A Cliché-Based Environment to Support Architectural Reverse Engineering

R. Fiutem, P.Tonella, G. Antoniol
IRST
I-38050 Povo (Trento), Italy
{fiutem,tonella,antoniol}@irst.itc.it

E. Merlo
Dep. Electrical and Computer Engineering
Ecole Polytechnique
C.P. 6079, Montreal, Quebec, Canada
merlo@rgl.polymtl.ca

Abstract

When programmers perform maintenance tasks, program understanding is required. One of the first activities in understanding a software system is identifying its subsystems and their relations, i.e. its software architecture. Since a large part of the effort is spent in creating a mental model of the system under study, tools can help maintainers in managing the evolution of legacy systems, by showing them architectural information.

In this paper, an environment for the architectural analysis of software systems is described. The environment is based on a hierarchical architectural model that drives the application of a set of recognizers, each producing a different architectural view of the system or of some of its parts. Recognizers embody knowledge about architectural clichés *and use flow analysis techniques to make their output more accurate.*

1 Introduction

Software maintenance activities usually make it necessary to understand the organization of a program at different levels of abstractions which are the entire program, a module, a procedure.

Approaching a legacy system maintainers may want to know its overall organization and high level structure, which requires to identify its subsystems, its functionalities and their interactions, i.e., its software architecture.

Other scenarios may require program understanding at the architectural level. For example, when the goal is to reuse some functionality of a system, programmers have to discover which part of a software system does a specific job, how it can be extracted from the system and what kind of dependencies and interactions it has with the rest of the system. Another motivation for architectural design recovery is to compare the architecture extracted from code with the intended design to detect discrepancy.

In the case of legacy systems, documentation is often lacking and people that originally developed the program are no longer available, so the only source of documentation about the program is the program itself. Since a large part of the effort is spent in creating a mental model of the system under study, tools can help maintainers in managing the evolution of legacy systems by showing them architectural information.

The reverse engineering research community has traditionally focused on understanding programs *in-the-small*, i.e. on trying to identify instances of *program concepts* at algorithmical and data-structure level using *plans*, which are abstract representations embodying the knowledge about program concepts, their components and constraints [1, 2].

Other approaches have investigated structural analysis and *in-the-large* re-documentation of software systems, by building tools that support methods for identifying, re-organizing and documenting layered subsystem hierarchies [3, 4].

Recently, some works that address the problem of high-level design recovery using reverse engineering technology have been presented [5, 6], integrating in a framework architectural styles representations and a library of recognizers to extract architectural information from source code.

In this paper an environment for architectural analysis of software systems is described. The environment supports the discovery of information at the architectural level and presents such information to programmers and maintainers through multiple hierarchical views. It is based on a hierarchical architectural model that specifies the components and connectors types of a software architecture at different levels of detail.

The architectural analysis environment, which is under development, is targeted to the C/Unix domain. It is built on top of the Refine/C [1] and exploits its capabilities to build and analyze Abstract Syntax Trees (ASTs).

Architectural recognizers [6] have been developed that work on ASTs to detect source code level constructs signaling the presence of architectural components or connectors, which are used to produce graphical views. To obtain more accurate results, flow analysis techniques have also been applied in many architectural recognizers.

The environment has been designed following the

[1] Refine, Refine/C, Intervista and Workbench are trademarks of Reasoning Systems Inc.

approach to architectural analysis that was presented in [19]. In this paper, the implementation of some architectural recognizers and the role of flow analysis techniques are described.

This paper is organized as follows: Section 2 illustrates the model that has been used to represent architectural information and describes the analysis environment that is based on it. In Section 3 the set of recognizers available in the environment is presented together with a detailed implementation example of a recognizer for client-server connections. Section 4 describes the user interface of our architectural analysis environment. Finally, in Section 5 some conclusions and related future work are presented.

2 Architectural analysis framework

2.1 Model

Several notations and languages have been proposed to describe software systems architecture [7, 8, 9, 10, 11]. However most of these notations have been used within forward engineering approaches, i.e. they are architectural or configuration/integration languages producing system descriptions that can be, for some of them, further processed to generate real software systems. These descriptions usually represent what is called the *idealized*[5] or *conceptual* [12] software architecture. From a reverse engineering point of view, it is difficult to recover such kind of information because it is often distant from the actual objects that are retrievable from source code. Code represents the *as-built*[5] architecture, that is the actual software organization. These two views do not necessary match and this is the bridge programmers have to mentally build while performing maintenance activity.

The model we have adopted for our architectural analysis environment starts from the *as-built* architecture and is based on partial views, each consisting of specific components and connectors, that are incrementally instantiated with the aid of the user.

These views are hierarchical so that zoom-in and zoom-out visualization is allowed. The hierarchical nature of architectural information and the need for multiple levels of representation have also been stated in [12, 13].

The highest abstraction level of our architectural model is the *system* and the corresponding *system view*. A system, $S =< P, I >$, which can also be distributed, consists of a set P of *program* components which communicate through a set I of connectors of type *inter-process-connector*(IPCs).

Programs, which constitute an application, cooperate by exchanging data and synchronizing themselves through connectors.

The IPCs that can bind program components are represented in the following taxonomy which is partially derived from [8]:

- **shared-file**: a data repository accessed sequentially by several programs;
- **shared-memory**: data that can be accessed at random and are shared among different programs;
- **remote procedure call (RPC)**: a procedure call-like mechanism between different programs;
- **stream**: a unidirectional or bidirectional, reliable and ordered connector between two programs;
- **message**: a connector that is not restricted to a fixed arity;
- **invocation**: it represents one program (process) starting another program (process).

Obviously, the system view is relevant particularly for systems consisting of concurrent communicating processes: applications consisting of a single program will be represented at the system level by a single component and no inter-process connectors. A program component can, in turn, be represented by several views. We have defined two views: the *module view* and the *task view*.

In the *module view*, a program $P =< M, U >$ is represented as a set M of components of type *module* and a set U of connectors of type *uses*. A module is an entity that provides a computational service [14] by exporting a well-defined functional or data abstraction and which often corresponds to separately compilable units such those provided in most programming languages. Connectors U represent *uses* relations and are sometimes called *depends-upon* or *implementation* relations) [14]. The module view is static and illustrates the import/export of resources among the modules of a program.

The *task view* of a program $P =< T, S >$ consists of

- a set of T components of type *task*;
- a set S of connectors of type *spawns*.

A task or thread is a subprogram or a piece of code that can execute concurrently sharing the address space with other tasks within a program (see for example [15]).

A *spawns* connector connects two tasks where the first is the creator (father) of the second task (child).

The task view represents a program's possible multiple threads of control and their relations. This is a slightly different view with respect to that of the invocation connector in the system view. In fact, in the case of the invocation connector the relation spans over executables, i.e. different processes, whereas for *tasks* the relation spans over subprograms or pieces of code executed in parallel within the same process.

Each module in the module view can in turn be analyzed and represented as a set of components and connectors generating the well-known *call-graph* and *data flow diagram* views. In the former case components are *subprograms* and its connectors represent control transfer from one subprogram to another; in the latter components are subprograms and data structures.

Finally, the *code view* illustrates the source code organization in the development environment. Code view elements are: source files, directories, libraries, include files and so on.

Level	View	Component	C Implementation	Connector	C Implementation
System	System View Code View	Program Program, File, Directory	same same	IPC connector Comprises, DependsOn	IPC (pipes, sockets ...) same
Program	Module View Task View Code View	Module Task Program, File, Directory	File Process same	Uses Spawns Comprises, DependsOn	Funcall, Var. access Fork same
Module	Call Graph Data Flow Code View	Subroutine Subroutine ,Variable Program, File, Directory	Function Function,Variable same	Call Data set/use Comprises, DependsOn	Function call Set/Use/Ref/Deref same

Table 1: *Summary of the architectural views with their components and connectors: the code view has been replicated through all the levels because it can be applied to any level. The fourth and sixth columns, titled* `C Implementation`, *give respectively the implementation of components and connectors for the C/Unix environment.*

At this level of abstraction possible relations are *is-component-of* and *depends-on*. The *is-component-of* relation means that system source code comprises certain directories, which contain in turn files, etc. *Depends-on* relations represent dependency information among source files such as the one that can be extracted for example from `makefiles`.

The code view corresponds to what is called *code architecture* in [12]. This view is not tied to a particular level, rather it can be computed at any levels, such as system, program and module level.

In Tab. 1 the different views defined together with the associated components and connectors are summarized.

The proposed architectural model is similar to others proposed in literature. Our extensions included the hierarchical organization of the different types of components and connectors in such a way that the software architecture of the system can be incrementally computed by the subsequent application of different analyses at increasing levels of detail.

The C/Unix environment was chosen for our experimental platform due to its rich variety of inter-process mechanisms.

Fourth and sixth columns of Table 1 shows the implementation of components and connectors at each level of the architectural model. The correspondence between generic components and connectors and their C/Unix instantiation is straightforward except for *modules*. C language does not provide explicit language constructs to define modules. A module could correspond to a group of files, a single file, a function or a group of functions. Nevertheless, in C the visibility of definitions (functions, global variables or constants) can be controlled at the file level using the keyword `static` and the include files. We will assume, in the following, that modules are implemented as separate files.

Components and connectors may be specialized in several different implementations. Fig 1 illustrates the relations among the general client-server connec-

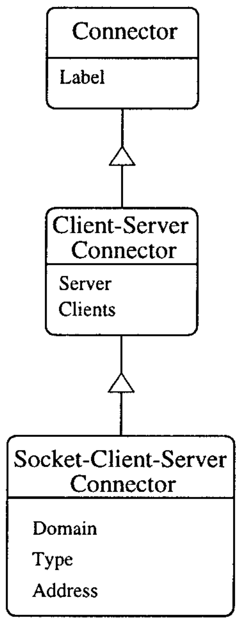

Figure 1: *An example of connector hierarchy: a socket based client-server connector for C/Unix environment. Under each connector class name, its attributes are listed. The arrows between boxes mean inheritance relations.*

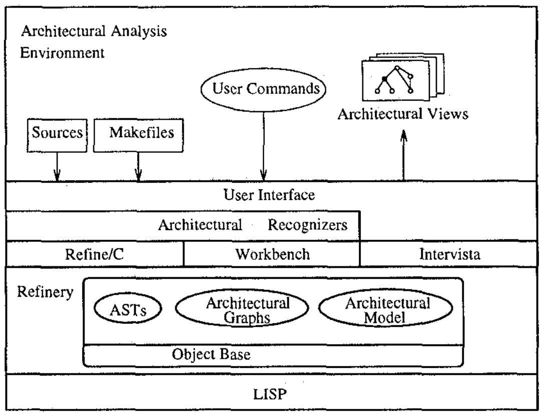

Figure 2: *Architecture of the analysis environment.*

tor and its C/Unix implementation based on sockets, using an object-oriented inheritance hierarchy. The same kind of connector could be implemented with other mechanisms such as pipes, RPC or others.

2.2 Analysis environment

In Fig. 2 the architecture of the analysis environment is illustrated. It is a layered system based on Refine. The first phase of the analysis process supported by the environment is called System Analysis and determines the components at system level. Since, in our model, such components are executable programs, we make the assumption that the system structure is specified using `makefiles` and the Unix command `make` is used to build a system. This is a reasonable assumption for most of the software produced in the Unix environment. Similar tools and concepts are also available in other operating systems, thus the above assumption is not a limitation.

A makefile is searched and analyzed in order to determine the separate independent programs that are part of a system. However, makefiles contain only a part of the information about the system level architecture of a software system and the rest of the information may not be explicit. To our knowledge integration or interconnection languages are not widely diffused although many such languages have been proposed and implemented in prototype systems [7, 9, 10, 13]. Therefore, retrieving system level information could involve the analysis of several different sources. In particular, in the Unix environment, shell scripts often contain relevant information about the architecture of distributed systems. Extracting shell script architectural information however is planned as future work.

Once the C programs involved in the system under analysis have been identified, the Refine/C package is used to parse them and produce the corresponding Abstract Syntax Trees (ASTs). In fact, all architectural recognizers work on the ASTs produced by the Refine/C parser. Several recognizers are applicable to ASTs and each of them is related to a specific level of the architectural model. They produce the different architectural descriptions of the source code. To represent such descriptions, hierarchical architectural graphs have been developed. These are directed graphs in which components are nodes while connectors are nodes or edges, depending on the specific view. They may be stored, retrieved (as objects in the Refine object base) or displayed producing hierarchical architectural views. In such a case, different layout algorithms are used depending on the graph to be displayed.

2.3 Progress status

Tab. 2 illustrates the set of recognizers that has been defined for each architectural level. Their development is in progress. Many recognizers at program and module level have been developed (e.g., `UsesRec, LayersRec, InterfaceRec, SpawnsRec, CallGraphRec`). Most of the the system level recognizers have not yet been implemented: we focused on distributed system and developed the `ClientServerRec, PipeRec` and `InvocationRec` recognizers.

The user interface implementation is also in progress: in particular we are investigating the use of some available graph display tools for the layout of system level views.

3 Architectural recognizers

An architectural recognizer is a fragment of Refine code that analyzes an AST identifying some special patterns of AST nodes called *architectural clichés* [1].

Usually, architectural clichés are AST fragments consisting of nodes corresponding to system calls (e.g. `fork, exec, system`) organized in specific control structures and revealing particular data dependencies among the variables involved. Fig. 3 shows two code fragments whose AST representations can be respectively matched by the *CreateChild* cliché and the *OpenPipe* one. It is worth noticing the difference in meaning of *architectural clichés* with respect to *architectural styles* [16]: styles are commonly used organizations of components and connectors while clichés are AST fragments that correspond to a particular component or connector.

Obviously many slightly different code fragments may be matched by a specific cliché. Hence some general features have to be recognized within source code to signal that a certain cliché is being implemented by that code. Moreover, the statements forming a cliché are usually spread [17, 18] i.e., they are not necessarily close each other nor even in the same procedure.

In the program concept recognition research community, clichés were originally developed using a *plan* representation [1, 2], where a plan is a collection of components and constraints. Fig. 4 shows a plan representing the first cliché instance of Fig 3. Control flow and data constraints are highlighted, as a general mean to express a cliché, independently of the physical location of its component statements.

Each recognizer embodies the knowledge of several plans. With respect to [19], recognizers have been improved by introducing flow analysis techniques to increase their capabilities and accuracy. In the next subsection the `ClientServerRec` recognizer will be used

Level	Recognizer	Component	Connector
System	SharedFileRec	Program	Shared File
	SharedMemRec	Program	Shared Memory
	PipeRec	Program	Pipe
	RPCRec	Program	RPC
	ClientServerRec	Program	Socket
	InvocationRec	Program	Fork, Exec
	CodeDepRec	Program	Dependency
Program	UsesRec	Module	Funcall, Shared var.
	LayersRec	Module	Funcall, Shared var.
	InterfaceRec	Module	Resource Export
	SpawnsRec	Task	Fork,Exec
	CodeDepRec	File	Dependency
Module	CallGraphRec	Function	Funcall
	DataFlowRec	Data/Function	Data set/use

Table 2: *List of defined recognizers and their level of applicability.*

```
/* creation of a child process */
    pid = fork();
    if(pid == 0) {
        /* son's code */ }
    else {
        /* father code */ }

/* opening a pipe with a child process
 to read its output */

    pipe(p);
    if((pid = fork()) == 0) {
        close(p[0]);
        close(1);
        dup(p[1]);

        execl(...)
    }

    close(p[1]);
    ...
```

Figure 3: *Examples of C code fragments whose ASTs can be matched by two common architectural clichés: code that triggers recognizers is in bold face.*

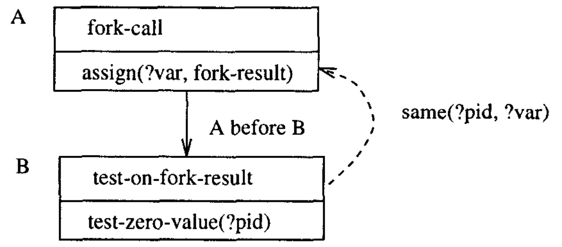

Figure 4: *A plan representing a process generation cliché.*

to illustrate flow analysis application within cliché matching.

At the system level, recognizers identify connectors: in fact, makefile analysis allows to determine the system components, which are in the set of makefile targets. Therefore, recognizers task is limited to discover relations among the previously extracted components. Some of the system level connectors, such as the *shared-file*, *shared-memory* and *message*, do not have a fixed arity, they are not limited to a specific number of participants. Their recognizers must be applied to each program that can possibly be engaged in a communication act with other programs.

Table 2 shows the set of recognizer available in our architectural model. At the system level the set includes:

- `SharedFileRec`: finds files that are used by different programs;
- `SharedMemRec`: finds shared memories referenced by different programs;
- `PipeRec`: finds a pipe communication channel between two processes, i.e. a specialization of the *stream* connector;
- `RPCRec`: finds calls to a remote service via the RPC mechanism;
- `ClientServerRec`: finds client-server connectors implemented by Unix sockets;
- `InvocationRec`: finds invocation relations between two programs;
- `CodeDepRec`: produces a view of the system in terms of files, directories, libraries and their dependencies. It can also be applied at the program level.

At the program level, each file of a program is analyzed by the `UsesRec` recognizer to discover references to external objects. An instance of a *uses* connector is created from a file to another when, for example, in the first file there are calls to functions or references to global variables defined in the second one. The output of this recognizer is a directed graph that is not necessarily acyclic. The `LayersRec` recognizer takes the graph output by `UsesRec` and compute a hierarchy of layers among modules i.e., a partial ordering. The following relation is used: level L of a module M belonging to a program P with respect to the *uses* relation is: $L(M) = i$ iff

$$\forall X \mid X \in Modules(P) \wedge M \in uses(X) \Rightarrow L(X) \leq i$$

Some details about the implementation of the `LayersRec` recognizer are described in Section 4.1.

The `InterfaceRec` recognizer recovers the interface of a module, that is the set of the resources exported to the other modules.

The `SpawnsRec` recognizer is used to extract the *task view* from program components. This recognizer builds a graph in which nodes represent threads of execution within the same process and edges represent *spawns* relations among nodes. A more detailed description is presented in [19].

`CallGraphRec` and `DataFlowRec` produce the well-known *call-graphs*, which represent control transfer among functions, and *data flow diagrams*, which describe systems as collections of data manipulated by functions. *Call-graphs* and *data flow diagrams* are the views available for module components.

Finally, the *code view*, merges information partly extracted from makefiles during the System Analysis phase (using the `CodeDepRec` recognizer) and partly obtained by analyzing the filesystem structure of the application source code.

3.1 An example: the ClientServerRec recognizer

`ClientServerRec` identifies the presence in the code of an architectural clichés implementing a client-server connector. Since components coincide with programs, the recognizer's task is limited to search for client-server communication channels. Fig. 5 shows a sample program that implements the opening of a communication channel from the server side based on BSD Unix sockets.

For each client-server connection found, the recognizer creates a `Client-Server-Connector` object (shown in Fig. 1). The attributes of such an object are retrieved from source code by the recognizer in two steps:

1. identify potential clients or servers separately;
2. identify possible bindings between them.

For the first step, the system calls `socket, bind` and `connect` are searched. Variables and values defined or used, by the system calls, are stored in a tuple according the syntax and semantics of the analyzed system calls.

```
main()
{
1    s_port = 3000;          {s_port=3000}
1'   s_port = atoi(argv[1]);                          {s_port:1'}
2    net_slisten(s_port);    {s_port=3000}            {s_port:1'}
     ...

}

int net_slisten (port)
int port; /* port on which to listen */
{
     ...                     {port=3000}

3    bzero (  (char *) &sin,
              sizeof (sin));
4    sin.sin_family = AF_INET;
5    sin.sin_port = port;   {sin.sin_port=3000} {sin.sin_port = port;
                                                  port=s_port}
6    s = socket(  AF_INET,
                  SOCK_STREAM, 0);
7    if (s < 0) return -1;

8    if (bind (s, &sin, sizeof (sin)) < 0)
9        return -1;

10   listen (s, 1);

11   for (;;) {

     ...
```

Figure 5: *A sample program opening a communication channel from the server side: two alternative ways of setting the* `s_port` *variable are shown (1 and* $1'$*). Each one produces different copy propagation and def-use information, illustrated within curly braces in bold face in two separate columns at the end of each statement: the first column corresponds to alternative 1 while the second to alternative* $1'$.

For each tuple `(socket, bind, listen)` and `(socket, connect)` a separate `Client-Server--Connector` object is instantiated. The attributes `server` or `clients` are respectively filled with the `program` object issuing such calls. The `domain` and `type` attributes are determined using information retrieved from the system calls arguments.

The `address` attribute value is needed for the computation of the second step: a client-server binding requires correspondence between the addresses used by a server and a client. If a binding is discovered, the two `Client-Server-Connector` objects corresponding to the server and the client are merged into one single object with the `server` and `clients` attributes filled. Fig. 6 presents an example of merged client-server attributes, where both the fields client and server are filled.

Flow analysis techniques can prove very useful to the computation of the `address` attribute, by providing more detailed information to recognizers. In Fig. 5, different ways of providing the `s_port` number are shown (labeled with 1 and $1'$).

If the structure member `sin.sin_port` in Fig. 5 is directly set to a constant value exactly once in the code, it is easy to determine the value of the `address` field and thus the establishment of a connection.

However, if variables or function call arguments are used to define `sin.sin_port`, then flow analysis could be used to retrieve some values for the `address` attribute.

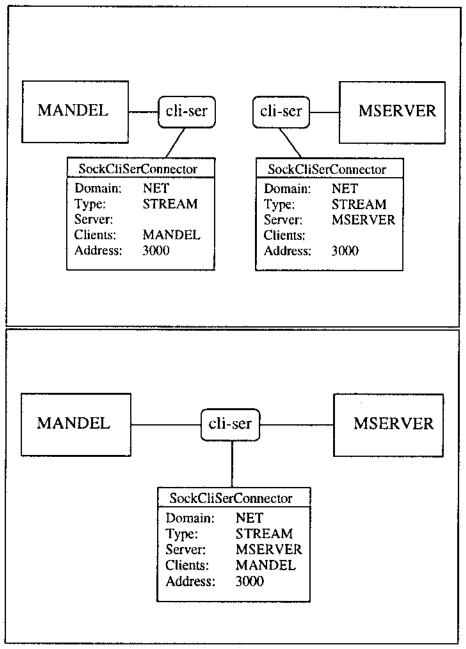

Figure 6: *Merging two client-server connector objects into one, after discovering a binding on the socket address.*

Whenever `bind` and `connect` have variable arguments resulting from a chain of variables copies starting from a constant, by using *copy constant propagation* [20, 21, 22], required attributes could be extracted. The first column beside statements 1, 2 and 5 in Fig 5 shows the information retrieved with such a technique.

When connection depends on the value of a variable, *copy propagation* plus *data-dependence* [20] allow to identify the statement which define the value of the first variable in the chain of copies. For example, considering the flow information of the second column beside statements 1′,2,5 of Fig 5), connection is established if `s_port` equals `3000`, i.e. if `atoi(argv[i])` equals `3000`.

If the previous techniques fail in providing the needed information, it is possible to compute a slice on `sin.sin_port` and determine the conditions for the connection by manually inspecting the slice.

Flow analysis techniques have revealed useful in the implementation of several architectural recognizers. The importance of program slicing [23] for such tasks has also been recognized in [6].

Slicing can be also used as a technique to support the extraction of the code implementing the components and/or connectors of a system. In program comprehension or in architectural restructuring and reuse, slicing may be useful to separate code according to the different functionalities accomplished. Program slicing has been used to segment large legacy systems written in Cobol [24] and also to identify reusable functions within C code [25]. For example, considering again Fig. 5, by merging the slices obtained on the variables used by the `(socket, bind, listen)` system calls at the end of the `net_slisten` function, a compilable and executable code to open a socket from the server side, that could be packaged in a function and reused.

The knowledge of which variables to slice on could be provided by mixing the use of architectural clichés and slicing: i.e., the knowledge of which calls are used to implement a specific architectural component or connector and how variables and function parameters are used and related each other, is codified in clichés and used by architectural recognizers to supply the slicer with the proper variables and the point where to compute the slice.

Obviously what can be obtained from slicing as suggested here is usually a raw approximation of a reusable object and must be rearranged manually by programmers. Nevertheless, it can be a valuable mean both to explain code and to extract reusable parts from it.

4 User interface

An explicit goal of our approach to architectural analysis is to provide a suitable visual representation for every analysis. Large textual structural descriptions of software systems can be summarized in a single diagram that carries the same amount of information. For this reason, a graphical user interface has been developed as a front-end to the analysis environment.

The user interface of our environment is based on the Intervista graphical package of Refine, that supports the creation and visualization of diagram windows, menus and text windows. It is organized hierarchically according to the architectural model. The analysis starts from the most general view, the *system view* and descends top-down, applying architectural recognizers to decompose components at one level into a view of components and connectors in the next level. For each view available, several operations may be performed by clicking either on window background or on nodes and edges of the displayed graph, and such operations are specific to each view.

The `mandel` system, which contains many architectural features, will be used throughout the following description of user interaction. It is a public domain system, about 3 KLOC of C, that performs Mandelbrot calculations and displays the results as fractals. The system architecture is client-server: a client, `mandel` in Fig 7, coordinates the work of multiple servers, performing Mandelbrot calculations, located on heterogeneous hosts on a local area network and displays the final result. Communication among `mandel` and each `server` is based on *sockets*. Each server is configured as a listener process that forks a separate process to handle client's requests and immediately goes back listening.

User interaction starts by specifying the root directory containing the source files of the system to be analyzed. During this phase, System Analysis is performed and the output is a preliminary version of the System View, in which only the program components are shown with no connectors among them. By

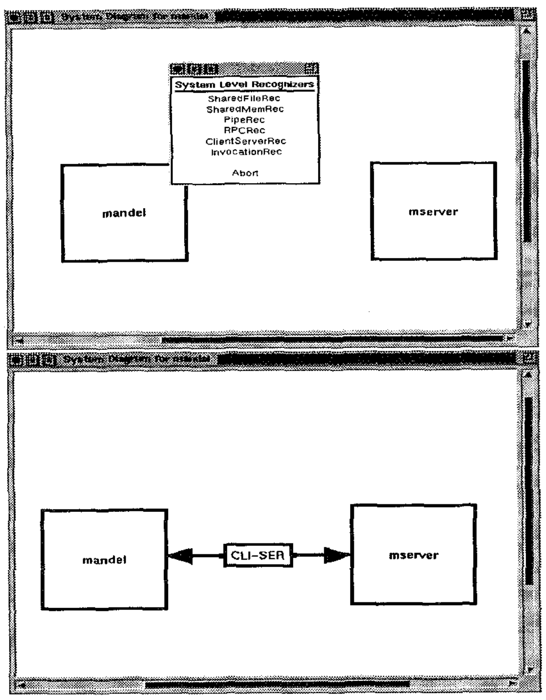

Figure 7: *A sample system view of the* **mandel** *system: after the application of a recognizer (superior view) a client-server connector is instantiated among two components.*

clicking on the window surface a menu of the system view level recognizers is displayed (see first part of Fig 7). The effect of the application of these recognizers is to connect some components with connectors corresponding to the chosen recognizer (as shown in the second part of Fig 7). For every level of analysis, the choice of which recognizer to activate is left to the user, who can have an a priori knowledge about the structure of the system and the mechanisms used.

The set of operations available for a component, corresponds to the set of architectural analyses, or recognizers, that can be applied to that element. Recognizers are selected via mouse by clicking on a component's icon. According to each specific recognizer, the result of applying a recognizer is either the creation of new objects in the current view or the creation of a new window in which the requested view is displayed. The `mserver` component of Fig 7, for example, may be further analyzed using the `SpawnsRec` recognizer: the resulting view is shown in the *task view* of Fig 8.

Clicking on connectors does not generate other views, but displays, when available, additional information about the connector. For example, in the case of the *uses* relation, clicking on a connector between two modules displays the list of resources belonging to the used module that are imported by the user module. In Fig. 9, a *module view* of the `mandel` program is shown, together with the import-export relations (their interface) between the two modules `gui` and `client`.

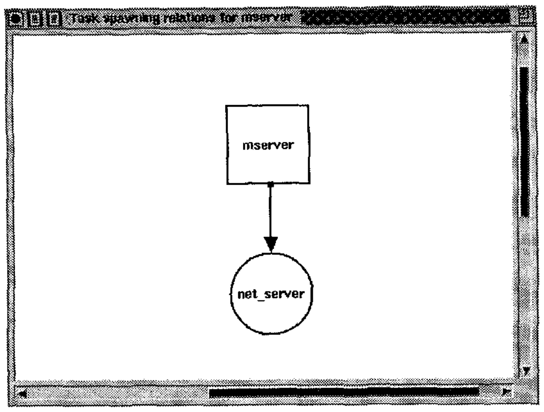

Figure 8: *A program task view: the rectangle represents the main execution thread (corresponding to the program name) while circles represent functions being called as task bodies.*

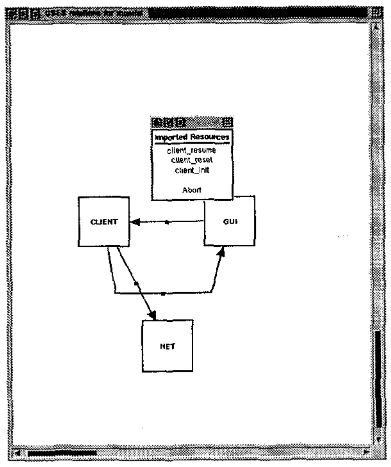

Figure 9: *An example of a module view of a program.*

Using the Refine environment capabilities, the results of the various analyses in the diagram form can be interactively manipulated (for example to add information obtained manually or through other tools such as the flow analyzer) and can be saved in a persistent database, for later use.

It is worth noticing that system level diagrams in our user interface usually represent *generic architectures*, concept introduced by Kramer and Ng in [13]. What is shown in a *system view* is a static view of a system's architecture, that is instantiated into a particular run-time system architecture. For example, Fig. 7 represents a client-server system, in which any number of instances of clients could be active at run-time.

4.1 Visualization

To visualize the various views of software architecture it is necessary to have a set of display algorithms, each suitable for the particular kind of information to show. Initially the nodes of the graphs produced by

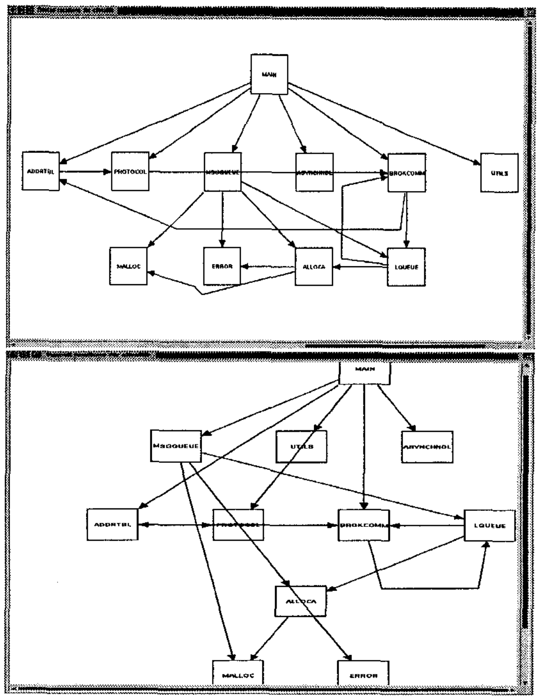

Figure 10: *Comparison of two different graph layouts for the same* uses *relation view: the first contains cycles among layers, while the in the second cycles are reduced within the same layers.*

recognizers have no position on the diagram surface. Thus, display algorithms have to assign a position to each node of a graph, according to the kind of graph to be displayed.

A heavily used layout for views computed by recognizers is a hierarchy of components based on their connectors. Hierarchies are the natural representation for the graphs produced by many of the recognizers available (such as **UsesRec**, **CodeDepRec**, **CallGraphRec**, etc.). To obtain a hierarchy layout, first the graph has to be organized as a set of layers and then each node has to be assigned a position and displayed in the layout.

To assign a hierarchical layer to each node of a generic graph, the **LayersRec** recognizer is available: it operates a transformation on the input graph corresponding to a clusterization of nodes into hierarchy levels. Presence of cycles within the input graph is handled by assigning all the nodes involved in a cycle to the same level, thus eliminating inter-layer cycles. In analogy with [26], cycles in a graph are detected by computing its *strongly connected components* [27].

Fig. 10 shows two different layouts to represent the *uses* relation among modules of a sample program. The first layout, obtained using an algorithm that starts from *unused* modules and builds each layer as the set of modules used by the previous layer, contains cycles. In the second, computed with the **LayersRec** recognizer, cycles are reduced within the same layers, and the resulting graph is actually organized as a hierarchy of layers.

For the visualization of system level views a hierarchical layout of components may not be a proper one, because components in this case represent independent interacting programs. At present, for such kind of views the system offers a default layout that can be rearranged by hand. It is our belief that the proper visualization of such kind of information is hardly feasible without a certain degree of user interaction. We advocate an assisted approach to diagram layout, such as the one presented in [13], where support tools, like the *magnet* tool, allow the user to easily specify the placement of graph nodes.

5 Conclusions

In this paper, an approach for the architectural analysis of software systems, together with an environment implementing such approach, have been described. The approach is based on a hierarchical architectural model that drives the application of a set of recognizers. Each recognizer builds an abstract view describing some architectural aspects of the system, or of some of its parts, and this view can be visualized. Recognizers may use flow analysis techniques to make results more accurate. The analysis environment is based on the Refine tools and its implementation is in progress.

The user interface of our analysis environment has been explicitly designed taking into account that each analysis should have a graphical output in terms of graph views and these views are organized hierarchically, according to the architectural model defined. The importance of a hierarchical approach to the visualization of software architecture has been underlined also by Kramer and Ng in the implementation of the Software Architect Assistant [13].

Our work presents analogies with that of Harris et al. [5, 6], and many similarities can be found between their work and ours: however, our approach differs from theirs as regards the conceptual organization of the architectural model, the analyses of system level information and dependencies contained in *makefiles* that our environment offers, the stress we put on visualization of analysis results and the different granularity of architectural recognizers (on average, our recognizers are more "coarse-grained").

Future work will be devoted to experimenting recognizers on larger systems, increasing the recognizers number and further investigating the use of flow analysis techniques for architectural recovery. Finally, we would like to investigate the analysis of other sources of information about the architecture of software systems, in particular those carrying information about multi-process and distributed system, such as for example shell scripts for the Unix environment.

References

[1] C. Rich and L. Wills, "Recognizing a Program's Design: A Graph Parsing Approach", *IEEE Software*, pp. 82-89, Jan 1990.

[2] V. Kozaczynski, J. Q. Ning and A. Engberts, "Program concept recognition and transforma-

tion", *IEEE Trans. Software Eng.*, vol. 18, n. 12, pp. 1065-1075, Dec. 1992.

[3] T. Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer*, July 1989.

[4] K. Wong, S.R. Tilley, H. A. Muller and M. D. Storey, "Structural Redocumentation: A Case Study", *IEEE Software*, pp. 46-54, Jan 1995.

[5] D. R. Harris, H. B. Reubenstein and A. S. Yeh, "Reverse Engineering to the Architectural Level", in *Proceedings of the 17th International Conference on Software Engineering*, pp. 186-195, Seattle, 1995.

[6] D. R. Harris, H. B. Reubenstein and A. S. Yeh, "Recognizers for Extracting Architectural Features from Source Code", in *Proceedings of the Second Working Conference on Reverse Engineering*, pp. 252-261, Toronto, 1995.

[7] R. Prieto-Diaz and J. Neighbors, "Module Interconnection Languages", *The Journal of Systems and Software*, vol. 6, no 4, pp. 307-334, Nov. 1986.

[8] T. R. Dean and J. R. Cordy, "A Syntactic Theory of Software Architecture", *IEEE Trans. Software Eng.*, vol. 21, n. 4, pp. 302-313, Apr. 1995.

[9] M. Shaw, R. Deline, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Eng.*, vol. 21, n. 4, pp. 314-335, Apr. 1995.

[10] R. Allen and D. Garlan, "Formalizing Architectural Connection", *Proceedings of the 16th International Conference on Software Engineering*, pp. 71-80, Sorrento, 1994.

[11] P. Inverardi and A. L. Wolf, "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model", *IEEE Trans. Software Eng.*, vol. 21, n. 4, pp. 373-386, Apr. 1995.

[12] D. Soni, R. L. Nord and C. Hofmeister, "Software Architecture in Industrial Applications", in *Proceedings of the 17th International Conference on Software Engineering*, pp. 196-207, Seattle, 1995.

[13] K. Ng and J. Kramer, "Automated Support for Distributed Software Design", in *Proceedings of 7th Int. Workshop on Computer-Aided Software Engineering*, pp. 381-390, Toronto, July 1995.

[14] C. Ghezzi, M. Jazayeri and D. Mandrioli, "Fundamentals of Software Engineering", *Prentice Hall*, Englewood Cliffs, NJ, 1992.

[15] H. M. Deitel, "An Introduction to Operating Systems", *Addison-Wesley*, 1990.

[16] D. Garlan and M. Shaw, "An introduction to software architecture", in *Advances in Software Engineering and Knowledge Engineering*, New York: World Scientific, vol I, 1993.

[17] S. Rugaber, K. Stirewalt and L. M. Wills, "The Interleaving Problem in Program Understanding", *IEEE Proceeding Second Working Conference of Reverse Engineering*, pp. 166-175, Toronto, July 14-16 1995 .

[18] L. M. Wills, "Automated Program Recognition by Graph Parsing", *Technical Report 1358, MIT Artificial Intelligence Lab.*, PhD Thesis, July 1992.

[19] R. Fiutem, E. Merlo, G. Antoniol and P. Tonella, "Understanding the Architecture of Software Systems", *IRST Technical Report 9510-06*, October 1995. To appear in *Proceedings of the Fourth Workshop on Program Comprehension [WPC96]*, Berlin, March 30-31, 1996.

[20] A. V. Aho, R. Sethi, J. D. Ullman, "Compilers Principles, Techniques and Tools" *Addison-Wesley*, 1988.

[21] M. Wagman and K. Zadeck, "Constant Propagation with Conditional Branches", *ACM Trans. Programming Languages and Systems*, 13,2(1991), pp 181-210.

[22] E. Merlo , J. F. Girard, L. Hendren and R. De Mori, "Multi-Valued Constant Propagation Analysis for User Interface Reengineering", *International Journal of Software Engineering and Knowledge Engineering*, Vol 5(1), 1995, pp 5-23.

[23] M. Weiser, "Program Slicing", *IEEE Trans. Software Eng.*, vol. 10, n. 7, pp. 352-357, July 1984.

[24] J.Q. Ning, A. Engberts and W. Kozaczynsky, "Recovering Reusable Components from Legacy Systems by Program Segmentation", *Proc. 1st Working Conference on Reverse Engineering*, Baltimore, USA, 1993.

[25] A. Cimitile, A. De Lucia and Malcolm Munro, "Identifying Reusable Functions using Specification Driven Program Slicing: a Case Study", *Proc. of the International Conference on Software Maintenance*, Nizza, France, 1995.

[26] Y. R. Chen, M. Y. Nishimoto, C. V. Ramamoorthy, "The C Information Abstraction System", *IEEE Trans. Software Eng.*, vol. 16, n. 3, pp. 325-334, March 1990.

[27] A. V. Aho, J. E. Hopcroft, J. D. Ullman, "The Design and Analysis of Computer Algorithms" *Addison-Wesley*, 1974.