A Case Study in Architectural Reconstruction

Michael Lykke
Software Developer
Bang & Olufsen A/S
Peter Bangs Vej 15
7600 Struer, Denmark
+45 9684 4525
lye@bang-olufsen.com

Jess Nielsen
Senior Developer
Trapeze Group Europe A/S
Søren Frichs Vej 38K
8230 Åbyhøj, Denmark
+45 8744 1631
jess.nielsen@trapezegroup.eu

Henrik Bærbak Christensen
Associate Professor, PhD
Department of Computer Science
University of Aarhus
Aabogade 34, Aarhus N, Denmark
+45 8942 5673
hbc@cs.au.dk

ABSTRACT

An understanding of a system's software architecture is central to successful system modifications. In the fortunate cases, the architecture is well understood as the original software architect and lead developers are responsible for maintanince. However, often systems must be modified based upon incomplete architectural information due to staff changes and incomplete or outdated documentation. In this case, software architecture reconstruction *is vital to reestablish overview and understanding of (parts of) the software architecture. In this paper, we report on a case study of architectural reconstruction using the Symphony process of an open source system. In the reconstruction process a set of open source and commercial reconstruction tools have been used and we report on benefits and liabilities of each tool, and dicuss their usefulness in building relevant architectural views.*

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.* This is just an example, please use the correct category and subject descriptors for your submission. The ACM Computing Classification Scheme: http://www.acm.org/class/1998/

General Terms

Design. Experimentation.

Keywords

Architectural reconstruction. Reconstruction tools. Symphony. Voice over IP. Lattix, MOOSE. Visual Studio 2008, Code Visual to Flowchart.

1. INTRODUCTION

The maintenance phase is well known to be costly for a succesful software product. A key aspect of maintaining, enhancing, and extending a software system is the ability of developers to overview, understand and analyze the software architecture of the system. However, more often than not, the software architecture is largely undocumented and only vaguely understood. A major problem with documenting software architecture is the uncertainty in predicting the exact nature of documentation necessary in the future. Often customer feature requests or new technological platforms demand documentation of a type and of aspects that are not obvious at the onset of system development. Thus, there will always be a demand for *architectural reconstruction*, that is, the ability to create a (partial) software architecture based upon the artifacts that define an existing software system: typically the source code base, old documentation, and maybe interviews with the initial architects, developers, and maintainers. This is a labour intensive and thus costly process and architects should have all necessary means at their disposal to ensure a cost-effective reconstruction process. Research efforts have thus been invested in defining tools and platforms to aid in the reconstruction process which has lead to a large number of tools. A comprehensive overview is given by Pollet et al. [5].

In this paper, we present a case study of architectural reconstruction on an open source *voice over IP* system, OpenSpeak. The main contribution of the paper is an evaluation of the Symphony process [9] and an analysis of benefits and liabilities of a set of tools to produce architectural views, in particular module, allocation, and component-connector views.

The paper is organized as follows: Section 2 and 3 give a brief overview of OpenSpeak and the Symphony process respectively. In section 4 we describe the set of tools tested to recover architectural information, followed by a discussion of experiences with both the Symphony process as well as the tools in section 5. Section 6 concludes on our experience.

2. OpenSpeak

OpenSpeak is an open source *voice over IP* application based on Speex and wxWidgets, aimed for casual gamers who like to chat while playing a game. The application runs on both a Windows and a Linux platform. Key metrics for the system are:

Programming Language	C++
Lines of Code	183.536
Available from	http://openspeak-project.org/
IDE	Visual Studio 2005 +
Components	Client, server

The article will depict a reconstruction of an open source project with the use of the principles described in [9] and by observing the systems runtime behavior. The basis for this reconstruction is to produce (or reproduce) the architectural documentation for the system.

3. Symphony

Symphony [9] is a view-based software architecture reconstruction process and defines three types of views:

- *Source view:* information that can be extracted from artefacts from the system, like source code, build files, configuration information, documentation, traces etc. Typically these views are non-architectural as they are detailed and contain much information.
- *Target view:* describes the as-implemented architecture of the system and contains information needed to inform the choice of architectural design change.
- *Hypothetical view:* the software architecture as the people who developed the system remember it.

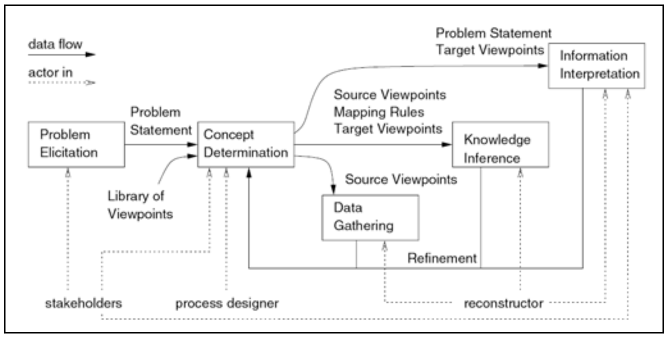


Figure 1: Symphony process.

As Figure 1 illustrates, Symphony defines a process in which stakeholders and architects define the system aspects they require a deeper understanding of, the problem elicitation phase, and the set of viewpoints required to make qualified decisions, the concept determination. The reconstruction execution consist of *data gathering* that establishes the source view by manual or tool-supported efforts, *knowledge inference* that applies mapping rule to filter, transform, and abstract the low level source views into target views, and finally *information interpretation* in which resulting information is presented and interpreted. The actual data gathering process is not a concern of Symphony and can be accomplished either by static or dynamic analyses.

4. Reconstructing OpenSpeak

The goal of the reconstruction process is to document the software architecture in three essential viewpoints as outlined by Bass et al [1, p. 37]: Module is a static decomposition in packages, classes, modules, etc. Component-connector (C&C view) is a dynamic decomposition in objects, processes and communication paths. Allocation view is a physical decomposition in deployment units, computing nodes, etc.

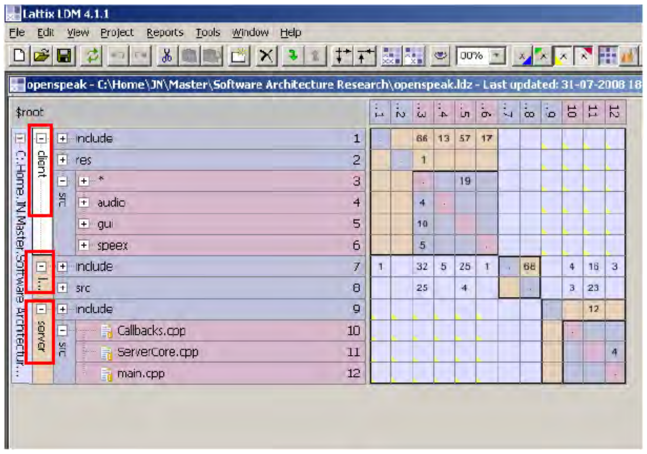
Below we outline the choice of tools used to try to construct these viewpoints.

4.1 Lattix

Lattix LDM is as commercial application, produced by Lattix, Ltd. (http://www.lattix.com). The application is primarily used to analyze the static structure of the source code. The producer highlights the following key features:

- It is capable of specifying the control and relationship between directories, source files, header files and idl files.
- It can analyze the relationship between the contents of the C / C++ source files and explores the dependencies at member level between: functions, variables, class methods, fields and macros significantly improves impact analysis and support for refactoring initiatives.
- The type of dependencies supported includes: class and data member references, invokes, inherits, constructs, include, and many others.

When structuring the source code it will typically be divided into directories. These directories might each represent a component of the system. An analysis of the #include directives in the source files will outline the dependencies among the source files. It will also outline the dependencies among the components for the source files that are placed in different directories.


Lattix LDM 4.1.1
File Edit View Project Reports Tools Window Help
00%
openspeak - C:\Home\JN\Master\Software Architecture Research\openspeak.ldz - Last updated: 31-07-2008 18
$root
C:.Home.JN.Master.Software Architectur...
client
l...
server
src
src

$root		1	2	3	4	5	6	7	8	9	10	11	12
include	1	.		66	13	57	17						
res	2		.	1									
*	3			.		19							
audio	4			4	.								
gui	5			10									
speex	6			5			.						
include	7	1		32	5	25	1	.	68		4	16	3
src	8			25		4			.		3	23	
include	9											12	
Callbacks.cpp	10										.		
ServerCore.cpp	11												4
main.cpp	12												.

Figure 2: Lattix's identification of the connections among the three major components of OpenSpeak.

In the illustration above, the tool has identified three directories as candidate components, each representing a part of the system. The three components are named "client", "lib" and "server". The numbers in the matrices shows the *degree* of dependency, if any. The numbers tells how many references it has found. Normally the number indicates that the vertical component uses the horizontal component n times, but when dealing with header files it is opposite. A missing number indicates that no relation exists.

To augment the visualization of the layers it is possible to add constraints to the components which is done by adding colors in the upper-right corner. By these constraints it can be illustrated whether the relation is legal or not. At the matrices above the yellow colors indicates the constraints that have been placed manually afterwards. The constraints state that all communication should be done through the communication layer, denoted *library*. Please note that no constraints are red. This means that no constraints have been violated. In other word it looks like a strict layered model.

Symbolism	Description	UML	View
Directories	Identifies the components	Packages and nodes	Module view (Allocation view)
Source files, #includes	Identify the relations	Object diagrams and classes	Module view
Source files, functions, class and data member etc.	Identify the relations	Object diagrams and classes	Module view
Source files, grouping	Group source files into components	Packages and nodes	Module view
Constraints	Identifies both legal and illegal intercommunication about components.		Module view

Table 1: The symbolism in Lattix mapped to a proper UML notation and an appropriate view.

The tool addresses especially the static module and allocation views. It identifies object diagrams and classes based on i.e. the classes and data member references that correspond to the module view and it identifies packages and nodes based on the directory structures that correspond to the allocation view.

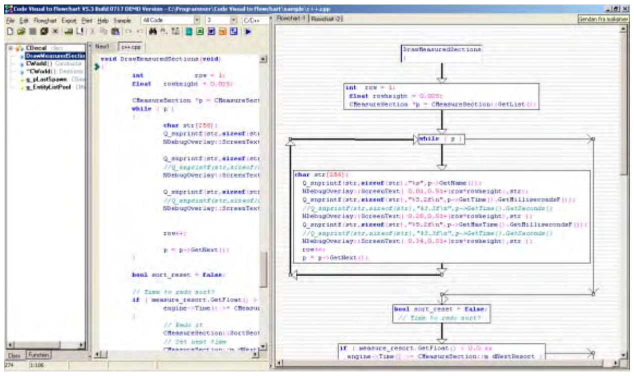
When analyzing the output generated by the tool be aware that not everything looks like what it seems to be like. As depicted earlier the tool recognized a strict layered model, but when doing a close manual inspection it was not just a layered model. In the matter of facts it was a complex reactor pattern [6, p. 179].

4.2 Code Visual to Flowchart

Code Visual to Flowchart is developed by FateSoft/Share-IT. The application has the capability to analyze source code and construct a flowchart diagram based on the flows in the source code. According to the company itself it has the following key features.

- It can reverse engineering a program with code analyzer; create programming flow charts from code, mostly used on flowcharting a program and documenting source code.
- It can generate Visio, Word, Excel, Power point, PNG and BMP flow charts document from code.

In the relation to these key features it can be used to examine the flows in the source code. A certain sequence of i.e. invocations or a large and complex function can be visualized to help the understanding of the source code. The flowchart can be augmented with code snippets and comments from the source code. It provides an easy navigation through the flowchart diagram.



```
void DrawMeasuredSections(void)
{
        int          row = 1;
        float   rowheight = 0.025;

        CMeasureSection *p = CMeasureSect
        while ( p )
        {
                char str[256];
                Q_snprintf(str,sizeof(str
                NDebugOverlay::ScreenText

                Q_snprintf(str,sizeof(str
                //Q_snprintf(str,sizeof(
                NDebugOverlay::ScreenText

                Q_snprintf(str,sizeof(str
                //Q_snprintf(str,sizeof(
                NDebugOverlay::ScreenText


                row++;

                p = p->GetNext();
        }

        bool sort_reset = false;

        // Time to redo sort?
        if ( measure_resort.GetFloat() >
                engine->Time() >= CMeasu
        {
                // Redo it
                CMeasureSection::SortSect
                // Set next time
```

```
DrawMeasuredSections

int  row = 1;
 float rowheight = 0.025;
 CMeasureSection *p = CMeasureSection::GetList();

while ( p )

char str[256];
  Q_snprintf(str,sizeof(str),"%s",p->GetName());
  NDebugOverlay::ScreenText( 0.01,0.51+(row*rowheight),str);
  Q_snprintf(str,sizeof(str),"%5.2f\n",p->GetTime().GetMillisecondsF());
  //Q_snprintf(str,sizeof(str),"%3.3f\n",p->GetTime().GetSeconds()
  NDebugOverlay::ScreenText( 0.28,0.51+(row*rowheight),str );
  Q_snprintf(str,sizeof(str),"%5.2f\n",p->GetMaxTime().GetMillisecondsF());
  //Q_snprintf(str,sizeof(str),"%3.3f\n",p->GetTime().GetSeconds()
  NDebugOverlay::ScreenText( 0.34,0.51+(row*rowheight),str );
  row++;
  p = p->GetNext();

bool sort_reset = false;
 // Time to redo sort?

if ( measure_resort.GetFloat() > 0.0 &&
  engine->Time() >= CMeasureSection::m_dNextResort
```

Figure 3: A screenshot from Code Visual to illustrate the flowcharts it creates.

The tool can be used to navigate through the code by clicking at functions. It will then illustrate the flow within the functions. Due to the usage of this tool it is not suitable as a standalone tool, because you have to know the source very well to know where to browse. Instead this tool can be used together with both static and dynamic analysis tools, when reconstructing a sequence diagram.

During the dynamic analysis a route of method invocations can be generated. This tool to illustrate the central flow inside the methods can then inspect the route further. This could for instance be an illustration of the flow used when establishing a connection or sending a message or adding a user as mentioned earlier.

Symbolism	Description	UML	View
State charts	Describes the flow using the syntax of state charts.	Object diagrams and stereotypes	C&C view

Table 2: The symbolism in Code Visual mapped to a proper UML notation and an appropriate view.

The usage of the tool addresses the dynamic component connector view by the use of a static analysis. The reason for this is that the analysis cannot be based on data collected by runtime, but the views themselves are illustrating a dynamic point of view – scenarios that can be walked through afterwards.

The output that is generated is done at a very low level basis. First of all you have to trace through the source code yourself – the primarily reason for it is a static analysis tool. Secondly, the flow that has been generated illustrates each method invocation and conditional statements found.

When using the tool it can be difficult to get an overview of a concrete scenario because of all the details. To make a more accurate addressing of the component connector view, it needs a trace log to maintain that remembers which methods that have been expected and in which sequential order. Finally an illustrative overview of the trace would be neat and it could be a sequence diagram.

4.3 Visual Studio 2008

Visual Studio 2008 Developer Edition (VS 2008) produced by Microsoft, also offers a set of analysis tools for both static and dynamic analysis. For static analysis, VS 2008 supports the following features for unmanaged C/C++ code:

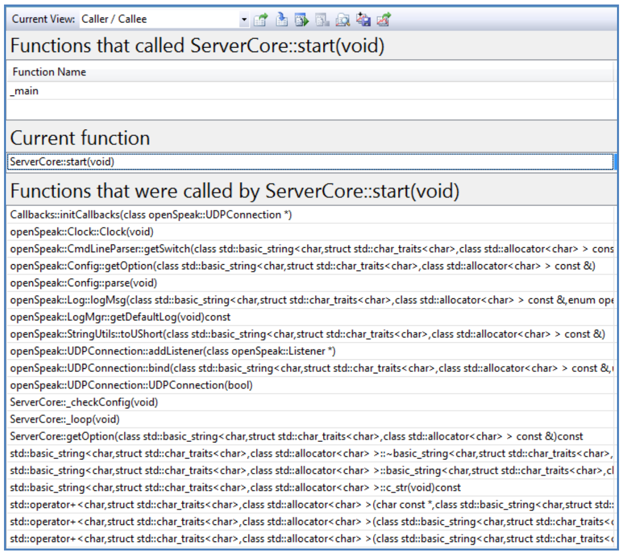
- Create a class diagram; however it only includes inheritance relations in the diagram, not the association and composite relations.

In dynamic analysis, it is possible to perform both code instrumentation and sampling. We only used code instrumentation, but the following views are available for both:

- Call tree view where it is possible to see the call stack of the system.
- Module view shows a list of the modules used by the system. A module is an executable file like the OpenSpeak.exe, a library as Kernel32.dll and User32.dll or other C/C++ native libraries.
- The caller/callee view shows all the functions calls made during system execution. Here you can enter an arbitrary function and see, which functions, has called this function and which functions this function is calling.
- Function view lists all the functions called during system execution.
- Memory allocation view shows how much memory each function and all its descendent has allocated.
- Object lifetime view shows the total instances of each type, and the amount of memory they consumes.
- Process view shows the processes that are executed during system execution.

Two views used when doing dynamic analysis, the memory allocation view and object lifetime view, required the code to be managed.

For the dynamic analysis, we expected to use the caller/callee view, shown below, as the primary tool, as this is a sequence diagram containing only a single executing, but surprisingly we did not, mainly because the amount of information and the details is overwhelming. And second, the list of "Functions that were called by …" is not ordered according to time (and it is not possible to do so either), making it difficult to find the sequence of methods calls.



```
Functions that called ServerCore::start(void)
Function Name
_main

Current function
ServerCore::start(void)

Functions that were called by ServerCore::start(void)
Callbacks::initCallbacks(class openSpeak::UDPConnection *)
openSpeak::Clock::Clock(void)
openSpeak::CmdLineParser::getSwitch(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > cons
openSpeak::Config::getOption(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &)
openSpeak::Config::parse(void)
openSpeak::Log::logMsg(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &,enum ope
openSpeak::LogMgr::getDefaultLog(void)const
openSpeak::StringUtils::toUShort(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &)
openSpeak::UDPConnection::addListener(class openSpeak::Listener *)
openSpeak::UDPConnection::bind(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &,
openSpeak::UDPConnection::UDPConnection(bool)
ServerCore::_checkConfig(void)
ServerCore::_loop(void)
ServerCore::getOption(class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > const &)const
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >::~basic_string<char,struct std::char_traits<char>,
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >::basic_string<char,struct std::char_traits<char>,cl
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> >::c_str(void)const
std::operator+<char,struct std::char_traits<char>,class std::allocator<char> >(char const *,class std::basic_string<char,struct std::
std::operator+<char,struct std::char_traits<char>,class std::allocator<char> >(class std::basic_string<char,struct std::char_traits<
std::operator+<char,struct std::char_traits<char>,class std::allocator<char> >(class std::basic_string<char,struct std::char_traits<
```

Figure 4: The caller / callee view from Visual Studio 2008

Even though VS 2008 is an advanced tool, it only includes inheritance relations when creating Class diagrams, as stated earlier. It is, however, possible to create relative simple mappings rules for finding association and composite relations, but never the less, they still have to be found manually.

Symbolism	Visual Studio 2008	UML	View
Class, function and data member	Class view	Classes	Module view
Class, function	Call tree view	Class and object diagrams	C&C view
	Modules view	Packages	Module view
	Caller / Callee view	Object diagram	C&C view
	Functions view	Object diagram	C&C view

Table 3: The symbolism in Visual Studio 2008 mapped to a proper UML notation and an appropriate view.

For example, we used the Class diagram from VS 2008 to identify the composite relation. When clicking on a class in the Class diagram, a list of all the names of the member variables is displayed along with their types. The association relation is more difficult to identify as they identify which classes that are either created in a function or a part of the parameter list of a function. The parameter list of a function, including the types, can be found in the Class diagram in VS 2008. Identifying classes created in a function can be done in MOOSE by selecting outgoing accesses from a class. Simple mappings rules can be written to identify member and parameter variables, and variables created in functions, for automating this task.

4.4 Moose

MOOSE is an open source project, started at the Software Composition Group in 1975. It is a language independent tool developed for reverse- and re-engineering legacy software systems [4, p. 2] and is a framework for software development based on formal models and code generation principles [8].

One of the nice features of MOOSE is that it allows developers to write their own parser to translate the code into the FAMIX meta-model, supported by the MOOSE core. The meta-model is an object-oriented representation of entities representing the software artifacts of the target system [4, p.2]. This means that all information transformed into MOOSE has the same internal representation, which other tools can use to extend the functionality of MOOSE by using the SmallTalk language.

MOOSE offers a lot of different views, where some of the views uses colors and shapes to express the essence of the view. The following is a short presentation of some of the views offered:

- Blueprint complexity: shows the internals of a class, by dividing the class into five layers, which are described in [7] (Initialization, Public interface, Private implementation, Accessor, Attribut).
- Method distribution map: visualizes the number of methods each class has.

- Method invocation: This view is similar to sequence diagrams in UML. It visualizes how the methods of the classes invoke each other by using a color scheme. Big rectangles represent classes and small squares are methods, as shown in Figure 5.

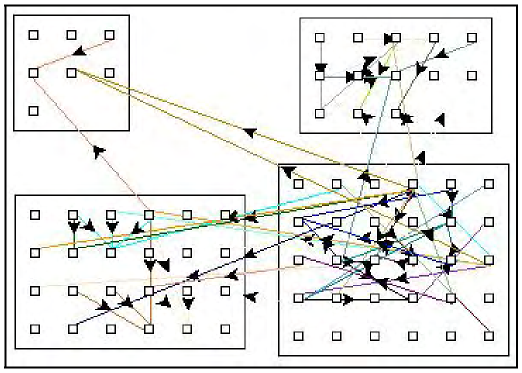


Figure 5: Method invocation diagram

- Class diagram: the traditional UML class diagram only containing the inheritance relations.
- System complexity: The system complexity visualizes a class hierarchy where each rectangle resembles a class. It uses colours and shapes to visualize the complexity.

Symbolism	**MOOSE view**	**UML entities**	**Architectural View**
Class, function and data member	Blueprint complexity	Classes and objects	Module view, C&C view
Class, function	Method distribution map	Class	Module view
Class, function	Method invocation	Class and Object diagrams	C&C view
Class, function and data member	Class diagram	Classes	Module view
Classes	System complexity	Classes	Module view

Table 4: The symbolism in MOOSE mapped to a proper UML notation and an appropriate view.

One of the more powerful features of MOOSE, is that the parser used, captures a lot of information about the source code. For instance, MOOSE creates different kinds of lists, which contains information's of all classes, variables, method invocation, namespaces, outgoing/ingoing accesses from a class etc. Unfortunately one of the weaknesses of MOOSE is that it does not support any search features. Finding specific variables to see which classes uses it, takes time.

The most used view from MOOSE was the Method invocation view. It gives a good overview and visualization of the collaboration among classes. This is useful when creating class diagrams that supports a scenario. The set of classes used in a scenario is typical only a subset of all the classes constituting the system. Finding the first class in this subset is the hard part, which may require domain knowledge, debugging or heuristics, but from that point it is easy to find the others by following the arrows in the Method invocation view.

The Method invocation view can also be used when reconstructing sequence diagrams for scenarios, as shown in figure 6. Again, finding the starting point is the hard part, then the overview provided by the Method invocation view, helps identify collaborated objects.

The time it takes to make sequence diagrams, using the Method invocation view, is roughly the same as without using it. It only helps to find collaborated objects, as it does not allow you to see the structure of the code, as Code Visual does, which is required to build the sequence diagram. However, using it for Class diagrams is faster, as you do not have to look inside the code to see collaborated classes. Besides, the list of outgoing accesses from a class is useful for finding the association relations.

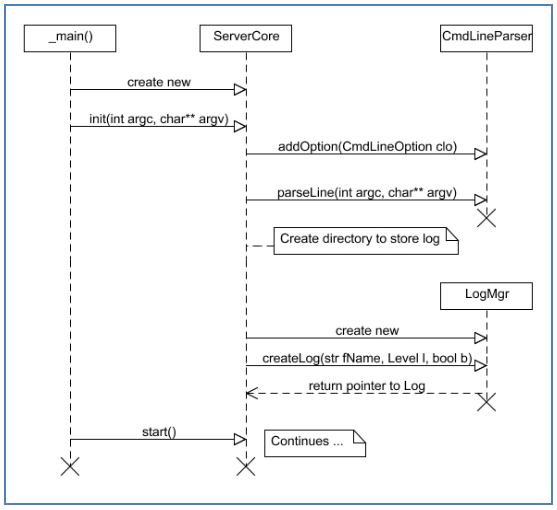

_main()
ServerCore
CmdLineParser
create new
init(int argc, char** argv)
addOption(CmdLineOption clo)
parseLine(int argc, char** argv)
Create directory to store log
LogMgr
create new
createLog(str fName, Level l, bool b)
return pointer to Log
start()
Continues ...

Figure 6: Sequence diagram illustrating the start process of the server.

5. Discussion

Next, we discuss the experience of the reconstruction process from two perspectives. First we outline our experience with the Symphony process and judge its value and second we examine the value of the tools used and reflect upon the amount of automation provided.

5.1 The Symphony Process

During the reconstruction process we noticed that rules and conventions that could be formalized as pure mathematic formulas, has been / could be integrated into reconstruction tools and executed automatically such as extracting inheritens relations when building class diagrams or defining dependencies among components by looking for the #include directives. However – no tools can be considered 100% automatic as it requires human interaction at some point [3, p. 3]. Especially some complex rules, during reconstruction, could not be formalized mathematically, but instead as heuristics, guidelines or other informal approaches accomplished by manual inspection.

The amount of information gathered in the data gathering phase can be overwhelming, depending on the tool used, as when using VS 2008 for code instrumentation during dynamic analysis or the variaty of lists produced by MOOSE, containing a lot of information about the source code, without search possibilities. During the knowledge inference phase, this information is used to populate the target views. This is done by using mappings rules and knowledge about the problem domain or whatever knowledge needed to take advantages of the information gathered. This is a hard job. It both requires a great deal of knowledge about the tool and the artifacts from which the information is extracted. For example, we used dynamic analyses on the source code in order to find out the execution path through the modules to build a sequence diagram. VS 2008 produced at lot of detailed information, where some of it only blurred the picture. Even the mapping rules defined in the source view, is not of any help if there is too many informations available. First, it is difficult to define a uniform way of using the information, because the next time you gathers the information it is just a little bit different, but still enough so you can't use the former rule and second, you loose orientation. To define formal rules in this situation, or just heuristics or guidelines, would produce a long detailed list of expressions that only can be understood by the person who wrote it. This part of the process; to figure out how to use the extracted information and define mappings rules, is not a part of Symphony, but is the product of domain knowledge (and fantasy) of the developer. If the information cannot be used, then the only solution is to try to find alternative ways of gathering the information needed. We gave up on the dynamic analyses approach, and used manual inspection together with Code Visual, to extract the information needed to build the sequence diagram.

Another place where mapping rules came to short, is when creating mappings rules for finding specific connectors in the Component & Connector view. First – rarely can a connector be identified by a single method call, but requires a set of related method calls in proper sequence, a protocol. Secondly – the set of related methods may be declared and implemented in different classes and/or modules, which makes it harder to identify, both for reconstruction tools and humans using manual inspection.

Even though we sometime had problems gathering the right information, the Symphony process works very well for building diagrams for architectural views. Through the process we iterated between the steps of phase two, as it seems the natural way of working. You can't use the waterfall approach when reconstructing the architecture; the process has to support an iterative way of working as Symphony does.

The tools managed to create almost all views using the tools. Lattix is good when getting an overview of the architecture for building packages diagrams for the module view. Code Visual is good for getting af overview of the flow of a method, when building sequence diagrams for Component & Connector view. MOOSE and VS 2008 is more allround tools and helped produced the information we needed to create the Class diagrams and helped supplementing the Sequence diagrams.

They did not manage to use any tools for building deployment diagrams for the allocation view. This was instead accompliched by trying different kind of installations/configurations settings. However Lattix did support us with the initial step, as it identified the major components; client and server. Secondly, the tools were unable to capture data from the running system such as the connectivity among components and which connector types that was used.

5.2 Experiences with the Tools

A few general methods that can be used to reconstruct software architectures are outlined in this section. The methods are categorized into two categories, which are static– and dynamic analysis. The static analysis is based on the applications artifacts while the dynamic analysis is used to capture data from the running system. Whether the tools support either static analysis, dynamic analysis or both are briefly summarized in the taxonomy below.

Tool	Static Analysis	Dynamic Analysis
Lattix	X	
Visual Studio 2008	X	X
Code Visual	X	( )
MOOSE	X	X

Table 5: Categorizing the tools according to the two types of analysis.

The extraction of the tools has been made with a number of different tools. These tools cover both static and dynamic analysis. It had the advantages that they augmented each other even that some gaps still remain. To close these gaps it has been necessary to use more manual procedures.

The package diagram was done with the help of the Lattix tool alone, in an iterative process. Lattix identified the three major components (the server, the client and the library), but the subcomponents required a little manual help.

MOOSE and Visual Studio did the class diagram, but Visual Studio includes more information. However, the only relation reconstructed automatically was the generalization. The missing relations, associations and composites were made manually to complete the Class diagram. Information for these relations was extracted using both tools.

The construction of the sequence diagram was a bit more challenging. The first approach was based on dynamic analysis, but this ended up being inappropriate because of the large amount of data produced by VS 2008, it was impossible to define neither mappings rules nor heuristics for using the data. The second approach was more successful and based on manual inspection as depicted in section 5.1.

For the dynamic analysis the tool, VS 2008, generally produced at lot of detailed information, where some of it only blurred the picture as depicted in 5.1. For example, calls used on C++ Standard Template Libraries, as vector or map to insert or get data, is too detailed. To make the dynamic analysis better, it would be more appropriated if it was possible to exclude or include certain libraries from the analysis.

The experiences gathered when examine the tools; can be outlined by describing the advantages and disadvantages for the tools, which can be found in the taxonomy in appendix at the end of the paper.

6. Conclusion

How much of an architectural description that needs to be reconstructed depends on the task at hand and can range from a single diagram, if the company regularly updates it documentation, to everything included in an architectural description. If everything is required, reconstructing the architectural description is not a process that is done over one night. A lot of data needs to be extracted from the artifacts of the system and further analyzed, visualized and then analyzed again.

To collect the data needed for the reconstruction two general techniques can be used: A static analysis which collects its data from the artifacts belonging to the application, and a dynamic analysis which collect its information from a running system.

In general it is very difficult for tools to collect data from an existing application. This makes it even harder to find tool that is suited for your special needs. Some commercial products exist on the market, but very few of them is capable of collecting all the data you might need for the reconstruction process as you have to combine several tools, but the disadvantages with multiple tools are often a missing collaboration between them. When selecting the tools to be used it is important to be aware of the prices (if commercial products are chosen) because the economically aspects are definitely not cheap and the initial costs therefore increasing unexpected.

The future work is to make a further investigation of the features of the tools to make a more automatically process of the sequence diagram generation and pattern recognition. We plan to continue to use the Symphony process when doing reconstruction as it describes a formal approach for a complex task. Secondly we think that MOOSE promote new ways of looking at data that could be suitable. Finally we will recommend the use of some of these tools, especially MOOSE and Lattix, in the daily work.

7. REFERENCES

[1] L. Bass, P. Clements, and R. Kazman.
Software Architecture in Practice, 2nd Ed.
Addison-Wesley, 2003.

[2] H. B. Christensen, A. Corry, and K. M. Hansen.
An Approach to Software Architecture Description Using UML. Technical report.
Department of Computer Science
University of Aarhus
URL: http://person.au.dk/da/hbc@cs.au.dk/pub.

[3] M. Jha, P. Maheshwar, T. K. Anh Phan
A Comparison of four Software Architectures. Reconstruction Toolkit
School of Computer Science and Engineering
The University of New South Wales, Sydney, Australia

[4] O. Nierstrasz, S. Ducase, and T. Girba.
The story of moose: An agile reengineering environment. Technical report.
University of Berne, 2005.
Accessed through www.bnet.com.

[5] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus.
Towards a process-oriented software architecture reconstruction taxonomy. In Proceedings of the 11th European Conference on Software of Maintenance and Reengineering (CMSR).
Pages 137 –148.

[6] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann.
Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.
John Wiley & Sons, 2000.

[7] University of Berne
Viz: Polymetric Views
http://moose.unibe.ch/docs/polymetricviews.
Accessed September 2008.

[8] Universität Kaiserslautern.
MOOSE: Model-Based Object-Oriented Software Generation Environment.
http://wwwagz.informatik.unikl.
de/projects/SFB501/D1/MOOSE/index.html.
Accessed September 2008.

[9] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva.
Symphony: View-Driven Software Architecture Reconstruction. In Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, pages 122 - 132, 2004.

8. Appendix

The table below summarizes benefits and liabilities of the selected set of tools.

Tool	Advantages	Disadvantages
Lattix	• Excellent clarification of the components and their relations.	• It cannot generate its output from source code itself. • Data is visualized in matrices not in diagrams. • Not suited for scenario based approaches.
Code Visual to Flowchart	• An easy navigation through the code by the use of automatically generated flowcharts. • Useful when diagnosing connectors and complex business rules.	• It operates at a very low level, which makes it difficult to extract high level / architectural information.
Visual Studio 2008	• Easy to get started. • Contains the most common views.	• Only possible to produce the views during dynamic analysis (except when building Class diagrams). • Only possible to use predefined filter options. • Some features only works with managed code.
MOOSE	• Supports many view. • Possible to make you own views of the data. • You can write your own parser and input the data into MOOSE. • A lot of views can be made based only on static analysis. • Promote new ways of looking at data (i.e. method invocation and blueprint complexity)	• Lack of documentation. • Difficult to make your • own views (because of the use of SmallTalk and lack of documentation) • Not possible to search through the data (i.e. when searching the list of all functions calls, it would be desirable to search for specific classes or methods).