Requirements for Integrating Software Architecture and Reengineering Models: CORUM II

Rick Kazman, Steven G. Woods, S. Jeromy Carrière
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

{kazman, sgw, sjc}@sei.cmu.edu

Abstract

This paper discusses the requirements and a generic framework for the integration of architectural and code-based reengineering tools. This framework is needed because there is a large number of stand-alone reengineering tools that operate at different levels of abstraction ranging from "code-level" to software architecture. For the purposes of reengineering a *complete* system however, these tools need to be able to share information so that not only can the code be updated or corrected, but also so the system's software architecture can be simultaneously rationalized or modernized. To this end, we have built upon the CORUM model of reengineering tool interoperation to include software architecture concepts and tools. This extended framework—called CORUM II—is organized around the metaphor of a "horseshoe", where the left-hand side of the horseshoe consists of fact extraction from an existing system, the right hand side consists of development activities, and the bridge between the sides consists of a set of transformations from the old to the new.

Keywords: Software Architecture, Source Model Extraction, Architectural Views

1: Introduction

In [26], the experiences of creating a data model for interoperability between several reengineering toolsets was described. The resulting model, the Common Object-based Reengineering Unified Model (CORUM), was proposed to members of the reengineering community, explicitly recognizing the fact that different tools had distinct strengths and weaknesses (for technical, commercial, and organizational reasons) and noting that it ought to be possible to exploit the strengths of a wide variety of tools in an opportunistic fashion in reengineering a system. CORUM was envisioned as a unification point of previous beginnings in the evolution of a broader model for interchanging reengineering information and eventually, for creating a larger family of interoperable reengineering tools. This paper explores the evolution of CORUM to include architectural reengineering tools.

Aside from the above technical motivation, why is such a model needed? Quite simply, it is needed because there have been few reengineering community-wide efforts at inter-tool information sharing standards. Such efforts are needed to prevent repetitive "wheel-creation" and to reduce the amount of effort required to assemble a set of reengineering tools to support some analytic or software improvement process. Single-institution efforts have often produced powerful systems which combine sub-tools in generic, extensible, cooperating paradigms—one need look no further than ASF+SDF [24] and its related Toolbus [23] architecture to see such an example. However, such efforts have yet to be widely disseminated. An agreement among even a small set of reengineering researchers and tool producers on a model of inter-tool communication will result in a set of tools that can be readily re-assembled, extended and leveraged to support the creation of powerful novel reengineering tools. For example, the original and evolving CORUM specification, which started out of a desire to interconnect a handful of software analysis and presentation tools, has extended its scope in less than six months to encompass more than fifteen interested researchers at seven institutions.

However, all of the tools included in the original CORUM schema development were aimed at "code-level" reengineering, typically concentrating on the creation and manipulation of Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). We realized that this was a significant shortcoming of the original model, because it excluded software architecture concepts and toolsets intended to support architectural views, such as Dali [13] and the Portable Bookshelf [4].

Architectural reengineering tools have many characteristics in common with the code-level tools (for example, they typically build upon a representation of source code based on parsers), but also some significant differences. In particular, the architecture-based tools typically extract and manipulate information at a more abstract level than an AST, CFG or DFG. For example, architecture tools usually use a function-based rather than AST-based representation of code. While both reengineering and architectural tools support the ability to iteratively build higher level (hierarchically structured) abstractions from lower level components through partially automated and expert-driven pattern matching processes, architectural tools focus more on issues regarding recognition of concepts such as architectural styles and design patterns [5,11] rather than of "program plans" [27,17].

The creation of architecture-level reengineering tools is motivated by the realization that it is crucially important to be able to extract, reason about, and evolve software architectures. A software architecture is the earliest realization of a software system's organization and thus embodies the most fundamental and hardest to change design decisions [2]. As such, it has a strong determining effect on:

- the system's realization of quality attributes, such as performance, modifiability, availability, security, etc.
- the work breakdown structure of the development; this will be determined by the breakdown of modules or subsystems within the architecture
- planning for a software *product line*, based upon a common architecture and a set of shared assets

For these and other reasons, it is important to be able to extract, reason about, analyze, and reengineer a system's software architecture in conjunction with its code. For example, a typical reengineering effort these days is to rewrite a large scale monolithic COBOL data processing application as a distributed client-server application in C++.

The goal of this paper is to describe how such a reengineering task might proceed, what tool support it requires, and how an unified model of the artifacts shared among such tools is necessary to effectively carry out the task. This goal will be realized through a presentation of CORUM II, a revision of the original CORUM framework extended to include concepts and tools from software architecture.

2: The Horseshoe

For the purposes of exposition we have created a visual metaphor of the integration of code-level and architectural reengineering views of the world. The process of moving and mapping between these views within the overall reengineering task is presented as a "horseshoe". The horseshoe is a framework on which we will hang the many analytical and transformational processes of both domains throughout this paper. The creation of both CORUM and this horseshoe model has been strongly influenced by earlier work in the construction of hierarchical views of software structure [17,18,27]. In particular, the CORUM II horseshoe can be seen as adding an abstract architectural layer to these previous hierarchies.

The horseshoe, shown in Figure 1, is divided into three related processes, operating across four levels of software representation. The first process (going up the left hand side of the horseshoe) is code and architecture recovery and conformance evaluation. Here, the architecture of an existing system is recovered from extracted source code artifacts. Included in this process is an architectural conformance step, in which the as-built architecture is compared with the as-designed architecture (if available) and deviations are noted. The discovered architecture can also be evaluated with respect to known architectural styles and analytic models to more clearly understand the architecture's fitness with respect to a variety of quality attributes, such as performance, reliability and security.

Given the existence of a desired new architecture based on specific new system requirements, the second process in the horseshoe is one of architectural *transformation*. In this step the as-built architecture is reengineered to become the desired new architecture. This new architecture will also be

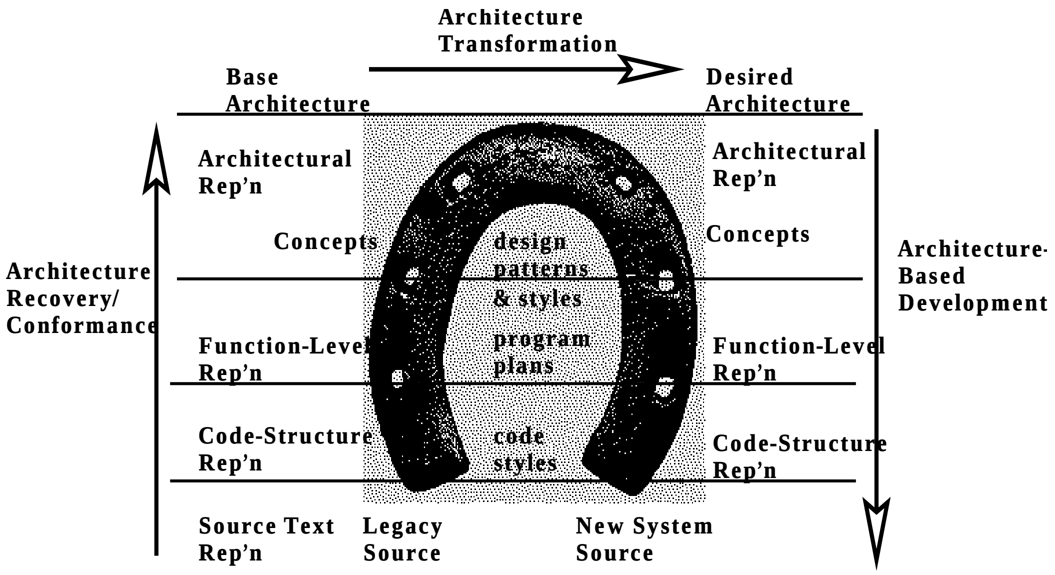

Figure 1: The Horseshoe

evaluated with respect to the system's quality goals. In addition, it will also be subjected to new requirements and constraints, such as buildability, time to market, and use of available development team skills.

The third process in the horseshoe is *architecture-based* development [2] in which the new architecture is instantiated. In this step the high level design detailed in the architectural transformation process is fleshed out: in particular, the architectural transformational process will be instantiated to entail many lower-level transformations [25] including rewrapping of code at the function level, adaptations or restructuring at the code-structure level, or simple string-based replacements at the textual level. In addition, in this step application program interfaces are set, module interconnections are determined, a work breakdown structure is planned, the allocation of code to processes and hardware is set, prototypes and simulations may be built (so as to better understand complex or risky areas), and finally code is written. Finally, extracted code-level artifacts from the original system might be re-incorporated to the system, either as-is or, more typically, partially wrapped or re-written so as to conform with the style of the new architecture.

For convenience we break the world of program understanding tools into categories according to the program, system, or design information they work *with* and the corresponding information that they produce. We break our knowledge schema into three distinct levels. The first, or base, is the code level which includes the source code and artifacts such as abstract syntax trees and flow-graphs obtained through parsing and rote analytic operations. The second is the function level which describes the relationship among a program's functions (calls for example), data (function and data relationships), and files (groupings of functions and data). Third is the concept level in which clusterings of both function and code level artifacts are assembled into patterns of architectural level components.

3: Reverse Engineering of Software Architectures

As already mentioned, the evaluation of a software architecture's properties is a critical form of risk mitigation in a complex system's development [2]. However, reasoning about a system's intended architecture must be recognized as distinct from reasoning about its realized architecture. As detailed design and implementation proceed, faithfulness to the principles of the as-designed architecture is not always easy to achieve. This is particularly true in cases where the intended architecture is not completely specified, documented or disseminated. The problem is exacerbated during maintenance and evolutionary development, as architectural drift and erosion occur. 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 as-built architecture *conforms* to the as-designed architecture.

To measure conformance, two preconditions must be met. There must be some documentation of the architecture available, or we must have access to the lead architect (for those all-too-common cases where the only documentation of the architecture lives in the head of an individual). And, we must be able to extract a *source model* of the system at the function level. In earlier reported extraction work [13], a typical schema for a source model consisted of elements such as *files*, *functions*, *classes*, *variables*, and *data structures*. It has relations such as *calls*, *contains*, *defines*, *has_subclass*, *writes*, and *reads*. The source model did *not* include constructs such as ASTs, CFGs, and DFGs such as described in the original CORUM specification [26].

Once these preconditions have been fulfilled, we are able to reconstruct the architecture from the extracted source model and then compare it, either manually or automatically, to the as-documented architecture. We use this process to document, and re-document, the software architectures of large systems and as an adjunct to architectural analysis [16].

The reconstruction of a software architecture involves several distinct activities: we have already mentioned source model extraction and pattern matching. The pattern matching process is facilitated by tools (e.g. [13,8]) that can aid a user in automatically or semi-automatically clustering source model elements according to user-specified patterns. Reconstruction of an architecture is an iterative, interactive process and so there must also be some means of visualizing, navigating, and manually manipulating the architecture. We do all of these activities to transform the *explicitly* extractable and viewable components of a source model into the *implicit* set of more abstract architectural constructs. In doing so, we make these implicit constructs explicit, by drawing specific mappings from the lower layers of the horseshoe to the upper layers, based upon the patterns that we have matched over the source model.

It should be noted—in fact, this is the main motivation for this paper—that identifying architectural patterns may rely on information from all layers. For example, identification of run-time socket-based interprocess communication topology in a client-server system will require data flow analysis to ascertain bindings of clients to servers. Shared memory usage may need to be tracked to understand the relationships between tightly coupled subsystems. Data type information may be required to understand how objects cluster into patterns. We have encountered many such instances in doing architectural recovery. In [12], for example, we describe an architectural recovery process that crucially depended upon being able to extract code level information, specifically: file accesses, and interprocess communication through the use of Unix mailboxes.

Extracting software architectures, redocumenting them,

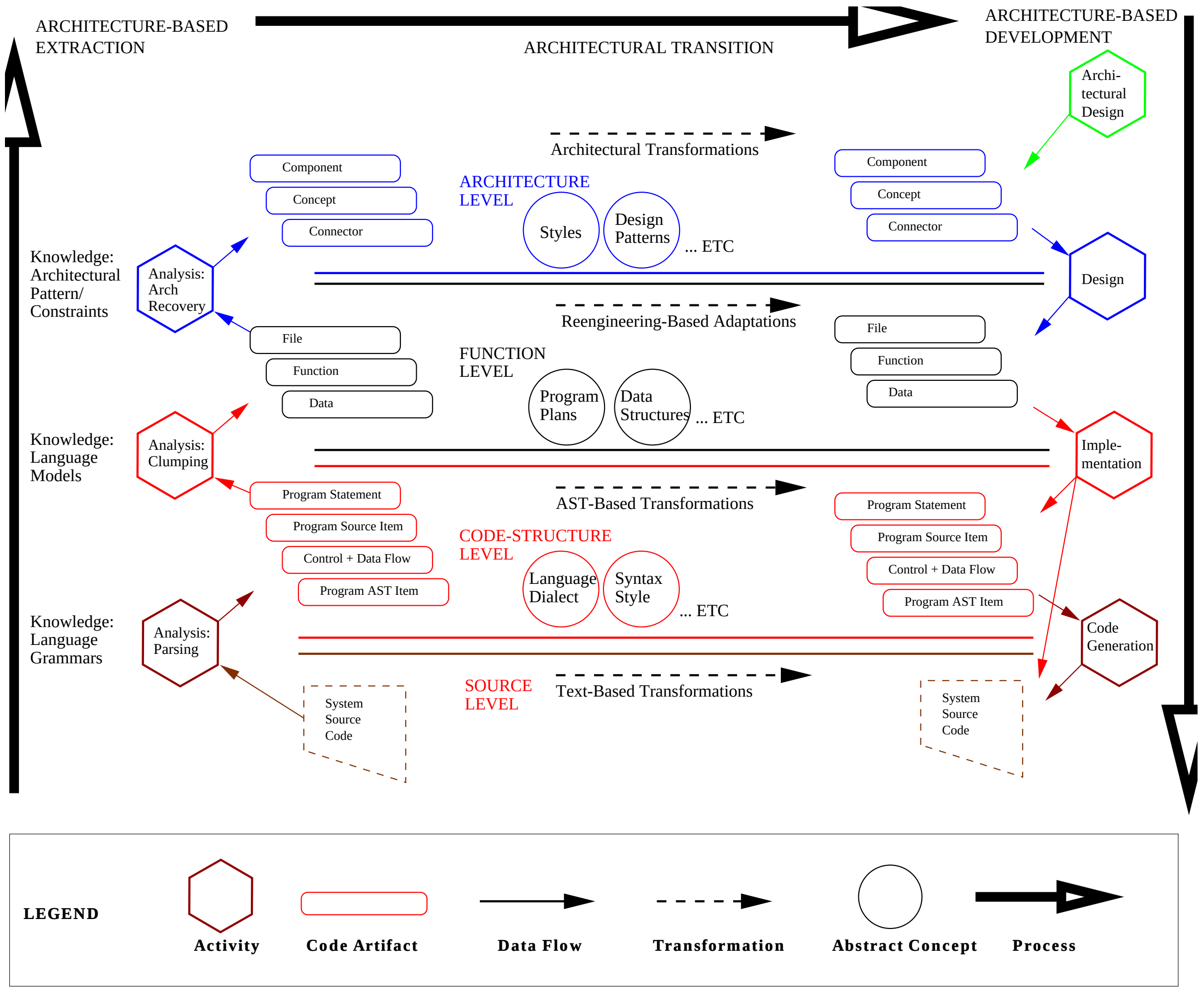

Figure 2: A More Detailed Horseshoe: Conformance, Transition, Development

assessing their conformance, and analyzing them is often not enough. We need to use this information as a guide to reengineering architectures. A system designer may need to use information about the existing system—its strengths and weaknesses—in determining how an architecture might evolve. In particular, questions can now be reasonably asked about what new technologies the design might incorporate or what new architectural style or styles it should reflect, or what new patterns it might encompass.

4: Reengineering Software Architectures

Architectural reengineering tasks abound at the moment. Many organizations are updating their core technologies, moving to incorporate the World-Wide Web, porting their systems to PCs, moving to an object-oriented architecture, distributing a system using a three tiered client-server architecture or using CORBA, layering a system for portability and device independence, and making the parts of a system more loosely coupled by using publish/subscribe connectors between the parts.

These tasks can be seen within the context of the horseshoe, as depicted—with more detail this time—in Figure 2. We distinguish four kinds of transformations within the context of the horseshoe: architectural transformations, function level transformations, code-structure level transformations, and textual transformations. Textual (or string-based) transformations are typically done through string matching and replacement—for example, recognition of candidate Y2K[1] exposures in code are frequently handled in tools utilizing regular-expression matching and replacement. Other examples in which textual transformations are useful include variable-name changes or straightforward code keyword changes during ports between platforms or compilers.

Code-structure-based transformations that exploit parsed versions of a system can accommodate changes directly at the AST level, supporting changes that are immune to surface syntax variations. Code-structure transformations are frequently done to port a system to a new implementation language (e.g. Fortran to C) or to automati-

1.“Year 2000” is typically abbreviated Y2K cally make systematic changes to the code (e.g. more advanced approaches to Y2K fixes). While reengineering approaches [17] typically distinguish between AST-based representations (“syntactic”) and transformations and AST representations augmented with derivative data and control flow information (“semantic”), we include both in the context of code-structure. The advantage of having flow information is that it allows for changes to code where the change is dependent on a complex control-flow condition.

Function level changes are typically done to move to a different encapsulation of functionality (e.g. moving from a functional design to an object oriented one or moving from a hierarchical database to a relational database model). Most wrapping approaches fit within the function level.

Architecture level transformations may involve changing the types of components and connectors used, the topology, the allocation of functionality to structure, and the run-time patterns of control and data exchange.

Within this multi-level view of transformations, the horseshoe is intended to depict architecture-level transformations as the context in which lower-level transformations live. For example, in a migration from a traditional client/server to a CORBA-based architecture, several code wrapping transformations might be used to reuse portions of the original architecture in a new context. In addition, the code structure might need to be transformed so that the client/server communication is changed to operate through the new “ORB” architectural component.

Clearly each level will have effects on its neighboring levels: an architecture must be realized in functions that are themselves realized in code. This is why an *integrated* model for reengineering tools is necessary. In particular, we need to specify the relationships between the architecture and lower levels in the reengineering process.

When migrating to an architecture-centric model of software development all levels of representation in the horseshoe must be designed, maintained, and analyzed for correctness. The practice of discovering a software architecture involves *a priori* knowledge about how systems are structured at each of these levels. It is not enough to understand that a particular design style (such as a client-server framework) has been used to architect a system if one does not know *how* this style is instantiated at the code level.

Architectural recovery is highly symmetric with the reengineering program understanding task. Many reengineering tools, process models, and cognitive models rely on local (bottom-up) conceptual recovery informed by the higher-level abstractions (top-down). Similarly, system experts construct the mappings both downwards from architectural concepts, as informed by high-level system design knowledge, and upwards by matching architectural and design patterns in the function and code levels, and program plans and data structures in the code level.

However, just as reengineering tool-based approaches to

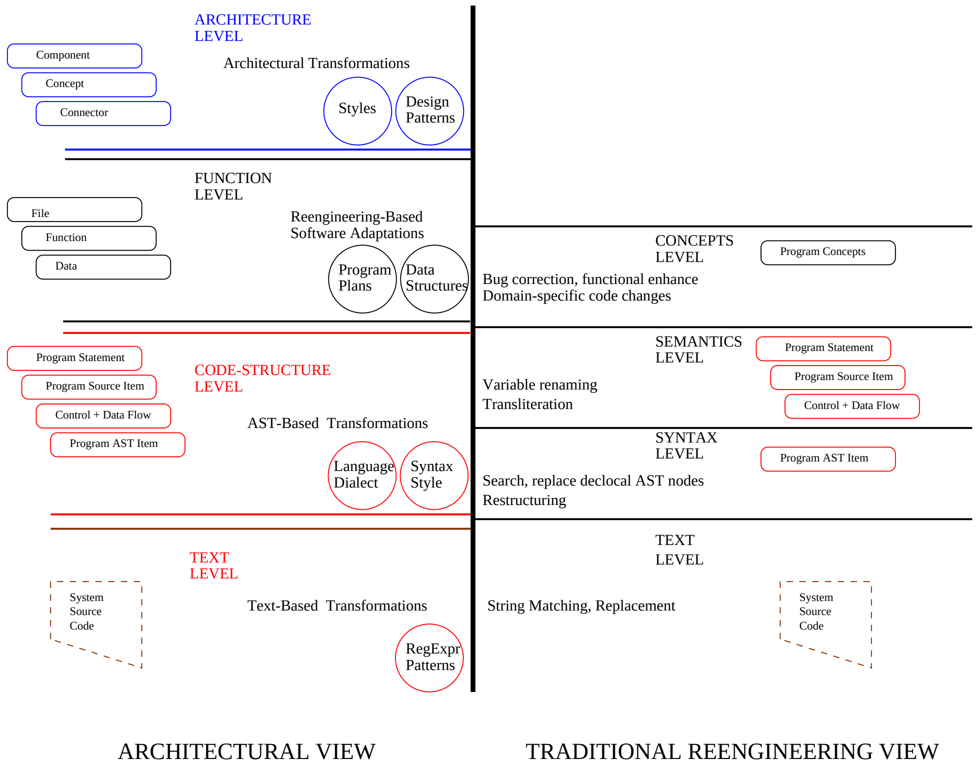

Figure 3: Architectural vs. Reengineering View of Software Artifacts

program understanding support the mapping between code and program plans, the architectural toolset must support mappings between architectural concepts and artifacts. Figure 3 shows the multi-tiered view of “program plan concepts” which reengineering tools typically describe, alongside the CORUM II model informed by the architectural view. In a tool supporting the architecture-to-system mappings, the abstractions of software architecture must be explicitly supported by the tools used at the various levels. This is not currently the case in the sense that code-based reengineering tools are not informed by architectural concepts. Tools exist at each of the levels, but they have their own concepts, vocabularies, representations, and manipulation techniques. Two criteria must be met to unify these tools:

- they must share a common *syntactic format* so that they can exchange information; and
- they must share a common *semantic underpinning*, so that explicit mappings can be drawn and checked among the levels.

The first criterion is relatively easy to achieve. It is simply a matter of negotiating a data format that is agreeable to a wide variety of tools---what HTML has done for the World-Wide Web. The second criterion is more difficult to achieve, and it is to this criterion that we turn our attention now as a key requirement for CORUM II.

5: A Semantic Model for the Horseshoe

In our proposal for CORUM II, we will use the taxonomy of architectural elements introduced in [14] and show how this provides a set of features that can be reasoned about in architecture-based design, analysis, and reengineering.[1] Our claim is that the set of features unifies the levels: they can be extracted and manipulated at *all* levels within the horseshoe.

These features are a set of semantic language-independent criteria that describe how an element receives, manipulates, stores, and transmits both data and control, as well as how and when it binds with its neighboring elements. They describe how architectural elements—components and connectors—resemble and differ from each other from an externally-visible perspective.

When crafting the set of features, two views have been shown to be useful. The temporal view considers the execution of the element as an episode in time. The temporal features are those that can be discerned by watching the element execute. The static view is complementary; those features that can be discerned by examining the specification or the code of the element. One view captures an element’s static invariants while the other captures an element’s temporal “signature”.

1. Other work similarly attempts to categorize software system components by (a different set of) feature vectors.[1]

The temporal view of an architectural element describes how it *may* behave over time. This views characterizes the execution of an element as a series of temporal “episodes” in terms of threads of control that flow through the element. Three important times within an element’s life are characterized: when it is created, when it is destroyed, and in between.

Useful distinctions among architectural elements may be made in terms of this temporal view. For example, an element that does not allow data to be dispatched in the middle of an execution episode has quite different characteristics (and usages) from an element that does. An element that does not accept control during execution cannot be re-used in a host of situations in which its re-entrant counterpart can.

The temporal view’s features are enumerated below. Of course, other features are possible, but these have been shown to be sufficient to distinguish the properties of a large set of architectural elements [11]. If we define *t_s* as the time at which a component starts executing (receives control), and *t_e* as the time at which a component completes executing (relinquishes control), we define the features as answers to a set of questions about the element being described: “Does every instance of this type of element dispatch data at *t_e*?”, etc. Temporal features include:

- Times of control acceptance: this feature describes the times when an element can receive control.
- Times of data acceptance: this feature describes the times when an element can receive data
- Times of control and data transmission: these features describe the times when an element can transmit data and control
- Forks: this feature is true if the element can spawn a new thread of control
- State retention: this feature describes the conditions under which an element retains state (i.e., where its behavior is a function of previous invocations). Possibilities are that it cannot do so, that it can do so but only within a single thread of control, or that it can do so across multiple threads of control.

The static view of architectural elements describes information based on static information, which does not change as a function of time or as a result of executing, using, or interacting with the element being characterized. The static structure of an architectural element does not change once it has been specified.

The static view suggests a set of features for architectural elements. Like the temporal features, the static features are posed as answers to questions about the element, in this case questions about its time-invariant capabilities and properties. The static view’s features are enumerated below, as with the temporal features, as a set of categories and an explanation of the reasoning behind each category:

- Data scope: what is the largest scope across which data can be passed by the element? Possible answers are that the element passes data within a virtual address space, across virtual address spaces but within a single physical address space, or across a network.
- Control scope: what is the largest scope across which control can be passed by this element?
- Transforms data: are the element's outputs a transformation of its inputs?
- Binding Time: when are the sources and sinks of an element (its ports and roles [6]) bound? Possible answers are at specification time, invocation time, or execution time.
- Blocks: does this element suspend when it transfers control to another element?
- Relinquish: if this element has a thread of control, does it ever voluntarily relinquish it?
- Ports: what are the directions of connections with other elements (i.e. is data or control coming in, going out, or some combination)?
- Associations: what number of elements can bind to this port simultaneously?

For example, a Unix filter has a data scope spanning a physical address space, it transforms data, it has no control scope (i.e. it never transfers control), it binds its ports at invocation time, and so forth. A remote procedure call has both data and control scopes spanning distributed address spaces, doesn't transform data, is bound at specification time, and so forth. The temporal features of these elements can be similarly specified.

5.1: Using the Semantic Model

This taxonomy has already been used for architectural representation [10] and architectural pattern matching [11]. Here we propose that it is a suitable foundation not only for architectural reengineering, but for uniting the reengineering tasks at various levels within the horseshoe.

For example, consider reverse engineering a system that consists of four subsystems, called A, B, C, and D. By fact extraction at the code and function level, we learn that A sets up a socket connection to B; in terms of the features, the connector between A and B transmits data at any time and has a data scope that spans virtual address spaces. In addition, we learn that the connector between C and D is a procedure call, and hence this connector transfers both data and control, but *within* a virtual address space.

We can quite simply derive these features via code and function level extraction: components and connectors can be characterized according to the set of features. A standard object, as typically implemented in C++ or Java, can be characterized as follows: it only accepts control and data (from its invoker) at *t_s*, it retains state between invocations within a single thread of control, its data and control scope are a virtual address space, it is bound to its ports at specification or execution time, and so forth. Similarly, the CORBA RMI (remote method invocation) communication mechanism can be characterized as follows: it only accepts control and data at *t_s*, it only transmits control and data at *t_e*, it doesn't retain state between invocations, its control and data scope are distributed, it is bound to its ports at execution time, and so forth.

The features, once derived, can then be percolated up to the architecture level (the left-hand side of the horseshoe). This will allow for a single continuous representation scheme for reengineering artifacts of any level of abstraction. Why is a single continuous representation scheme necessary?

Consider a second example, as depicted in Figure 4: here a system is represented as having two subsystems, 1 and 2, that communicate with each other. The realization of this communication among the code-level artifacts takes many forms: function calls, exception raising/catching, and shared memory access. When these subsystems are realized at the architectural level, they are typically depicted as single (aggregated) elements. In such a case, they are typically also shown as being connected by a single relation. But this relation is actually a composite. To reason about reengineering such a relation between subsystems, we need to know about the properties of the relation between the subsystems (for otherwise, we cannot plan the mechanisms for the reengineered version of this relation), and so we need to be able to represent the properties of the composite connector between 1 and 2. We can do so by combining the properties of the elemental connectors as shown in Figure 4.

When a system is reengineered (moving across the horseshoe), its features might be maintained, or perhaps the system will be redesigned in such a way that the features are altered. For example, it might be that the reengineered system will be more fully distributed. In the previous example the connections between subsystems 1 and 2 might be changed to have a wider data and control scope. The point is not that reengineering will always *preserve* these semantic features, but rather that these features need to be explicitly represented so that they can be explicitly reasoned about.

Finally, when the system is redeveloped (moving down the right-hand side of the horseshoe) the features that were designed into the architecture must be manifested by the implementation mechanisms chosen. For example, in the case of subsystems 1 and 2 having a wider control scope, we might choose remote procedure call, or CORBA remote method invocation to realize this connection. Such a connection would satisfy, at the code level, the feature requirements specified at the architecture level.

6: Related Work

This work is closely related to many commercial reengi-

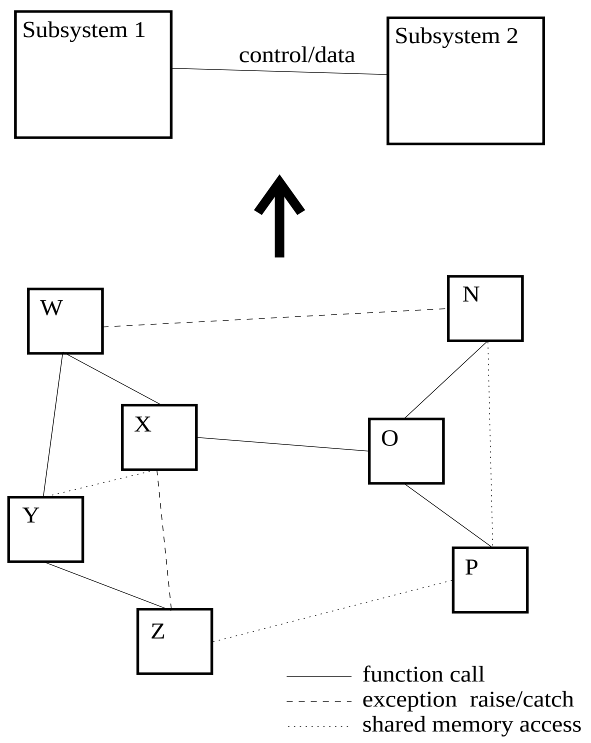

Figure 4: Aggregation of Connector Semantics.

neering approaches [9,7,22] that exploit parsed intermediate-representations based upon ASTs to support query mechanisms that allow one to recognize concept structures and, in some cases, to transform these structures. Common research-based reengineering representations [17,27,21] have been attempting to push beyond this paradigm to not only manipulate the underlying source ASTs structures, but also to augment the ASTs with extracted control and data flow information. In a project with similar objectives to CORUM II, described in this proceedings [15] presents an intermediate representation for reverse engineering that allows for multiple views of an extracted system and for the annotation of these views with analysis results.

While there are relatively few tools and representations dedicated to the capture, evaluation and manipulation of software architecture, several tools and languages [19,8] have been created and are evolving in support of this task. Other tools have been developed that support the creation of architectural views from the functional or lower levels, based on recognition of connector cliches [28]. Also, Architecture Description Languages (ADLs) have enjoyed a great deal of research attention in recent years; [20] presents a comprehensive survey. An ADL designed for interchanging information among ADLs is the ACME [6] language.

Computer-Aided Software Engineering (CASE) tools have been moving towards supporting the creation of actual source code from abstract “design” specifications for some years. However, the level of “design” is more appropriate to the functional level of the horseshoe, and does not include such system aspects as architectural styles or design patterns. Other forward engineering tools such as 4th generation languages support a higher-level specification of common domain or application-specific tasks, but once again do not allow the specification of an architectural view.

7: Background: CORUM I

The work in this paper builds directly upon the concepts for tool interoperability and data exchange outlined by CORUM [26]. CORUM was initially proposed as a discussion model with respect to a perceived need for a Code-Base Management System (CBMS) that could represent information about software systems. Its initial instantiation was designed to provide a mechanism for storing extracted information (e.g., ASTs) and products produced from this information (e.g. Symbol Tables, CFGs, DFGs, Data Slices, Programming Plans) as well as to provide interoperability between tools that accessed and produced this information. While the model and behavior described by CORUM has been implemented, this work was primarily intended to spur discussion among practitioners for the evolution of a shared CBMS. The initial CORUM environment consisted, in much the same way as commercial CBMS’s, of four parts: language gateways, an information model, processing tools, and a component repository.

CORUM’s conception was that language gateways were made up of set of tools that were capable of parsing and loading information about source code into the CORUM representation. This information’s fundamental representation was in a particular schema for ASTs. The CORUM representation was initially described as the information model. This model was composed of the various products generated from directly processing the source code (e.g., through parsing and loading) as well as artifacts obtained from applying processing components (e.g., from analyzing information already stored in the information model by other tools). The tools that operate on the contents of the information model either produce new products to be stored in the information model (including artifacts such as metric calculations or flow graph annotations) or visualizations of those products.

CORUM I’s initial focus was narrow—to support the interoperation of a small set of reengineering tools, each with its own internalized reengineering schema. No particular emphasis was placed upon developing tool coordination; rather it sought to provide a middle-layer of data representation that encompassed and combined the semantic meanings of several tools. Others have, in closely related work (e.g. Toolbus [3]), pursued a much larger goal—the creation of a CBMS that supports not only the interchange of data among reengineering tools, but also the coordination of the behaviors of those tools.

The hope for CORUM initially was to support interchange between existing tools. A set of standard APIs using an accepted schema for the representation of software artifacts would at least offer tool developers the option of building their toolsets on top of these APIs, and those tools that utilized this language could readily share the information that they had populated and manipulated in their respective repositories.

8: Future Work

In pursuit of the goal of a unified model based on a common syntactic format and a common semantic underpinning, our future work is two-pronged. First, we are working with several groups concerned with the representation of software knowledge for the purposes of recovering and manipulating architectural representations to develop a schema rich enough to support the tasks and needs of each group. Second, we are working on defining a semantic basis for the categorization and grouping of software artifacts. These two tasks will be realized through the construction of a representational schema that unifies the artifacts produced by low-level software analysis, such as parsing, with the high-level concepts utilized by software architects.

One representative candidate formalism for the specification of software knowledge is *Annotated Term Format (ATF)* [23], a language specifically designed for the data exchange between (possibly) heterogeneous or distributed components in a software system. The exchange of ATF information is at the heart of the interconnection of sub-tools in the ASF+SDF workbench introduced earlier.

We are simultaneously experimenting with extracting architectural information with Reasoning Inc.'s analysis toolset [9] through extending their proprietary CBMS to represent architectural concepts. The task of exporting information from this representation to an open format accessible to other tools is not impossible, but neither is it a simple task to re-import artifacts to their CBMS from other tools' analyses.

9: Conclusions

This paper has presented a generic framework for the integration of architectural and code-based reengineering tools. This framework is needed because there are a wide variety of stand-alone reengineering tools that operate at different levels of abstraction ranging from "code-level" to software architecture. For the purposes of reengineering a *complete* system however, these tools need to be able to share information so that not only can the code be updated or corrected, but also so the system's architecture can be simultaneously rationalized or modernized. To this end, we have built upon the CORUM model of reengineering tool interoperation to include concepts and tools from software architecture. This extended framework—called CORUM II—is organized around the metaphor of a "horseshoe", where the left-hand side of the horseshoe consists of fact extraction from an existing system, the right hand side consists of development activities, and the bridge between the sides consists of a set of transformations from the old to the new.

We have argued that any combined representational framework for reengineering and architectural manipulation tools needs to have a strong semantic underpinning to be able to extend architectural reasoning upward on the left hand side of the horseshoe (where extracted code-level facts contribute to an understanding of software architecture), across the horseshoe (where architectural concepts are manipulated), and downward on the right hand side of the horseshoe (architecture based development). We have shown how an existing taxonomy of architectural elements supports this form of reasoning.

This work is a part of a larger context at the Software Engineering Institute—the construction and use of "Dali"—a workbench designed to support architectural recovery and analysis, and "CORUM", the attempt to provide a representational schema that can accommodate the information needs of a significant set of reengineering tools. Our next step in the evolution of CORUM II as the heart of "Dali II" is to formalize a specification of a schema for CORUM II that is sufficient to support both reengineering analysis and architectural reasoning, and to examine how the integration of tools utilizing CORUM II can be best achieved within the context of a CBMS.

10: Acknowledgments

Thank you to Alex Quilici, Chris Verhoef, Ric Holt, Hausi Müller, Kenny Wong, Dennis Smith, and Scott Tilley for their many helpful discussions about reengineering and architectural analysis tools and approaches.

References

[1] P. Bailes, P. Burnim, M. Chapman, and D. Johnston. Derivation and Presentation of an Abstract Program Space for Ada. In *Proceedings of the Fourth International Conference on Program Comprehension*, 1996.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[3] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. *Lecture Notes in Computer Science*, 1101:286–305, 1990.

[4] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Müller, J. Mylopoulos, S.G. Perelgut, Stanley, and M. K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4), 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented*

Software. Addison-Wesley, 1994.

[6] David Garlan, Bob Monroe, and David Wile. ACME: An interchange language for software architecture, 2nd edition. Technical Report CMU-CS-95-219, Carnegie Mellon University, 1997.

[7] COSMOS Group. Introduction to the COSMOS Technology. http://www.technforce.nl/m2_cosmos.html, 1998.

[8] R. Holt. Introduction to PBS: The Portable Bookshelf. Technical report, University of Toronto, 1997.

[9] Reasoning Systems Inc. Transformation Software Overview. http://www.reasoning.com/products.html, 1998.

[10] R. Kazman. Tool Support for Architecture Analysis and Design. In *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, pages 94–97, 1996.

[11] R. Kazman and M. Burth. Assessing Architectural Complexity. In *Proceedings of 2nd Euromicro Working Conference on Software Maintenance and Re-engineering (CSMR '98)*, pages 104-112, 1998.

[12] R. Kazman and S.J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *Proceedings of the 5th International Conference on Software Reuse*, pages 290–299, 1998.

[13] R. Kazman and S.J. Carriere. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, April 1999.

[14] R. Kazman, P. Clements, L. Bass, and G. Abowd. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *Proceedings of COMPSAC 97*, pages 14–17, 1997.

[15] R. Koschke, J.-F. Girard. An Intermediate Representation for Reverse Engineering Analyses. In *Proceedings of the Fifth Working Conference on Reverse Engineering*, 1998.

[16] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and S.J. Carriere. The Architecture Tradeoff Analysis Method. In *Proceedings of ICECCS'98*, 1998.

[17] W. Kozaczynski, J. Ning, and A. Engberts. Program Concept Recognition and Transformation. *Transactions on Software Engineering*, 18(12):1065–1075, 1992.

[18] W. Kozaczynski, J. Ning, and A. Engberts. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.

[19] H. A. Müller. *RIGI - A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications*. Ph.D. thesis, Rice University, 1986.

[20] N. Medvidovic, A Classification and Comparison Framework for Software Architecture Description Languages, Technical Report UCI-ICS-97-02, University of California at Irvine, 1997.

[21] S. Rugaber and L. Wills. Creating a Research Infrastructure for Re-engineering. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 98–102, 1996.

[22] S. Tilley. Discovering Discover. Technical Report CMU/SEI-97-TR-012, Carnegie Mellon University, 1997.

[23] M. van der Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In *Proceedings of the 23rd Seminar on Current Trends in Theory and Practice of Informatics: SOFSEM '96*, 1996.

[24] M. van der Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *Proceedings of the Sixth International Conference on Program Comprehension*, 1998.

[25] N. Weiderman, J. Bergey, D. Smith, and S. Tilley. Approaches to Legacy System Evolution. Technical Report CMU/SEI-97-TR-014, Carnegie Mellon University, 1997.

[26] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An Architecture For Interoperable Program Understanding Tools. In *Proceedings of the Sixth International Conference on Program Comprehension*, 1998.

[27] S. Woods, A. Quilici, and Q. Yang. *Constraint-Based Design Recovery for Software Re-engineering: Theory and Experiments*. Kluwer Academic Publishers, 1998.

[28] A. Yeh, D. Harris, and M. Chase. Manipulating Recov-

ered Software Architecture Views. In *Proceedings of ICSE 19*, pages 184–194, 1997.