# Bi-directional approach to modeling architectures

Maritta Heisel
Institut für Angewandte Informatik
Technische Universität Berlin
Sekr. FR 5-6, Franklinstr. 28/29
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de

Balachander Krishnamurthy
Software Engineering Research Dept
AT&T Bell Laboratories
Room 2B-140, 600 Mountain Ave
Murray Hill, NJ 07974 USA
bala@research.att.com

**Abstract**

Software architecture can be broken down into a collection of architectural styles and services. We seek to capture a set of criteria to characterize architecture styles. The presence or absence of a criterion may be reflected in the features or limitations of the realization of an architecture. Our idea is to enumerate a set of criteria for a given architectural style (e.g. event-action) and try to map it to a formal specification of an instance of the style. We then map the formal specification to a reverse engineered level of the code. The goal is to bridge the gap between the architecture and the realization. In addition, the breakup into criteria and the mappings provides a way to locate functional and non-functional aspects of the architecture in the code.

## 1 Introduction

Software architecture is a description of a set of components and relationships among them, how such components can be put together to build some class of structures, and implications on the resulting structures' usability, integrity, overall performance and construction cost. The notion of architectural styles has received some attention recently [Dag95]. An architectural style consists of rules, constraints and a collection of characteristics that are both common to application architectures that are designed to solve a family of problems and are sufficient to identify such a collection [BCK95]. Architectural styles range in complexity from simple pipelines, to complex transaction processing and real time applications. For example, the ACID (atomicity, consistency, isolation, and durability) characteristics [GR93] identify a system in the transaction processing style, and can be used as a way to partition parts of a larger application.

In this paper we address three steps:

- Refinement of architectural style into a set of *criteria*
- A bi-directional mapping between criteria and the formal specification
- A bi-directional mapping between the formal specification and a reverse engineered layer of the actual implementation.

Figure 1 depicts our proposed Bama architecture (bidirectional approach to modeling architectures). We begin with a set of architectural styles such as pipe/filter, Online Transaction Processing (OLTP), event-action etc. We consider one of the styles (event-action) and look at its set of criteria (defined below). The criteria in turn can be mapped onto a formal specification of an instance of the style. The realization (source code) of the style instance can be reverse engineered to a level above that of the code—the resulting

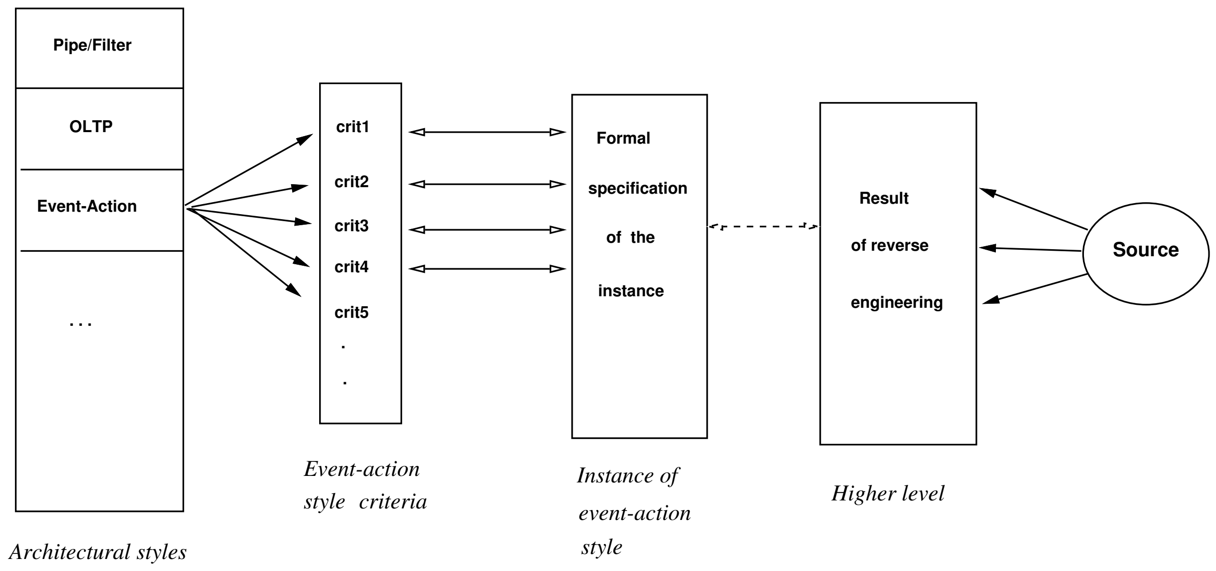
Pipe/Filter
OLTP
Event-Action
...
Architectural styles
crit1
crit2
crit3
crit4
crit5
Event-action style criteria
Formal specification of the instance
Instance of event-action style
Result of reverse engineering
Higher level
Source

Figure 1: Bama architecture.

component is RRE (result of the reverse engineering). The mapping between all these components is bidirectional since we approach the problem from both directions. The lighter two-way arrow between the formal specification and the RRE indicates that this link is not as strong as others; i.e. not enough tools exist as yet to do this.

Before we look at the various components of the Bama architecture, we briefly discuss their temporal ordering. In the construction of most software we begin with high level architecture and proceed towards code development. In the presence of legacy code, we work backwards from reverse engineered code apart from creating a formal specification for the existing realization. We believe that our mappings would be beneficial independent of the order in which some of the Bama components were created. For example, future realizations of a particular style can benefit from the criteria that have been enumerated; the mapping between the formal specification and RRE can help in any modifications to the code. At each step we use any of the components available to create or improve the other components and mappings.

The criteria are a decomposition of a style into parts that can be treated as logical components of any realization of the architectural style. We define an architectural style criterion as an inherent property of the style which would be reflected in most realizations of the architecture. It can reflect a functional or a non-functional property and can be a combination of them. In the event-action architectural style, a set of events are matched subject to conditions, and actions are to be triggered as a result of the matching. In this style, the order of matching of events is a criterion: it may be necessary to match events in order of occurrence. Other examples of criteria include matching co-occurrence of events (say in a distributed implementation) and taking contextual information of events into account while matching.

Having a set of criteria permits future realizers of the implementation to have a reasonably comprehensive working set to consider *a priori*. Early decisions regarding inclusion or exclusion of a criterion can help in a modular realization. Second, the set of criteria can be analyzed for degrees of mutual exclusivity in any realization. Presence of a criterion may have implications on other criteria. Third, the criteria span a set of possible implementations and if we are able augment the criteria set, it would be of help for future realizations.

Our second step is to map the set of criteria onto a formal specification of an instance of the architectural style realization. A formal specification describes *what* a system should do and is silent on *how* it should be done. This would give a firmer grounding of the notion of criteria. The key observation is that we will be able to identify the precise parts of the formal specification that deal with each of the criteria.

The final step is to map the formal specification of the instance to the implementation via reverse engineering techniques, assuming a realization already exists. Currently this is the weakest link since most of the reverse engineering techniques take us only a small step higher than the code itself. Instead of trying to map the code directly onto the architecture, our goal is to move upto the level of the formal specification of an instance of the style.

In essence, we are trying to bridge the gap between architectural style and code by working from both ends. The notion of criteria is informal; thus any persuasion on our part has to be based on arguments that rely on examples, assertions and appeal to logic. While we do not have an algorithm for a reasonable and complete enumeration of criteria of an architectural style, a starting point might be the list of its necessary and sufficient conditions as presented in [BCK95]. Architecture style criteria provide a way to narrow down the properties that end up being reflected in the code. Interposing the formal specification of the criteria of an instance gives us a more precise way of comparing different realizations of a particular style.

The rest of the paper is organized as follows: We begin with an example architectural style to describe our notion of criteria in detail. We then describe a realized instance of this style. We use a formal specification of the instance in Z to provide a mapping between the criteria and the formal specification. We discuss the mapping between the formal specification and the reverse engineered level of the implementation. We discuss details of our choice of reverse engineering tools. We then evaluate our mapping and see if our approach generalizes to other architectural styles as well as other realizations. We look at related work in the area and finally conclude with some future directions.

## 2 Case study

In this section we look at a case study we conducted to examine the notion of architectural style criteria in depth. We generated a set of criteria for one style (event-action), and examined a realization of the style. We then came up with a Z specification for that realization and mapped a subset of the criteria to this specification (not all the criteria were present in the chosen realization). Finally, we look at the use of the result of the reverse engineering (RRE) component.

### 2.1 An architectural style example with criteria

A system to watch for patterns of signals from a set of remote computers and take action (such as raising an alarm) when selected patterns are detected is an example of of an event-action architecture style. Arbitrary collections of programs (or users) interact in a loose manner via the event broadcast (and matching) mechanism. Systems that are instances of this architectural style have been built and such systems can be modeled via a Finite State Machine or a Petri Net. YEAST [KR95] is one tool that implements an instance of an event-action style (see Section 2.2).

To understand the notion of criteria associated with the event-action architectural style we formally specified the behaviour of the system. The patterns of events together with the associated action (called an event-action specification) can be matched by a matching engine (which may be central or distributed) in many ways.

We enumerate the criteria for the event-action architectural style:

1. **Matching style**: The occurrence of an event can be recognized by a matching engine either *transiently* or *perpetually*. For example, temporal events which are based on a monotonically increasing clock would result in a perpetual match. If event patterns are complex, combinations of events would have to be matched in parallel if matching style is transient.
2. **Context**: Events occur in a context and the elements of the environment of the event constitute its context. In the domain of events generated in a network environment, the identity of the machine on which the event was generated, the time at which it was generated, the user or process id responsible for generating the event are all contextual elements of the event. It is likely that specifications may want to take advantage of the contextual information of events. The matching engine may permit constraints on matching based on the context of events.
3. **Constraints beyond context**: The matching engine may allow additional constraints such as recent match history, the number of times a particular event has occurred etc., in constraining further matches. This would require maintaining additional information about the events.
4. **Event independence**: Occurrence of events once known by the engine can be ordered in some fashion before matching is attempted. Event independence would permit matching in any order while matching a pattern of events. A serialized event matching application would explicitly require the absence of this criterion.
5. **Handling unmatchable specifications**: It is possible to have event-action specifications whose event patterns will *never* match once certain time event descriptors stop matching. Ideally, a system would detect unmatchable specifications and discard such specifications. Ability to detect unmatchable specifications and taking remedial action is a criterion that would impact performance and correctness.
6. **Grouping specifications**: Logically related sets of specifications should be grouped together and named so that they can be referred to by the user interface. Being able to interact with an arbitrary subset of specifications enables coarser grained handling. Specifications should be able to belong to multiple logical groups.
7. **Privacy/Encapsulation**: Specifications should be private by default but permitted to cross access boundaries based on authenticated requests. By permitting sharing of specifications, applications can register shared interest and have common actions triggered. For example, a diverse set of people may want to receive notification when a particular pattern of event occurs.
8. **Extensibility**: It should be possible to build new constructs based on the primitives of the event-action language. New object classes, attributes should be specifiable and, most importantly, the system should be able to watch for new kinds of events.
9. **Self-modifiability**: The ability to add and delete to the internal collection of specifications as a result of external actions is a useful criterion. It permits applications to dynamically reconfigure their specifications.
10. **Event attributes**: This criterion permits external events to be structured so that matching can be restricted based on a variety of factors. The event could be broadcast (i.e. match all valid specifications), narrowcast (i.e. match only specifications that meet a condition; for example belonging to a particular specification group), time bound (i.e. how long the event should be stored so that it can match future specifications) etc.
11. **Matching co-occurrence of events**: In a distributed implementation of an event-action style application it would be useful to be able to watch for concurrent events.

## 2.2 Yeast: A realization of the style

Yeast runs as a daemon on a single machine on a network and accepts client specifications from users. When these specifications are matched, user-specified actions are triggered by Yeast. Client commands—such as, add, remove, and suspend specifications—can be issued from any of several machines in the network. Yeast permits arbitrary actions to be triggered when event patterns of interest are matched. It can match temporal events and nontemporal events, such as changes of attributes in objects belonging to a variety of object classes. The event language of Yeast allows one to build more complex event expressions, using the connectors *then* (sequencing), *and* (conjunction) and *or* (disjunction). The list of predefined attributes for these object classes can be extended, as can the classes themselves. With the use of external event notifications (called *announcements*), Yeast can model a wide range of applications that follow the event-action paradigm. Yeast is approximately 8400 lines of C code with parts written in Lex and Yacc. Yeast runs on a variety of hardware and software platforms.

Since Yeast supports an event-action architectural style, several applications can be built on top of Yeast. The application first decides on the set of events of interest and the manner (both temporal ordering as well as combination of patterns) in which the events should be detected. Then, a collection of Yeast specifications is registered with a Yeast daemon. Several applications have been built on top of Yeast, ranging from automatic coordination of database files from a network of machines to a schedule maintenance system and are in production use in AT&T.

Several of the criteria of event-action styles discussed in Section 2.1 are reflected in the implementation of Yeast while some are absent. The criteria that are present in full include matching style, context, constraints, event independence, handling unmatchable specifications, grouping specifications, and extensibility. Some criteria are partially matched including privacy, self-modifiability, and event attributes. Criteria not reflected include matching co-occurrence of events.

## 2.3 The Z specification and its mapping to the criteria

A formal specification of a large subset of Yeast in Z [Spi89] was created by examining the various aspects of the Yeast implementation and its capabilities. The second author was co-designer and co-implementor of the Yeast system which simplified the task greatly. Thus, someone knowledgeable with the realization would come in handy at this stage. If the set of criteria were already available one could just create Z schemas for each of them. The complete Z specification of Yeast can be found in [HK95].

The three main parts of the formal specification deal with the specification state, matching intrinsics, and client commands. All the criteria enumerated above have been mapped onto the formal specification. The mapping was relatively straightforward since the formal specification describes the operational behaviour of the event-action realization. The inability to map some of the criteria onto the realization showed that these criteria were not present in the realization (e.g. co-occurrence of events). Thus, the map is useful both in locating and showing the absence of criteria. Criteria such as grouping and extensibility are explicitly represented in the formal specification. The ability to add new user-defined events (extensibility) is captured in the state portion of the formal specification as well as in operations changing the respective state components. In this section, we present portions of the formal specification that reflect two of the criteria, namely, grouping of specifications and extensibility.

Events in Yeast can be temporal (TE) or non-temporal (NTE). Non-temporal events can either be (predefined) object events (OE) or user-defined events (UE). These are the *primitive* events in Yeast. The specification of these event sets in Z is straightforward.

Each Yeast specification consists of an *event part* and an *action part*. It belongs to a *owner* and has an associated *label* for easy reference. It can be *suspended*, i.e. it remains present in the system but is not considered for matching. The current match status of a specification has to be recorded.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Spec}} \\ \hline
e : \mathit{EVENT\_EXP} \\
a : \mathit{ACTION\_EXP} \\
\mathit{match\_status} : \mathit{MATCH\_STATUS} \\
\mathit{own} : \mathit{OWNER} \\
\mathit{label} : \mathit{LABEL} \\
\mathit{suspended} : \mathit{YesNo} \\ \hline
\end{array}$$

The internal state of the YEAST system consists of three major parts: the specifications currently present in the system, the possible events, and the environment in which YEAST operates.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SpecState}} \\ \hline
\mathit{specs} : \mathbb{F}\, \mathit{Spec} \\
\mathit{specMap} : \mathit{LABEL} \rightarrowtail\!\!\!\!\!\!\!\!+\;\; \mathit{Spec} \\
\mathit{groups} : \mathit{GNAME} \leftrightarrow \mathit{LABEL} \\ \hline
\mathit{specMap} = \{ s : \mathit{specs} \bullet s.\mathit{label} \mapsto s \} \\
\mathrm{ran}\, \mathit{groups} \subseteq \mathrm{dom}\, \mathit{specMap} \\ \hline
\end{array}$$

The invariant of the specification state requires that each label be unique and that each label referred to by the relation *groups* belong to an existing specification. Specifications can be grouped together, as indicated by the component *groups*. This feature has to be built in explicitly. If groups are to be allowed the system must keep track of them. We thus see that the **grouping specifications** criterion is present. Location of this criterion in legacy code is discussed in Section 2.5.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PossibleEvents}} \\ \hline
\mathit{user\_events} : \mathbb{P}\, \mathit{UE} \\
\mathit{possible\_events} : \mathbb{P}\, \mathit{EVENT} \\
\mathit{attrs} : \mathit{OBJECT\_CLASS} \rightarrow\!\!\!\!\!\!+\;\; \mathbb{F}\, \mathit{ATTRIBUTE} \\
\mathit{class} : \mathit{NTE} \rightarrow\!\!\!\!\!\!+\;\; \mathit{OBJECT\_CLASS} \\ \hline
\mathit{possible\_events} = \mathit{TE} \cup \mathit{OE} \cup \mathit{user\_events} \\
\mathrm{dom}\, \mathit{class} = \mathit{OE} \cup \mathit{user\_events} \\
\mathrm{ran}\, \mathit{class} = \{ \mathit{oc} : \mathrm{dom}\, \mathit{attrs} \mid \mathit{attrs}\ \mathit{oc} \neq \varnothing \} \\ \hline
\end{array}$$

The feature that distinguishes YEAST from other existing event-action systems is the possibility of user-defined events. Users can define new object classes and attributes, i.e. the system fulfills the criterion of **extensibility**. The set of possible events depends on the defined object classes and attributes. There are client commands for changing the set of possible events.

The predefined temporal and object events are of course possible events. The set of attributes belonging to a given object class is recorded in the function *attrs*. The function *class* yields the object class a given non-temporal event belongs to. Each predefined object event and each valid user-defined event must have a corresponding object class. If an object class has a non-empty set of attributes, then there must be some non-temporal event referring to it.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{YeastState}} \\ \hline
\mathit{SpecState} \\
\mathit{PossibleEvents} \\ \hline
\forall s : \mathit{specs} \bullet \mathit{events}\ s.e \subseteq \mathit{possible\_events} \\ \hline
\end{array}$$

The schema *YeastState* relates the two sub-states. Each event contained in a specification must be a possible event.

## 2.4 Reverse engineering

The last component in Figure 1 is the source and the component to its left is the result of reverse engineering the code. Our goal is to move the code to a higher level and try to shrink the gap in Figure 1 from both sides. Towards this end, we have started identifying properties and criteria in the code via various reverse engineering techniques.

A brief explanation of the various locally developed tools we used is in order now. We used four tools: *cia*, a reverse engineering tool, *app*, an annotation pre-processor, *dotty*, a graphical editor tool, and *xray*, which is a combination of the first three tools. The reverse engineering tool CIA, the C Information Abstractor [CNR90], extracts entity and relation information between entities in C and C++ programs. Two databases are generated for the complete collection of code representing a system. The first, the entity database, consists of files, functions, global variables, types and macros in the code. The second, the relationship database, contains the closure of the relations between all the entities in the entity database. Once the program databases are created it is possible to run a variety of query operators to obtain a wide range of relationships between the entities. For example, the include file relationship can be generated as can the function call graph. It should be stressed that the information in the database is based on static analysis and thus only static dependency relationships can be extracted from the code.

The *app* [Dav95] tool is an annotation preprocessor system that preprocesses programs embedded with assertions (constraints) to generate instrumented source code. The embedded assertions in the source code are processed by *app* to create and run self-checking programs. *Dotty* [KN94] is a customizable graph editor with a programmable front end. It uses a graph layout tool *dot* [GKNV93] as a co-process and a programming language *lefty*, which is a graphics editor. With the help of the build tool *nmake* [Fow95] it is possible to *automatically* turn on *app* instrumentation of a piece of legacy code without touching the source code.

## 2.5 Location of a style criterion in the code

We now present an example of locating an architectural style criterion in the reverse engineered portion of the code. Consider, the specification grouping criterion: One could simply search the code–since YEAST is relatively small it would be possible to try and locate most of the code dealing with handling of specification group. This approach would not scale up however. Another approach is to look for meaningful names (especially if the code was written with proper function and variable naming guidelines). Such information is not often available and naming guidelines are not often followed.

Our approach is a combination of the static and dynamic information that can be extracted from the legacy code. To locate the portion of the code that implements specification groups, we ran about a dozen YEAST client commands that *should* trigger the code associated with specification grouping. The formal specification provided us with some hints which client commands to run. From the schema *SpecState* given in Section 2.3, we see that the type *GNAME* denotes group names and that groups are stored in the *groups* component of the schema. We then looked for schemas in the Z specification that have either an argument of type *GNAME* or change the *group* component of *SpecState*. The test cases were generated manually, although it is conceivable to automate this task. Our approach is feasible, since the Z specification of Yeast is short enough to search it in its entirety.

Merging the function trace output generated via the tool *app* with the static function call database generated from the *cia*, we were able to pinpoint the set of functions involved in implementing specification groups. The tool *xray* [Yih95] makes such location trivially possible. *No* code had to be written to accomplish this task.

Figure 2 shows the set of functions that are called to deal with the specification grouping criteria. The functions are clustered in the files they belong to, providing another level of abstraction. The functions in green are called once and the functions in red are called at least twice. The figure shown is a significantly reduced one as compared to the complete function call graph which has 262 nodes and 854 edges. In contrast, the figure shown has only 54 nodes and 102 edges of which only 22 nodes are red. Eliminating library functions and other utility functions lets us zoom in on just the handful of functions that implement the grouping criterion. Our confidence in the coverage of the functions arises from using the test cases generated with the help of the formal specification, and the fact that we are operating at the function rather than the line of code level.

## 3 Evaluation of the case study

Software systems undergo an *evolution* during their life cycle. If this process is to be supported appropriately, a truly bi-directional approach like the one presented here is of much advantage. In this section we evaluate the Bama approach of using the various components to help bridge the gap between architecture and code.

Section 2.5 presents an example of finding parts of the code dealing with a given criterion. Since the formal specification is short, it can be searched for operations and parts of the state space where the given criterion is reflected. The map between the criteria and the formal specification tells us where the criterion in question is located but may not enumerate *all* the relevant points in the specification. In the grouping criterion, for each client command we have a version that takes a group name instead of a list of labels as an input; this criterion is distributed over the whole specification. The mapping will only point to state space *SpecState* where the group information is maintained and to those operations that change this component. Thus, while it is necessary to search the whole specification, the mapping provides us with the *names* that have been used for the relevant entities. The syntactic information can then be used to locate all the relevant user operations. These operations are used to run test cases. The reverse engineering tools introduced in Section 2.4 can then be used (as described above) to locate the code dealing with the criterion in question.

We now consider the case of locating criteria in the formal specification. Suppose we have located a set of functions in the code dealing with a certain criterion. We can use an analysis of the code to rule out unrelated parts of the specification. Again, since the entire formal specification [HK95] of Yeast is only about 15 pages long (including comments) this task is doable. Consider locating parts of the formal specification dealing with specification grouping. A look at the functions shows that none of them has anything to do with matching, and that many of them deal with interacting with the user. This eliminates half of the formal specification since formalization of matching is the largest chunk. The fact that many of the located functions can be invoked by the user hints at considering the Z schemas that specify client commands.

Suppose a new version of a system has to be created that differs from the previous one in important aspects. Seeing how the old system was realized is important for the new realization. We can see if parts of the older system can be reused and if parts have to be changed. The changes should be reflected in the formal specification and the corresponding parts in the code can be located and changed accordingly.

We now see if our case study yielded results that can be generalized to other architectural styles. Clearly, the notion of criteria is generic since a variety of realizations of each style exist with overlapping sets of criteria reflected in them. For coming up with the set of criteria for event-action systems, two aspects turned out to be crucial: the existence of a fairly elaborate instance of the style, and a formal specification of this instance. An existing system can help generate the characteristic features of the style. However,

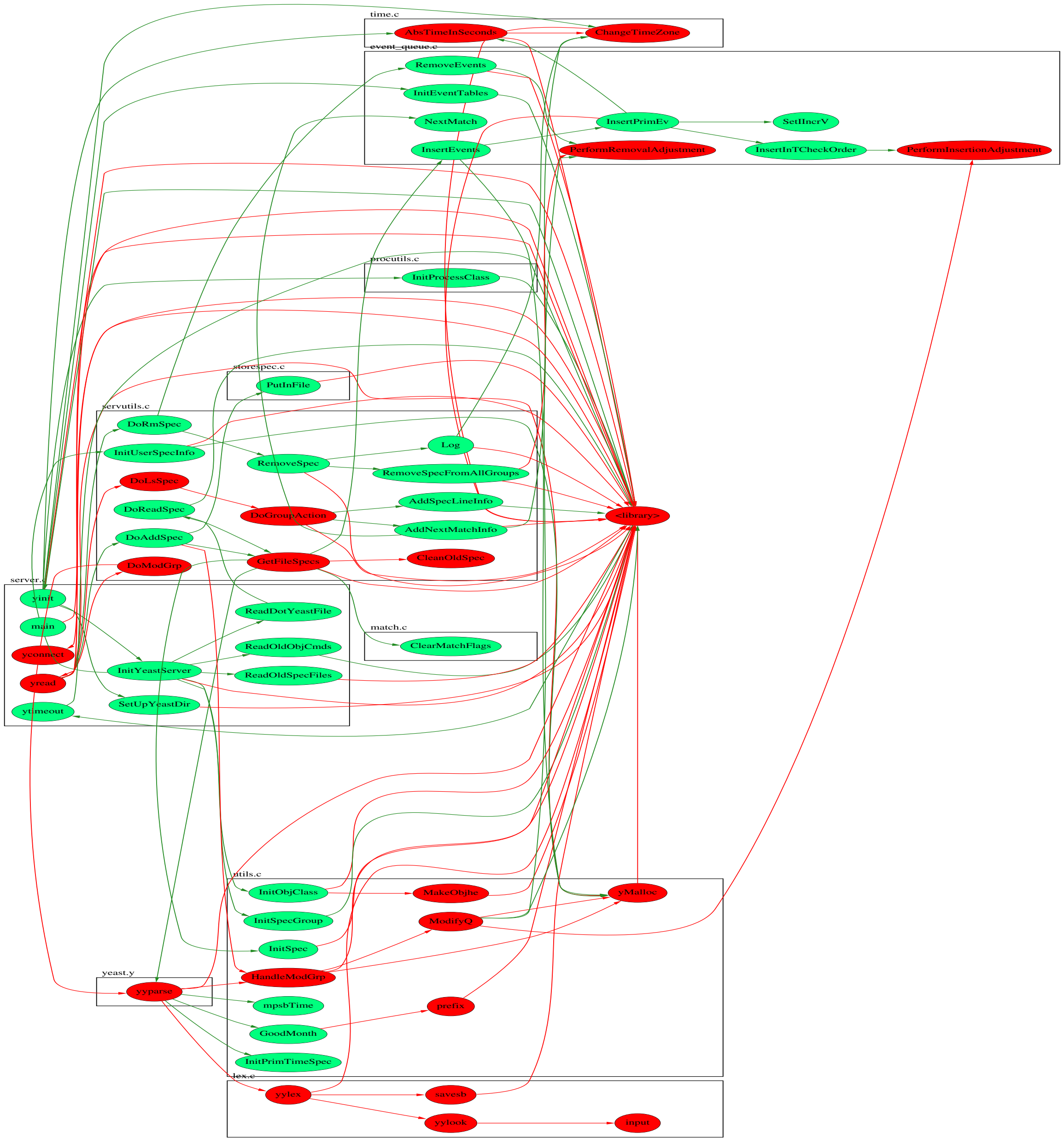

Figure 2: Functions dealing with grouping.

the code of an existing system is certainly too detailed, and the system documentation serves purposes other than exhibiting general features of architectural styles. What is needed is an abstract representation of the system that clearly documents the design decisions that have been taken. For each such design decision, possible alternatives can be discussed. This discussion will give rise to the definition of the criteria characterizing the given architectural style.

A formal specification is an ideal means to obtain such an abstract description of a given instance of the style. It abstracts away all implementation details and only presents the essence of the system. Each part of the formal specification can be analyzed in order to decide if the specified feature is indeed characteristic of the architectural style or not.

In summary, we propose the following "algorithm" for the elicitation of criteria for architectural styles: First, select an existing instance of the style that is typical and not too simple. Next, set up a formal specification for the instance. This exhibits the relevant design decisions. Now analyze the formal specification to extract those features that are characteristic of the given style. Finally, explore the alternatives that would be possible and reasonable for other instances of the style and formulate these as criteria. What we lack is an overall algorithm that would apply to all styles, and tools that would help analyze the formal specification (to automate the mappings).

## 4 Related work

In this section we look at work related to components of the Bama approach. Broadly there are two related areas we have to look at: formal specification of architectural elements and reverse engineering in this context.

In the area of formal specification of architectural components, Garlan and Notkin [GN91] discuss a formal specification of event system style. However their notion of an event system is restrictive compared to ours since they focus on implicit invocation mechanisms. Garlan Allan and Ockerbloom [GAO94] characterize architectural styles as specialization of their object model. Their interest lies in generating architectural design environments from style descriptions. Ideally, a formal specification of the architectural style would let us identify the criteria. The cited works have not yet done this. We also believe that it is easier to move closer to the implementation from the criteria than from the style.

Several tools exist to provide various aspects of reverse engineering: some provide information based on the source code while others (including CIA) create a program database and permit query operations on the database. We prefer the latter since it permits modular construction of new query tools and is a proven approach dealing with very large pieces of code. In the reflexion model [MNS95] approach, a map between the high-level model and the actual code is specified by an engineer and the system shows points of agreement and disagreement. Unlike CIA, the reverse engineering technique in the reflexion model does not seem to have the ability to compute closures.

## 5 Conclusion and future work

We have discussed a bi-directional approach to modeling architectures. Our approach is significant in the sense that we try to reduce the gap between high level architecture and low level code from both sides. While we have presented an example architecture style (that of event-action), our approach will carry over to other architectural styles as well. Making design knowledge explicit via the criteria should help future implementations of the various styles.

The map from architectural criteria to formal specification has been carried out for a concrete instance of an architectural style. We plan to break down other architectural styles into criteria and see if we can replicate our results for the event-action style. Consider transaction processing style: The ACID (atomicity, consistency, isolation, durability) properties characterize current Online Transaction Processing systems. However, several special purpose transaction systems are built either as extensions to or as relaxations of the ACID properties. A set of criteria for transaction processing style could include: Atomicity, consistency, isolation, durability, choice of protocols for deciding if/when to commit, additional criteria (such as pre-release upon request), control over dynamically backing out from incorrect parts of computation (used in nested transactions), committing intermediate results while controlling resources (used in chained transactions), early commit of intermediate results at a certain level of abstraction (multilevel transactions), ability to undo part of a longer transaction (compensating transaction) etc. The above criteria were generated primarily from [Moh94, GMS87, Jim93].

We also plan to locate non-functional properties such as security and fault tolerance in the code via the RRE as well.

The mapping between formal specifications and criteria does not only help us to *analyze* a given instance of an architecture; it also helps to *design* and *change* such instances. For each criterion, location of its reflection in the formal specification is available. For a new system, the formal specification can serve as a template that can be changed when other realizations of the criteria are needed. In this situation, and in the situation where we just want to change the realization of a criterion for a given system, the mapping tells us where to look in the formal specification and what parts to change accordingly. Moreover, it is advisable to change the formal specification before attempting to change the code.

Each step provides additional input to the process due to the inherent feedback nature: presence of criteria helps create the formal specification which helps in locating the portion of the code dealing with a criterion and so on. Likewise, the existence of a formal specification of an instance helps in rounding out the criteria list.

YEAST is being rewritten in Concurrent-ML [Rep95] now and will serve as a vehicle for comparing this realization with the C version along the architectural style criteria lines discussed in this paper. For example, co-occurrence of events criterion is likely to be handled in the concurrent version.

## Acknowledgments

We thank Emden Gansner for many useful discussions.

## References

[BCK95] Architectural Styles and Services: An Industrial Experiment, July 1995. David Belanger, Yih-Farn Chen, Balachander Krishnamurthy and K. P. Vo, Submitted to ICSE-18.

[CNR90] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transaction on Software Engineering*, 16(3):325–334, March 1990.

[Dag95] Summary of the Dagstuhl Seminar on Software Architecture (#9508), David Garlan, Frances Paulisch, Walter Tichy (editors), February 20-24 1995.

[Dav95] David Rosenblum. Self-checking programs and program instrumentation. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 5. John Wiley & Sons, New York, 1995.

[Fow95] Glenn Fowler. Configuration management. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 3. John Wiley & Sons, New York, 1995.

[GAO94] David Garlan, Robert Allan, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of ACM SIGSOFT '94 Symposium on Foundations of Software Engineering Design*, pages 175–188, Dec 1994.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A Technique for Drawing Directed Graphs. *IEEE Transaction on Software Engineering*, 19(3):214–230, March 1993.

[GMS87] Hector Garcia-Molina and Ken Salem. SAGAS. In U. Dayal and I. Traiger, editors, *ACM SIGMOD 1987 Annual Conference*, New York NY, May 1987. ACM Press. Published as special issue of *SIGMOD Record*, 16(3):249-259.

[GN91] David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of VDM'91: Formal Software Development Methods.* Springer-Verlag, October 1991. Published as *Lecture Notes in Computer Science* 551.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[HK95] YEAST – A Formal Specification Case Study in Z, July 1995. Maritta Heisel and Balachander Krishnamurthy, Submitted to IWSSD—8.

[Jim93] Jim Gray and Andreas Reuter. Transaction models. In *Transaction Processing: Concepts and Techniques*, chapter 4. Morgan Kaufmann, 1993.

[KN94] Eleftherios Koutsofios and Stephen C. North. Applications of Graph Visualization. In *Proceedings of Graphics Interface 1994 Conference*, pages 235–245, Banff, Canada, May 1994.

[KR95] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 1995. To appear.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models, March 1995. University of Washington Department of Computer Science and Engineering Technical Report 95-03-02.

[Moh94] C. Mohan. Advanced transaction models—survey and critique, May 1994. Tutorial presented at ACM SIGMOD 94.

[Rep95] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1995. To appear.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[Yih95] Yih-Farn Chen, Eleftherios Koutsofios, David Rosenblum and Kiem-Phong Vo. Intertool connections. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 11. John Wiley & Sons, New York, 1995.