reuse is the cornerstone practice of product line development.

In the remainder of this paper we will discuss several issues relevant to the successful reconstruction of software architectures. These issues include the need to expand our horizons beyond the use of purely static information during reconstruction, the need to support the reconstruction of software architectures based a paucity of information, such as non-compilable code, obsolete artifacts or the absence of architectural information, and the need to leverage existing commercial and research tools within a comprehensive reconstruction framework.

3. STATIC INFORMATION IS INSUFFICIENT
A significant quantity of information may be extracted from the static artifacts of software systems such as source code, makefiles, and design models, using techniques that include parsing and lexical analysis. Unfortunately, system models extracted using these techniques provide a minimum of information to describe the run-time nature of the system. The primary factor contributing to this deficiency is the widespread use of programming language features, operating system primitives and middleware functionality that allow the specification of many aspects of the system’s topology to be deferred until run-time. All but the simplest systems use some subset of:

- language features such as polymorphism and first-class functions (including approximations such as those provided by C and C++);
• operating system features such as proprietary socket-based communication and message passing; and
• middleware layers such as CORBA.
These mechanisms permit systems to be designed with low coupling and a high degree of flexibility. While these are laudable goals, they obscure the architecture reconstruction process.

In particular static extraction techniques can provide only limited insight into the run-time nature of systems constructed using such techniques, because many of the details that determine actual communication and control relationships simply do not exist until run-time, and hence can not be recognized until run-time. For example, relationships among communicating processes might be determined via an initialization file or even dynamically, based upon the availability of processing resources. However, most existing architecture reconstruction tools depend almost exclusively on abstract syntax trees (ASTs) extracted using parsing-based (i.e. compile time) approaches, and so they cannot gather such information.

To achieve a better understanding of the run-time nature of systems that leverage mechanisms such as those described above, we must consider how we may go about extracting dynamic information from a running system. Some techniques to accomplish this include profiling and user-defined instrumentation. Profiling is a technique that is traditionally used for system performance analysis. When profiling, one typically compiles a system with a special flag that instructs the compiler to instrument the code such that it records information pertaining to function invocation during execution. The system is then exercised and the recorded information is analyzed. We may use this technique to determine actual function invocations, augmenting our statically-extracted models with improved information concerning polymorphic functions and functions executed through function pointers (e.g. in C or C++)

In a similar fashion, user-defined instrumentation is a technique for adding special-purpose tracing functionality to a system to allow monitoring of its operation. For example, instrumentation can be added to application code responsible for interprocess communication to determine the system’s run-time communication topology. In addition, it is sometimes possible to instrument libraries or even the operating system. This allows the possibility of instrumentation of systems without modifying any application code and requires less application-specific knowledge.

4. NON-COMPILABLE CODE
Throughout our experience with software architecture reconstruction, we have frequently been faced with systems that cannot be compiled. Often we are provided with a complete body of application code, but we are missing the header files and/or some of the libraries that are needed to compile it. In other cases, the application code is written in a peculiar dialect of a standard language (e.g. Objective C or an obscure dialect of Fortran), or implemented on an uncommon hardware platform, and thus may only be compiled with specialized tools that are not available to us. Off-the-shelf parsers and analyzers simply do not apply in these cases. There are cases when this lack of information makes a tool-facilitated reconstruction of the architecture nearly impossible; for example, when dynamic binding of function calls or dynamic relationships between processes are used extensively in a system, we may need to compile and run the system to determine the true relationships between the components.

Fortunately, we can often reconstruct software architectures even under these difficult circumstances. To accommodate non-compilable code whenever possible, we can avoid complete reliance on parsing-based techniques and can resort to lexical analysis for information extraction. While this is not an ideal situation (you often don’t have as much information to work with in these circumstances, and lexical techniques are frequently unreliable [8]), it is reality.

5. RECONSTRUCTING ARCHITECTURE IN A VACUUM
Unfortunately, it is frequently the case that our efforts to reconstruct the software architectures of systems must contend with a complete lack of pre-existing architectural information. This often occurs when the system being analyzed is particularly old (and thus there are no longer any designers or developers that can relate architectural information) or when the system’s intended architecture was never documented. Furthermore, these are typically the situations in which we are most interested in recovering a system’s architecture. In particular, it is common for such systems to be involved in ongoing maintenance or even undergoing a more global re-engineering effort (e.g. modernization or porting).

Successful architecture reconstruction revolves around the acquisition of architectural constraints and patterns that capture the fundamental elements of the architecture. Regardless of the mode of development of a system, its evolutionary state, or its age, such constraints and patterns are always present. However, they are rarely (even when a truly architecture-based development process is followed) captured explicitly. Thus, the primary task of the reconstruction analyst is acquisition of this information by means other than investigation of documentation.

One mechanism for acquiring architectural patterns is exploration of the information extracted from the system artifacts. Such exploration will often uncover frequently-used idioms of control flow, data flow, interface usage or interaction among the architectural components of the system. An alternative to this type of ad hoc exploration is the application of an architecture analysis method, such as the Architecture Tradeoff Analysis Method (ATAM) [6], as an elicitation device. Although ATAM was developed as a method for analyzing the tradeoffs inherent in architectural design, it may also serve as a structured way to elicit architectural information. ATAM is scenario-based:
scenarios are used to capture uses of and changes to the system being analyzed. It is the “scenario mapping” step of the method that is useful for architectural elicitation; in this step, the scenarios are traced through the architecture. For uses of the system, the participating components and their communication patterns are identified. For changes to the system, the architecture-level impacts are identified. In this way, we have a structured exercise for exploration of the architecture of the system. If the system being analyzed is indeed undergoing some level of maintenance, there will always be one or more developers that has some knowledge of its operation that can participate in such an exercise.

6. AN ARCHITECTURE RECONSTRUCTION WORKBENCH

Based upon the above observations, in developing tool support for software architecture reconstruction, our realization was that no particular static set of tools (extractors, visualization tools, analysis tools, etc.) would suffice for every task. At the very least, we wanted to support many implementation languages, many target execution platforms, and many techniques for architecture analysis. Thus, we wanted a workbench: an open and lightweight environment that provides an infrastructure for the opportunistic integration of a wide variety of tools and techniques. The workbench must be open to allow easy integration of additional tools. The workbench must be lightweight to ensure that no unnecessary dependencies exist among already-integrated tools and data.

We realized the workbench concept in Dali, a support environment for software architecture reconstruction [5]. Dali’s current implementation uses an SQL repository for data storage and manipulation. SQL provides the medium for the primary activities of the Dali user: architectural pattern definition, view creation, view fusion, and pattern recognition.

View fusion is a technique for combining extracted information—views—from different sources (such as code artifacts, directory structures, naming conventions, execution information) to improve the quality of the overall model of the system [4]. Architectural patterns are the medium for a Dali user to express their understanding of a system’s architecture as structural and attribute-based relationships among its components. The patterns are most commonly expressed via SQL, but may, in principle, be expressed via any of the tools that Dali integrates.

Rigi [10], which provides flexible graph editing functionality and effective end-user programmability, supplies Dali’s user interaction and also acts as a vehicle for tool integration.

Dali, as represented in Figure 1, currently integrates several commercial and research tools for extraction of information (rigiparse for C, Imagix [2] for C and C++, SNiFF+ [9] for C++ and Fortran, LSME [8] for C++), visualization (dot for graph layout) and architectural analysis (IA PR [3] for architectural complexity analysis, RMTool [7] for automatic conformance measurement). The use of commercial tools has provided a great deal of leverage during the development of Dali: these tools are typically robust, well documented, and well tested.

Dali offers an iterative, interpretive, user-driven approach to architectural reconstruction. Our view is that no system has “an” architecture. It has many views of a complex body of interrelated information and the choice of which views to extract and reconstruct is driven by the user’s information needs. It is, however, interesting to consider the implications of the user-driven nature of Dali with respect to the issues raised above.

Consider the case of non-compilable code: because of the iterative nature of Dali, we can use the information gleaned from purely lexical extractors to guide the user’s investigations. For example, in this case the user could extract some portion of the architectural information—knowing that it was incomplete—and then reconstruct just part of the architecture. This would then lead to new questions being asked, new lexical extractors written, and the process repeats with increasing fidelity.

In a similar fashion, when we extract views from dynamic information, these views are incomplete, since dynamic information, like run-time testing, only reflects the parts of the system exercised during the execution. To gain a complete picture of the system, run-time views need to be fused with other (possibly static) views to improve the overall quality of the reconstruction.

Lastly, it is again Dali’s iterative and interactive nature that makes it appropriate for supporting a process of reconstructing architecture in the absence of significant architectural information. Lightweight view extraction and flexible visualization provide an environment that fosters opportunistic exploration of the available information. These features also provide the necessary support for the
scenario mapping task that is central to the application of the ATAM method in this context.

It is the combination of Dali’s openness and its interactivity that helps us overcome the inherent limitations of any single extraction and reconstruction technique. Dali does not offer a monolithic solution, but rather a set of tools, some of which are heuristic. As a result, we have been able to apply Dali to a wide range of systems, written in many different languages and dialects, written for different hardware platforms and operating systems, and created with vastly different levels of architectural expertise.

7. REFERENCES