REVERSE ENGINEERING OF DISTRIBUTED SYSTEM ARCHITECTURE – A CASE STUDY

Nabor C. Mendonça
nabor@unifor.br

Jeff Kramer
jk@doc.ic.ac.uk

Resumo

O entendimento efetivo de um sistema de software, muitas vezes, exige que informações sobre sua arquitetura sejam extraídas diretamente de seus artefatos de código. Este processo, conhecido como *engenharia reversa*, é particularmente desafiador para sistemas distribuídos, uma vez que suas arquiteturas são especificadas em termos de abstrações existentes apenas em tempo de execução e cuja implementação tende a ser negligenciada pela maioria das tecnologias de engenharia reversa existentes. Este artigo descreve os resultados de um estudo de caso onde um novo ambiente de engenharia reversa, chamado X-ray, foi utilizado com sucesso para ajudar a recuperar abstrações de arquitetura a partir do código fonte de um sistema distribuído, baseado no modelo cliente-servidor. Estes resultados, juntamente com os resultados obtidos em outros estudos de caso envolvendo sistemas distribuídos de diferentes tamanhos e domínios de aplicação, mostram que o ambiente X-ray pode ser uma alternativa efetiva e de baixo custo com relação às soluções propostas na literatura.

Palavras-chave: *sistema distribuído, arquitetura de software, engenharia reversa.*

Abstract

To effectively reason about an existing software system it is often necessary to reverse engineering architectural information about the system from its implemented software artifacts. This process is particularly challenging for distributed systems since their architecture are specified in terms of runtime abstractions whose implementation is notoriously difficult to detect using existing reverse engineering technologies. This paper reports on a case study in which a new reverse engineering environment, called X-ray, was successfully applied to help extract architectural runtime abstractions from the source code of an existing client-server distributed system. The results of this and other similar case studies, involving distributed systems of varying size and application domains, indicate that X-ray can be a cost-effective alternative to other approaches proposed in the literature.

Keywords: *distributed system, software architecture, reverse engineering.*

1 Introduction

Software architecture is now widely recognized as a crucial aspect of software design that affects the entire software life-cycle (BASS, CLEMENTS and KAZMAN, 1998; HOFMEISTER, NORD and SONI, 2000). However, because software systems are seldom documented properly, to effectively maintain, reengineer or reuse parts of an existing system it is often necessary to recover up-to-date architectural information from the system's implemented artifacts. As it is common with most reverse engineering processes, this can be very demanding, especially if the system's original developers are no longer available.

Current work on architecture recovery aims at providing reverse engineering techniques and tools to support software engineers in extracting high-level design abstractions from an existing system implementation. Technically, this process can

be seen as an attempt to reverse the process that might have been used to derive a system's implemented artifacts from a suitable architectural specification. In practice, though, architecture recovery is challenged primarily by the fact that most traditional programming languages provide only limited support to implement architectural concepts (SHAW and GARLAN, 1996). In particular, those languages provide no primitive construct to implement *architectural runtime abstractions*, such as servers, clients, and message-based interaction protocols, which are typical to the design of distributed software systems. As a consequence, recognizing those abstractions in a distributed system implementation can be exceedingly difficult if based on plain source code information alone.

Most of the available reverse engineering tools are limited in supporting recovery of distributed architectures because they tend to neglect aspects of component distribution and runtime organization. The exception are dynamic analysis based tools (e.g. KAZMAN and CARRIÈRE, 1998) and domain-oriented pattern matching tools (e.g. FIUTEM et al., 1996; HARRIS, YEH and REUBENSTEIN, 1996). Dynamic analysis tools are effective in capturing architectural components and interactions at runtime, but may be insufficient to reveal how a runtime element relates to, or is realized in terms of, the various elements of the source code. In addition, those tools tend to require costly event-tracing techniques, such as code instrumentation and symbolic execution, which may be impractical for embedded and real-time systems (BRATTHALL and RUNESON, 1999). Domain-oriented pattern matching tools in turn trade off accurate runtime information against a better mapping of potential runtime events to their implementing code fragments. However, those tools are still limited in revealing how the recognized code fragments may be used across the implementation of different executable components.

This paper describes a new architecture recovery environment for distributed systems, called X-ray, and reports on the results a case study in which this environment was successfully used to recover the runtime architecture of Samba, a well-known file and printer sharing open source distributed system (TRIDGELL, 1994). X-ray integrates three reverse engineering techniques, namely *component module classification*, *syntactic pattern matching*, and *structural reachability analysis*. Used complementary, these three techniques can help to identify which executable components might be implemented in a given distributed system source code, and also through which mechanisms those components are expected to interact at runtime. The results of the Samba case study, along with the results of similar experiments involving other distributed systems, build confidence that these three techniques can be of great value to help form a static approximation of a distributed system's potential runtime organization. The fact that X-ray is primarily based on static source code information makes it a cost-effective alternative to other more demanding approaches based on dynamic analysis. Alternatively, X-ray can be used as an early source of architectural expectations about a distributed system, before one goes through the burden of instrumenting the code, generating input data, and running the execution scenarios selected for dynamic analysis. A static approach may also be the only viable solution to architecturally reasoning about legacy distributed system implementations whose original development and/or execution platform is no longer available.

The rest of the paper is organized as follows. Section 2 introduces X-ray. Section 3 describes the method and the detailed results of the case study involving Samba. Section 4 evaluates the merits of X-ray by discussing its main features and limitations in light of the case study results. Section 5 covers related work. Finally, section 6 concludes the paper by summarizing the research and suggesting topics for future work.

2 The X-ray Environment

As mentioned before, X-ray comprises three reverse engineering techniques. The main concepts and tools underlying each of these techniques are described in the following subsections.

2.1 Component Module Classification

Because the code is the primary—and often the only—locus of maintenance, traditional source code artifacts, such as configuration files and directory structures, may not be the most reliable source of information on the implementation of distributed system components. To avoid these limitations, X-ray relies on the component module classification technique. This technique identifies which source code files (or modules) constitute the implementation of each distributed system (executable) component by automatically classifying them according to the way they are used by those components. The technique is based on the concept of entry and library modules, and on the analysis of module dependency information statically extracted from source code.

2.1.1 Entry and Library Modules

In most programming languages, the code for an executable component comprises an execution *entry module* and, optionally, one or more *library modules*. An execution entry module is a module containing a language-designated *execution entry point*. In Java, for example, an execution entry point corresponds to a main class method (in the case of Java applications) or a class constructor derived from one of the constructors of the Applet class (in the case of Java applets). Library modules in turn are all modules that do not contain an execution entry point. In contrast to execution entry modules, library modules may be reused across different executable programs. In X-ray, identifying component entry modules and the library modules they depend upon is key to recognizing what, how many, and where executable components are implemented in the source code of an unfamiliar distributed system. The module classification technique identifies component entry modules and their dependencies by means of the *module dependency graph* system model.

2.1.2 Module Dependency Graph

A module dependency graph ("MDG") is an abstract graph model of the source code modules of a distributed system and their "depends-on" relationships. In an MDG, each node corresponds to a distinct source code module. A directed edge between two nodes, say p and q, represents the fact that module p depends on (i.e., uses resources defined and exported by) module q.

It is important to emphasize that an MDG only captures implementation dependencies between modules that are part of the system source. Information on the use of external header files and libraries is completely omitted from the graph. The advantage of filtering out this kind of information is that it reduces the library saturation problem during model inspection and visualization.

Two important relations over MDG nodes, that are useful to reasoning about a number of interesting MDG properties, are the relations of *reachability* and *dominance* (HECHT, 1977). Let x and y be two (not necessarily distinct) modules in an MDG. Module x is said to "reach" module y if and only if there is a path in the MDG connecting x to y. On the other hand, x is said to "dominate" y if and only if every path connecting the root of MDG to y contains x. The reachability relation captures the notion of *transitive dependency* among modules: if p reaches q, then p depends directly or indirectly on q. The dominance relation in turn captures the notion of *exclusive transitive dependency* among modules: if p dominates q, then every module not dominated by p that depends on q does so only indirectly via p.

The reachability and dominance relations are well-known and have been used to reason about a variety of flow-graph based program models (CIMITILE and VISAGGIO, 1995; BURD and MUNRO, 1999).

2.1.3 Module Classification

The reachability and dominance relations provide a basis for the classification of the source code modules of a distributed system according to three categories: *component relevant modules*, *component exclusive modules*, and *shared modules*. Let p be a module in an MDG. p is classified as a *relevant module* of a component if and only if the execution entry module of that component reaches p in the MDG. In addition, if the component's entry module dominates p in the MDG, then p is also classified as an *exclusive module* of that component. On the other hand, if p is not dominated by any of the component entry modules in the MDG, which means that p is reachable from at least two different entry modules, then p is classified as a *shared module* with respect to the source code as whole.

In X-ray, module classification serves two main purposes. First, it explicitly distinguishes which parts of the source code (and the functionalities implemented therein) are unique to certain components and which are shared by multiple components. This knowledge is important in that it helps to understand how components differ in terms of size, number of implemented functionalities, overall complexity, etc. Second, it guides the application of the other two X-ray techniques and helps to interpret their results. For example, if the implementation for a certain architectural feature is recognized in a single module of the source, and that module has been classified as being exclusive to a certain component, then it is safe to say that this component is likely to be the only component in the system that implements that feature.

2.2 Syntactic Pattern Matching

Module classification alone is insufficient to reveal the architectural roles that each identified executable component may play at runtime (which components are clients and which are servers?), nor does it say anything regarding how those

components are expected to interact at runtime (which components interact with which other components? Do they interact via sockets, RPC, or otherwise?). Recovering this kind of architectural abstractions from source code is not a trivial task, as it depends on external domain knowledge on both the architectural style and the development platform of the system (HOLTZBLATT et al., 1997).

The syntactic pattern matching technique of X-ray is aimed at facilitating the recognition of code constructs that may implement runtime interaction features. The technique comprises a notation for the definition of syntactic *program patterns*, and an associated pattern-matching mechanism for automatic search of program patterns in a syntactic source code representation. The underlying idea is to use program patterns as the means to represent the stereotyped implementation of typical interaction features under a certain platform of interest. In this way, the interaction features of a distributed system, developed under that platform, could be automatically recognized by matching the appropriate program patterns against an abstract representation of the system's source.

2.2.1 Defining Program Patterns

A program pattern is an abstract representation embodying program knowledge at the lexical, syntactic or algorithmic levels (PAUL and PRAKASH, 1994; GRISWOLD, ATKINSON and MCCURDY, 1996). A program pattern can be used to represent a wide range of program concepts, from simple programming constructs and techniques (e.g., subroutine calls, variable swaps, iteration loops, and recursion), to more general algorithms (e.g., sort and search). X-ray allows the definition of syntactic program patterns through a library of special Prolog predicates. These predicates can be used to both specify and execute a variety of matching operations on an abstract syntax tree ("AST") representation of a program. An X-ray program pattern is then any valid Prolog clause defined in terms of one or more of the special predicates provided. Predicates used specifically to match AST nodes and node attributes constitute the fixed parts of a pattern, while their arguments constitute the pattern's varying parts. The varying parts can be further constrained by expressing *equivalence* and *structural* relations among them. Equivalence relations can be enforced by specifying arguments with the same Prolog name. Structural relations in turn are enforced by means of *structural predicates* that match AST nodes according to their relative position within the overall AST structure.

To illustrate how a program pattern can be defined using this notation, consider the following Prolog predicate:

```
ast_assign(AssignId,Lhs,Rhs,(File,Line))
```

This predicate succeeds if it matches a node of type *assignment expression* in a depth-first visit of the AST. Its four arguments match, respectively, the identifier of the main node (AssignId), the node identifiers for the assignment's left-hand-side ("LHS") and right-hand-side ("RHS") expressions (Lhs and Rhs), and the source code location of the assignment in the form of a (File,Line) pair.

Here is another example:

```
ast_call(CallId,CallName,ParList,Loc)
```

This predicate succeeds if it matches an AST node of type *subroutine-call*, with its four arguments matching the identifier of the matched node (CallId), a string with the callee's name (CallName), a list with the node identifiers of each parameter expression of the call (ParList), and the source code location pair of the call (Loc).

The following composite clause illustrates how more elaborate patterns can be defined taking advantage of Prolog's standard library predicates:

```
member(File, ["init.c", "lib.c"]),
ast_assign(AssignId,LHS,RHS,(File,AssignLine)),
ast_call(CallId,CallName,ParList,(File,CallLine)),
ast_parent(RHS,CallId)
```

This clause (or pattern) succeeds if it matches an assignment and a subroutine-call node pair, with the equivalence constraint (expressed through a common predicate argument instantiated via the standard list library predicate of Prolog, member/2) that both nodes must occur in AST branches generated from the same file. Note that the list ["init.c","lib.c"], passed as an argument to the member/2 predicate, restricts the name of the file to be either init.c or lib.c. In addition, the

pattern also includes the structural constraint (expressed by means of the notation's structural predicate ast_parent/2) that the node corresponding to the assignment' RHS expression must be the parent, in the AST, of the node corresponding to the subroutine call. In other words, the above pattern will match any assignment construct, in either init.c or lib.c, whose RHS expression (perhaps an arithmetical or type casting expression) contains a subroutine call.

2.2.2 Socket Creation Patterns

This section illustrates how the Prolog-based pattern notation of X-ray can be used to represent the C implementation of a *socket*, an inter-process communication mechanism typically used to implement client/server interactions in distributed software applications (STEVENS, 1990). In this example, two program patterns are used to capture socket creation at both the server and client sides. The two patterns are defined using the same set of arguments: an expression corresponding to the matched socket descriptor; an expression list with the socket's domain, type, protocol and address parameters; and the location of the match in the source code. The patterns for socket creation at the server and client sides, respectively, are shown in Fig. 1. Note how the client-side pattern uses the auxiliary predicate connection_call/2 to allow matches containing either connect() or sendto() as the main socket connection routine.

Using these patterns, three conditions must be satisfied for one to be able to recognize a potential implementation for a socket-based interaction between two distributed system components. First, each pattern has to be matched in a section of code associated with a distinct executable component. Second, sockets created by the two matching code fragments must be of compatible type, domain, and protocol. Finally, the values of the server address parameters (particularly its port number field) used at the client side must refer to the corresponding parameter values defined at the server side. For systems developed following a traditional single-server/multiple-client style, it is possible to recognize potential socket-based interactions even when the server address parameters used to establish the connections are not statically defined in the code. This is because all clients in a single-server/multiple-client system are expected to communicate with the same server component at runtime. For systems of the multiple-server/multiple-client style this would not be possible, though, as the only way to determine which server a client is expected to communicate with would be via analysis of the server address parameters defined at both sides.

```
server_socket(SId,[Dm,Tp,Pr,Srv],(F,[L1,L2])) :-
   % get socket descriptor
   ast_call(SCall,"socket",[Dm,Tp,Pr],(F,L1)),
   ast_assign(_,SId,SCall,(F,_)),
   % socket binding
   ast_call(BCall,"bind",[BId,Srv,_],(F,L2)),
   % constraints
   ast_before(SCall,BCall),
   ast_sameident(SId,BId).

client_socket(SId,[Dm,Tp,Pr,Srv],(F,[L1,L2])) :-
   % socket creation
   ast_call(SCall,"socket",[Dm,Tp,Pr],(F,L1)),
   ast_assign(_,SId,SCall,(F,_)),
   % generic connection call
   connection_call(CCall,CId,Srv,(F,L2)),
   % constraints
   ast_before(SCall,CCall),
   ast_sameident(Id,CId).

connection_call(CCall,SId,Srv,Loc) :-
   % connection via connect() - stream sockets
   ast_call(CCall,"connect",[SId,Srv,_],Loc).
connection_call(CCall,SId,Srv,Loc) :-
   % connection via sendto() - datagram sockets
   ast_call(CCall,"sendto",[SId,_,_,_,Srv,_],Loc).
```

Figure 1 - Socket creation patterns.

Similar patterns can be defined, with varying degrees of precision and expressiveness, for other types of component interaction mechanisms, such as process invocation, event-notification, and pipe-based communication (STEVENS, 1990).

2.3 Structural Reachability Analysis

The encapsulation of architectural features in shared modules is a common practice in the development of many distributed systems. However, since shared modules may be used differently by different executable components, it is often difficult to determine which executable components a syntactically recognized architectural feature might be associated with. The structural reachability analysis technique of X-ray is aimed at facilitating this task. The technique computes transitive closures over the activation relations between program units so as to determine which components may "reach" the implementation of a particular architectural feature in an *activation graph* system model.

An activation graph (AG) is an abstract graph model of the well-defined *activation units* (procedures, functions, methods, etc.) of a software system and the "activates" (or "calls") relationships between them. In an AG, each node corresponds to a distinct activation unit. A directed edge between two nodes, say p and q, represents the fact that p directly activates q. The relations of reachability and dominance over AG nodes can be defined exactly as those for an MDG. An executable component is then said to "reach" a syntactically recognized architectural feature if its execution entry point reaches (i.e., directly or indirectly activates) that feature's containing activation unit in the distributed system's AG. In the case of distributed system written in C, for example, an executable component is said to reach a server-side socket creation feature if its main() function reaches a function which invokes both the socket() and bind() socket library routines. It is important to emphasize that this notion of feature reachability is language and platform dependent. This is because the resources needed to implement a particular architectural feature may vary across different operating systems and/or development environments.

2.4 Tool Support

To provide automated support for X-ray, several types of reverse engineering tools were either implemented from scratch or reused "off-the-shelf". These include extraction tools (which are used to build the necessary source code models from a given distributed system implementation), source analysis tools (which implement the three techniques themselves), and visualization tools (which are used for automatic visualization of the results).

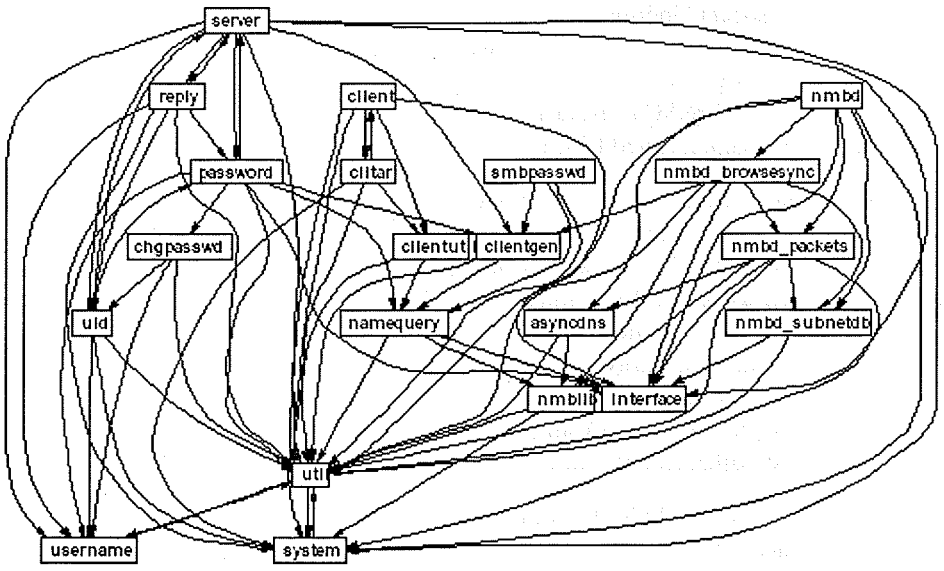

Figure 2 - Samba's MDG as originally extracted from the source code.

Currently, the X-ray tool set is targeted for distributed systems developed under the C/Unix domain. This is justified since C/Unix has long been the platform of choice for the development of a large number of distributed software applications, many of them now in a legacy state. From the research point of view, C/Unix was also attractive since it offered a rich set of runtime interaction mechanisms to experiment with.

3 Samba – A Case Study

Samba is a suit of C programs that implement the SMB protocol for Unix systems (TRIDGELL, 1994). The SMB protocol allows sharing of file and print services over a local area network. Operating systems that support SMB natively include Windows, OS/2, and Linux. The Samba distribution package contains the source code for a number of components: a SMB file and printing server, a NetBIOS name server, an ftp-like Unix SMB client, as well as several other utility programs (Samba version 1.9.18p8 was used in this case study). Altogether, the distribution contains 139 C files, comprising over 78,000 non-blank lines of code. Interestingly, the system configuration file provides an informal yet clear distinction between files used exclusively to build a single executable component and files that may be used to build two or more components. Therefore, using Samba as a case study brought about an interesting opportunity to correlate existing source code and configuration file information with the types of architectural abstractions that can be recovered with X-ray.

3.1 Method

The case study was carried out in four subsequent stages. In the first stage, a set of extraction tools was used to build, from the Samba source, the three types of source models (i.e., MDG, AST, and AG) used by each respective X-ray technique. In the second stage, the module classification technique was used to identify Samba executable component modules, and to distinguish between component exclusive modules and shared modules. In the third stage, the syntactic pattern matching technique was used to search for potential Samba implementations of several interaction features typically found in the C/ Unix domain. Finally, in the fourth stage, the structural reachability analysis technique was used to help assign interaction features recognized in shared modules to each individual Samba component.

Due to space restrictions, this paper focuses on a reduced yet representative subset of the Samba source. However, to certify that the results presented are of value, the output produced by each technique was analyzed with respect to other sources of evidence, such as additional system information gathered through manual examination of the code and the available documentation, and also through informal consultation with Samba's chief developer. The components selected for presentation are listed in Tab. (1). Also listed in the table are the components' executable name, entry module, and size in both number of modules (#Mod) and number of non-blank lines of code (#LoC).

Table 1 - Selected Samba components and their characteristics.

Component Name	Executable	Entry Module	#Mod	#LoC
SMB server	Smbd	server	12	22.251
SMC client	Smbclient	client	10	17.536
NetBIOS name server	Nmbd	nmbd	12	15.062
Password utility	smbpasswd	smbpasswd	8	12.230
Total			21	32.105

3.2 Results

3.2.1 Component Module Classification

Figure 2 shows the Samba MDG as it was originally extracted from the source code. Figure 3 in turn shows the same MDG after processing by the module classification technique. In the latter, component exclusive modules are shown clustered together with their respective entry modules (highlighted in grey). Modules shown unclustered (such as clientgen and util) are shared by at least two components.

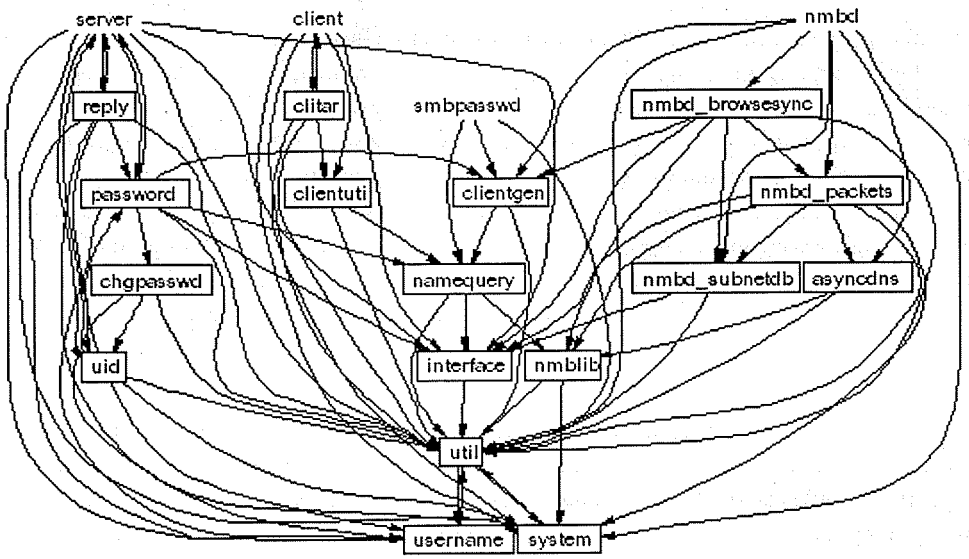

Figure 3 - Samba's MDG after processing by the module classification tool.

Figure 3 offers a number of interesting insights into how runtime components may be implemented in the Samba source. For example, although the set of exclusive modules identified for some of the components show a few name similarities (for example, all modules exclusive to the NetBIOS name server contain the prefix nmbd, no rigid naming convention regarding component implementation can be observed for the source code as a whole (for example, the modules exclusive to the SMB server, of entry module server, show no name similarity at all). Also, in at least one situation the informal module classification provided by the system configuration file was found to be inconsistent with the source code evidence. This is the case of the clientgen module, which was identified as a shared module by the module classification tool, but is incorrectly listed among component exclusive modules in the configuration file. Finally, the fact that only a single exclusive module was associated with the Password utility, of entry module smbpasswd, is indicative that this component may implement a relatively simple set of functionalities.

3.2.2 Syntactic Pattern Matching

Program patterns representing several types of runtime interaction mechanisms were successfully matched against the AST model extracted from the Samba source. These include patterns for *process spawning*, *process overlay invocation*, *shell invocation*, *pipe-based communication*, and *socket creation*. The modules where those patterns were matched, along with the name of the Samba component to which they were exclusively associated, if applicable, are described in Tab. (2). Note that three of the patterns (one for process spawning and two for socket creation) could not be immediately associated with any component in particular since they were matched in a shared module (util).

Table 2 - Samba's recognized runtime interaction patterns.

Interaction Mechanism	Module	Component
Process spawning	server	SMB server
	uid	SBM server
	asyncdns	NetBIOS name server
	util	N. A. (shared)
Overlay invocation	uid	SMB server
Shell invocation	client	SMB client
Pipe creation	asyncdns	NetBIOS name server
Socket creation (server)	util	N. A. (shared)
Socket creation (client)	util	N. A. (shared)

3.2.3 Structural Reachability Analysis

In an attempt to identify which Samba components might actually invoke those patterns recognized in a shared module, the patterns' containing C functions were subjected to the structural reachability analysis tool. The results were as follows. The process-spawning pattern was found to be reachable both from the SMB server and the NetBIOS name server. This makes sense since the two components are described as *daemon* processes in the system documentation, and, as expected, both use this pattern to detach from their controlling terminal after invocation. The server-side socket creation pattern was found to be reachable from each of the four Samba components considered in this case study. This result was somewhat of a surprise since there are only two components described in the Samba documentation as playing the role of servers. A manual analysis of the code involved in the (indirect) invocation of the pattern helped to clarify this issue. The NetBIOS name server seems to use the pattern to create UDP-type sockets whose port numbers default to either 137 or 138. The SMB server in turn uses it to create TPC-type sockets whose port number defaults to 139. In addition, the SMB server, the SMB client and the password utility all use this same pattern to create UDP-type sockets whose port number is automatically assigned by the underlying operating system.

A similar analysis was then performed with the code involved in the invocation of the client-side socket creation fragment. By matching port number values for each compatible pair of sockets recognized in the source code of the four Samba components, it was possible to identify which components rely on sockets to interact at runtime and through which type of socket (i.e., either TCP or UDP). These results offered enough evidence to convey a number of interesting insights into Samba's potential runtime organization. For example, it became clear that both smbclient and smbpasswd are potential clients of the two Samba servers, namely smbd and nmbd. Moreover, it was also evident that both servers are potential clients of each other.

4 Discussion

Even though X-ray has been successfully applied to recover runtime abstractions from the source code of "real-world" distributed systems, its current scope and approximate nature still limit its application within a broader context. Some of these issues are briefly discussed below.

Understanding the functionality of a distributed system component requires a detailed knowledge of the component's implementation as well as the environment within which the component will be executed. Clearly, the kinds of architectural abstractions extracted with the help of X-ray are insufficient to reveal the full set of functionalities that might be implemented by a particular runtime component. For example, in the case of Samba, identifying how the SMB server may be implemented in terms of source code modules is far from revealing how it is expected to behave at runtime. On the other hand, by explicitly distinguishing component exclusive modules from shared modules, and by identifying which code fragments might implement important runtime interaction mechanisms, X-ray help maintainers in focusing their understanding efforts only on those regions of the code considered more relevant to the maintenance task at hand.

Another issue concerns that fact that most distributed systems are designed in a way to support dynamic runtime organizations. This is especially true for systems that follow a client-server style, such as Samba, in which both the number of runtime components, and the patterns of interaction between those components, may vary during system execution. X-ray is limited with this respect since it only reveals a static approximation of the possible architectural organizations that a distributed system may undergo at runtime.

5 Related Work

There are number of other reverse engineering approaches that, to a greater or lesser extent, can be used to support architecture recovery. Some of the most representative tools and technologies are discussed below. When pertinent, they are also compared or contrasted with X-ray.

5.1 Source Model Query Tools

Tools of this category extract static source models of interest into a source code database, and then allow users to interactively abstract from those models using the database's query and visualization facilities. Typical examples include CIA (CHEN, NASHIMOTO and RAMAMOORTHY, 1990), Acacia (CHEN, GANSNER and KOUTSOFIOS, 1998), and

the Software Bookshelf (FINNIGAN et al., 1997). Conformance checking tools (MURPHY, NOTKIN and SULLIVAN, 1995; SEFIKA, SANE and CAMPBELL, 1996) use source model queries in a different way. The user provides those tools with an idealized high-level model, along with a set of rules mapping the elements of that model to source code entities. Those tools then check whether or not the given model complies with the source code evidence. In X-ray, source model queries provide the basis for its three reverse engineering techniques. They are applied to three different source models, at different levels of abstraction, and are formulated considering domain knowledge specific to how runtime components and interaction mechanisms are typically implemented in a distributed system source code.

5.2 Clustering Tools

Clustering tools attempt to reconstruct a system's implemented design by automatically or semi-automatically clustering source code elements into a hierarchy of logical design entities (MULLER et al., 1993; MANCORIDIS et al., 1998; KAZMAN and CARRIÈRE, 1999). Clustering tools are limited to support architecture recovery because they tend to recognize only interaction mechanisms that are explicitly represented in the code (such as definition/use relationships between program entities). In X-ray, clustering is used not as the final product of the architecture recovery process, but rather as a means through which to identify and classify executable component modules. The resulting module classification is then complemented with, and provides a basis for, the application of more sophisticate static analysis techniques.

5.3 Pattern-matching Tools

Two pattern-matching based tools, namely ManSART (HARRIS, YEH and REUBENSTEIN, 1996) and ART (FIUTEM et al., 1996), are more closely related to X-ray. Both rely on the definition of a generic architectural model which role is to capture important domain and language knowledge pertaining the implementation of several *architectural styles* (SHAW and GARLAN, 1996), such as *client-server*, *pipe-and-filter*, and *task-spawning*. A set of predefined syntactic queries is then used to recognize potential implementations of style-specific architectural elements in the code. The syntactic pattern matching technique of X-ray follows a similar approach. However, since X-ray does not define an architectural model explicitly, styles elements are only indirectly represented in the form of the several types of runtime interaction mechanisms described via program patterns. For example, X-ray only captures elements of the client-server style indirectly, in the form of socket-creation program patterns. Another salient characteristic of X-ray is that it not only identifies but also classifies component modules based on module dependence information automatically extracted from the source code itself. ManSART and ART, in contrast, have to rely on (possibly inaccurate) build dependencies gathered from manual examination of configuration files.

6 Conclusion

This paper described a new architecture recovery approach for distributed systems, called X-ray, and reported on the results of a case study in which the approach's support tools were successfully used to recover architectural runtime abstractions from the source code of Samba, a client-server distributed software system. These results, along with the results of other similarly successful experiments reported elsewhere (MENDONÇA and KRAMER, 2001) help to build confidence that combining multiple static source models and reverse engineering techniques, in a complementary way, as advocated by X-ray, can be a practical means of reasoning about a distributed system's "as-built" architectural design. In addition, the fact that the case study was based on a well-known open source system may contribute to encourage other architecture recovery researchers and tool developers to perform similar experiments, thus fostering further developments in this important research area.

A promising application of X-ray would be to recover runtime abstractions from multiple versions of the same distributed system. By comparing and contrasting the abstractions recovered from each version, it might be possible, for example, to assess the extent to which the system maintains or degrades its conceptual integrity over time. Another interesting topic for future research concerns investigating how to extend X-ray so that it can be applied to modern Web and agent based distributed software architectures.

References

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software architecture in practice*. Reading: Addison-Wesley, 1998. 452p.

BRATTHALL, L.; RUNESON, P. Architecture design recovery of a family of embedded software systems. In: IFIP WORKING CONFERENCE ON SOFTWARE ARCHITECTURE, 1., 1999, San Antonio. *Proceedings*... San Antonio: Kluwer Academic Publishers, 1999. p. 3-14.

BURD, E. L.; MUNRO, M. Evaluating the use of dominance trees for C and COBOL. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 1999, Oxford. *Proceedings*... Oxford: IEEE. 1999. p. 401-410.

CHEN, Y. F.; NASHIMOTO, M. Y.; RAMAMOORTHY, C. U. The C information abstraction system. *IEEE Trans. Software Engineering*, Los Alamitos, v. 16, n. 3, p. 325-334, Mar. 1990.

CHEN, Y. F.; GANSNER, E. R.; KOUTSOFIOS, E. A C++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Software Engineering,* Los Alamitos, v. 24, n. 9, p. 682-694, Sept. 1998.

CIMITILE, A.; VISAGGIO, G. Software salvaging and the call dominance tree. *J. of Systems and Software,* Amsterdam, v. 28, n. 2, p. 117-127, Feb. 1995.

FINNIGAN, P. et al. The software bookshelf. *IBM Systems Journal,* Yorktown Herghts, v. 36, n. 4, p. 564-593, Nov. 1997.

FIUTEM, R. et al. A cliché-based environment to support architectural reverse engineering. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 1996, Monterey. *Proceedings*... Monterey: IEEE. 1996. p. 319-328.

GRISWOLD, W. G.; ATKINSON, D. C.; MCCURDY, C. Fast, flexible syntactic pattern matching and processing. In: IEEE INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 4., 1996, Berlin. *Proceedings*... Berlin: IEEE. 1996. p. 144-153.

HARRIS, D. R.; YEH, A. S.; REUBENSTEIN, H. B. Extracting architectural features from source code. *Automated Software Engineering Journal,* Dordrecht, v. 3, n. 2, p. 109-138, Apr. 1996.

HECHT, M. S. *Flow analysis of computer programs*. Amsterdam: Elsevier North-Holland Inc. 1977. 232p.

HOFMEISTER, C. R.; NORD, L.; SONI, D. *Applied software architecture*. Reading: Addison-Wesley. 2000. 397p.

HOLTZBLATT, L. J. et al. Design recovery for distributed systems. *IEEE Trans. Software Engineering,* Los Alamitos, v. 23, n. 7, p. 461-472, July 1997.

KAZMAN, R.; CARRIÊRE, S. J. View extraction and view fusion in architectural understanding. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 5., 1998, Victoria. *Proceedings*... Victoria: IEEE CS Press. 1998. p. 290-299.

KAZMAN, R.; CARRIÊRE, S. J. Playing detective: reconstructing software architecture from available evidence. *Automated Software Engineering Journal,* Dordrecht, v. 6, n. 2, p. 107-138, Apr. 1999.

MENDONÇA, N.; KRAMER, J. An approach for recovering distributed system architectures. *Automated Software Engineering Journal,* Dordrecht, v. 8, n. 3, p. 311-354, July 2001.

MÜLLER, H. A. et al. A reverse-engineering approach to subsystem structure identification. *J. of Software Maintenance: Research and Practice,* West Sussex, v. 5, n. 4, p. 181-204, Dec. 1993.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: bridging the gap between source and higher-level models. In: SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 3., 1995, Washington. *Proceedings*... Washington: ACM SIGSOFT. 1995. p. 18-28.

PAUL, S.; PRAKASH, A. A framiwork for source code search using program patterns. *IEEE Trans. Software Engineering,* Los Alamitos, v. 20, n. 3, p. 463-475, Mar. 1994.

SANCORIDIS, S. et al. Using automatic clustering to produce high-level system organizations of sources code. In: IEEE INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 6., 1998, Ischia. *Proceedings*... Ischia: IEEE. 1998. p. 45-52.

SEFIKA, M.; SANE, A.; CAMPBELL, R. H. Monitoring compliance of a software system with its high-level design models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 18., 1996, Berlin. *Proceedings...* Berlin: ACM Press. 1996. p. 387-396.

SHAW, M.; GARLAN, D. *Software architecture: perspectives on an emerging discipline.* Upper Saddle River: Prentice-Hall. 1996. 242p.

STEVENS, W. R. *Unix network programming.* Upper Saddle River: Prentice-Hall. 1990. 768p.

TRIDGELL, A. SAMBA: Unix talking with PC's. *Linux Journal,* Seattle, v. 4, n. 7, Nov. 1994. Available in: <http:// linuxjournal. com>. Accessed in: 21 mar. 2003.

ABOUT THE AUTHORS

Nabor C. Mendonça
B.Sc. in Data Processing, Federal University of Amazon, Brazil, 1991; M.Sc. in Computer Science, State University of Campinas, Brazil, 1993; Ph.D. in Computing, Imperial College of Science, Technology and Medicine, University of London, U.K., 1999. Currently works as a full professor at the Center for Technological Sciences, University of Fortaleza, Brazil. His research interests include distributed systems, reverse engineering, and software engineering for Web-based and mobile applications.

Jeff Kramer
B.Sc. (Eng.) in Electrical Engineering, University of Natal, South Africa, 1971; M.Sc. in Computing Science, Imperial College of Science, Technology and Medicine, University of London, U.K, 1972; Ph.D. in Computing Science, Imperial College of Science, Technology and Medicine, University of London, U.K., 1979. Currently is Head of the Department of Computing at Imperial College. His research interests include behaviour analysis, the use of models in requirements elaboration and architectural approaches to self-organising software systems. Professor Jeff Kramer is a Chartered Engineer, Fellow of the IEE and Fellow of the ACM.