Library Evolution for Reliable Software

Noritoshi Atsumi  
Department of Information Engineering  
Nagoya University  
Furou, Chikusa, Nagoya, 464-8603, Japan  
atsumi@agusa.nuie.nagoya-u.ac.jp

Shoji Yuen  
Department of Information Engineering  
Nagoya University  
Furou, Chikusa, Nagoya, 464-8603, Japan  
yuen@nuie.nagoya-u.ac.jp

Kiyoshi Agusa  
Department of Information Engineering  
Nagoya University  
Furou, Chikusa, Nagoya, 464-8603, Japan  
agusa@nuie.nagoya-u.ac.jp

Shinichirou Yamamoto  
Department of Information Science  
Aichi Prefectural University  
Ibaragabasama, Kumabari, Nagakute, Aichi 480-1198, Japan  
yamamoto@ist.aichi-pu.ac.jp

ABSTRACT
This paper shows a methodology to evolve a library of functions to be with common patterns of usage, called idioms. In source programs written in C, it is commonly seen that library functions are combined in a certain pattern since functionalities required in application software are more abstract than those of the library functions. This means that using appropriate idioms for the purpose is one of the key issues in implementing functionalities of the software. Investigating idioms in the existing elaborate software in terms of control and data dependencies of library function-calls, it is observed that “good” idioms contribute to the reliability in developing and maintaining software. Based on this idea, we extract idioms from the existing open-source software in the form of FCDGs (Function Call Dependency Graphs). By categorising those idioms as a database of FCDG patterns, we propose evolved libraries where each library function is accompanied by the typical reliable patterns of usage. By exploiting an evolved library, we show how effectively potential defects that may lead to unexpected behaviour are resolved by checking the conformance to our idioms.

Categories and Subject Descriptors
D.2 [Software]: Software EngineeringReusability

Keywords
Library Functions, Function Call Graph, Idiom, Domain Specific Library, Reuse

1. INTRODUCTION
In application software, library functions are the primitive functionalities to interface with the run-time environment or application packages for special purposes. While developing application software, the library functions are usually combined in certain patterns since the required functionality is much more abstract than those of the library functions themselves. Moreover, the library functions are inherently designed to be used in such a combinatorial way rather than independently. This enables the library to provide generality and flexibility in that they can provide the necessary information about the specific states of what the library controls on demand in various situations.

This inevitable feature of library functions is one of the main causes of the potential defects of application software. An incomplete combination of library functions may cause unexpected behaviour against rarely used input values or rarely occurring contexts. For instance, unclosed file descriptors may cause a problem when too many files are open, where the number of opened files is not checked by the application. This kind of problem is difficult to detect by testing the software in general since it depends on the run-time environment where the test is performed. Such a knowledge about the usage of library functions is not usually documented in the library manual, nor in the specification directly.

In order to give a solution to the problem above from the software engineering point of view, we propose evolved libraries where each library function is referred with the common patterns of usage. As seen in many long-used application programs, certain patterns of library function-calls are repeatedly used in order to achieve reliable implementation. Analogous to the natural languages, we call these patterns idioms. We claim that “good” idioms should improve the software reliability not only in development but also in maintenance. In addition, it also promote the reuse of software components. We believe, by following the reliable idioms syntactically, it is possible to uniformly assure reliability for that software.

For this purpose, we formulate idioms by function call dependency graphs[1][9], FCDG for short. An FCDG is able to express the abstracted dependencies of library functions both in terms of control and data. Focusing on a particular library function, we extract a substructure relevant to the library function as a partial FCDG from the whole FCDG. Examining common partial FCDGs in open source software written in C to be well structured in the dependencies, we...
construct an evolved library of functions as a database of those well-formed partial FCDGs that are "idioms". Given a library function, if the FCDG derived from an application program contains some partial structure that does not conform to partial FCDG for the function, then the program may have some potential problem even though it works correctly at the moment. By completing the pattern that conforms to the idioms, more reliable coding is obtained without changing the original functionality.

We have been investigating the substantial amount of open-source software to find out the candidate patterns of idioms. By analysing dependency in those patterns, we give categorisation of patterns to obtain reliable idioms. Based on our categorisation, we build an evolved library of the functions for the system calls in FreeBSD 4.3-Release. A more reliable version of software is obtained by replacing a code fragment recognised as an idiom with a function from the evolved library.

Finally, to show the contribution of our method to the evolving process of software, we present a preliminary result in seeking the appropriate idioms useful for version change with respect to the size and the number. We present that the idioms extracted for FreeBSD 4.3-Release also appears those for FreeBSD 4.4-Release by setting the parameters for idioms. This shows our method should ease and promote the reuse of code fragments as idioms without losing the reliability while the software evolves.

This paper is organised as follows: in section 2, we briefly show how a library pattern appear in a program. In section 3, we define the dependency relation between library functions and in section 4 we review FCDGs. In section 5, we propose a evolutional library that is a library accompanied by reliable idioms implemented in the form of partial FCDG database. Section 6 gives our concluding remarks.

2. IDIOMS IN LIBRARY FUNCTION CALLS

In the network programming, a common pattern of library function-calls typically appears as follows: in order to communicate through a socket, a socket is created as a descriptor, bound to a memory area, enable to listen, and finally accepted. For each step, one function of the network library is called. In this case, the sequence of the functions is open, bind, listen and accept. Although the intended functionality is to begin a communication through a TCP socket, in most cases we always need to follow the sequence above. In this example, one reason for these fine grained library calls stems from the protocol specification. If the communication channel should fail to open, different error handlings are necessary dependent of the stage in establishing the connection. However, at writing a programming, following the function sequence is a tedious job.

Another example is that the correspondence of open and close for a file descriptor. It is desirable to close a descriptor when it is opened in the program. exit function normally closes all the descriptor when the program is terminated. However, in a program such as a daemon process it is very important to keep the correspondence for reliable behaviour of the program.

The exception handlings usually follow in common patterns. For instance, it is desirable to check whether NULL pointer returns as the return value of fopen function call. Sometimes, this kind of typical check may be omitted when a programmer is sure that the exception unlikely happens.

But in such a case that the component is embedded in other modules, an unexpected error may occur. In the dynamic memory management using malloc as in the heap data structure, more complicated situation may arise unless the exception is strictly handled since the memory acquisition and release are highly dependent on the dynamic run-time environment.

Observing the above examples, the patterns of library function calls are very important when a programmer tries to implement a certain abstract functionality in a reliable way using library function calls. Ressinfect of the natural languages, such idioms of library function calls are recognised as the common patterns of usage. We consider that idioms of library function calls are very important in the following senses:

1. Idioms form abstract high-level functions:
   In many cases for basic functionalities, idioms suited to the objectives exist. A programmer does not need to build the functionality from scratch. Thus, idioms contribute to the productivity.

2. Idioms contribute to the software reliability:
   If an idiom is used as the template for a functionality, to fill in a part of the idiom can complete the whole idiom. If an idiom is known to be reliable, all patterns following the idiom are also reliable.

3. Idioms promote re-use:
   Idioms are repeatedly used for a common functionality.
   If a programmer would like to implement a similar functionality, it is possible to re-use the part of the idiom and fill in the rest. Idioms are first advocated for C++ code re-use[4]. The methods to reuse similar code fragments in existing codes are proposed[2][3].

In defining idioms, we focus on the dependency relation between library function calls. The dependency relation explicitly exists as passing parameters and returning values. Or, the relation implicitly exists as the control flow within the run-time environment, as seen in the network programming. Therefore, we define idioms as the common structures in the dependency relation. Among those structures, well-formed structures in terms of the dependency relation and exception handling are considered as "good" idioms. We believe that those good idioms must appear in the existing reliable software and should be encouraged to appear to produce reliable software.

By explicitly formalising idioms as the dependency relation structure, it is possible to help a programmer to encourage to follow good idioms in the development environment in general. The environment is able to warn the programmer against the incompleteness of exception handling or able to automatically generate the complete code in certain cases. We present the framework of idioms and their use in the software engineering view point.

3. LIBRARY EVOLUTION

In our framework, Library Evolution is to give the typical and reliable FCDGs as idioms with respect to a given library function. Since we aim at providing an effective method for the practical programming development, we shall give idioms based on the actual usage in the existing C programming. As the material for this, we treat the source codes distributed as “Open sources”. Different from the source codes
of commercial software, the source codes in the open-source community can be widely used freely. As the result, they may be used in almost every possible situation. If any malfunction should occur, the defect can be a feedback as the bug report. This means that the source codes are brushed up in order to achieve the desired functionality in a reliable manner for a long period of time. We believe that that kind of source codes should contain “good” idioms.

Based on this observation, we construct FCDGs of FreeBSD 4.3-Release as a typical example of open-source software. We extract common patterns of code fragments as idioms in the following three steps: (Step 1) Common substructures of the FCDG are extracted, whose size are determined by a fixed parameter. (Step 2) For each library function, sort the partial FCDGs by exact matching. (Step 3) A set of the partial FCDGs is realized as an idiom when the number of identical partial FCDGs exceeds a fixed parameter. An evolved library is obtained as a set of such idioms. This process is illustrated in Figure 1.

3.1 Example of Evolved Library

Figure 2 shows the process of an evolution of library functions. As in [9], FCDGs are extracted from source codes by using Sapid[7].

We actually tried to create FCDG for 212 programs in “/usr/src/usr.bin” on FreeBSD 4.3-Release, and build FCDG Database. Figure 3 show the result which searched “fopen” from the FCDG Database. This figure shows partial FCDG and the rate of the same partial FCDG of FCDG including fopen. A part of the result is shown in Figure 3, and the remains are idioms which the same partial FCDG is only several percents. The same pattern as partial FCDG (a) exists mostly in the database. Therefore, the idiom (a) is a candidate of high level library function. The high level library function is shown in Figure 4.

In this example, in extracting partial FCDG we determined the rate of the combination pattern between nodes 40%. As mentioned above, this rate determine the size of partial FCDG. In the programs of FreeBSD, 40% is best. When the value is 30%, the number of partial FCDG patterns are so little in order that the idiom size is big. When the value is 50%, the number of partial FCDG patterns are so many in order that the idiom size is small. The suitable value is different by target programs.

Table 1 show the results obtained by searching library functions, “gethostbyname” and “socket”, from the FCDG DB. The rate of the identical partial FCDGs among the whole FCDGs with the library functions and the nodes ap-

Figure 2: experimental system

Figure 3: fopen search result
Figure 4: example of evolved library

Table 1: search result
gethostbyname

<table>
<thead>
<tr>
<th>Rate(%)</th>
<th>54</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>gethostbyname</td>
<td>gethostbyname</td>
</tr>
<tr>
<td></td>
<td>(=, NULL)</td>
<td>(=, NULL)</td>
</tr>
<tr>
<td></td>
<td>warn</td>
<td>perror</td>
</tr>
<tr>
<td>socket</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rate</th>
<th>48</th>
<th>22</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>socket</td>
<td>socket</td>
<td>socket</td>
</tr>
<tr>
<td></td>
<td>(0, 0)</td>
<td>(0, 0)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>perror</td>
<td>perror</td>
<td>warn</td>
<td>bind</td>
</tr>
<tr>
<td>bind</td>
<td>(==, -1)</td>
<td>(==, -1)</td>
<td>(==, -1)</td>
</tr>
<tr>
<td>perror</td>
<td>close</td>
<td>connect</td>
<td>(0, 0)</td>
</tr>
</tbody>
</table>

Figure 6: evolved library function for the idioms

Figure 5

Figure 7 shows some examples of the idioms for “fopen”. In the later version, it is seen that the idioms in the previous version appear as the substructure of the new idioms. Thus, by replacing the idioms with the functions in the evolved library, the idioms in the later version should have been smaller in size or it could have been possible to capture more abstract idioms. This fact is generally seen throughout the whole program. This means that our method promotes the software reuse without decreasing the reliability in the evolving process.

Although it is important to determine the appropriate parameters to determine idioms in order to achieve the effective reuse, we do not know yet how to determine those parameters in advance. So far, they have been obtained by trial-and-errors. For the time being, we have to run our tool several times to determine the parameters.
4. RELATED WORK

Program Dependence Graph is used in order to solve this problem. The method to identify similar code with Program Dependence Graphs in order to capture reusable code is proposed\cite{8} \cite{5}\cite{6}. But it is necessary to give suitable standard in order to get suitable code.

The common technique in these approach is taking into consideration elements which constitute a program, such as expressions and a statements. But these approaches are very difficult. It is because the dependance analysis used as the base of such reuse technology is very difficult.

5. CONCLUDING REMARKS

We proposed an evolved library where library functions are accompanied by their usage patterns, called idiom. Calling a function in an evolved library, a common reliable usage of the function is always available. A user is encouraged to use the library function following the “good” idiom. As the result, an evolved library contribute to improve the reliability of application software. It also promote reuse of software components since "good" idiom are tend to be used repeatedly. In order to find out the “good” class of idiom, we investigated the existing open-source software that has the history long enough to be considered reliable. We built the FCDG database and from the database, we extracted partial reliable structures. We built the evolved library of system functions in FreeBSD 4.3-Release and 4.4-Release to justify that our idiom improve software with respect to reliability by applying the evolved library to the open-source software.

Most of the cost in generating an evolved library is spent on matching partial FCDGs. To generate the evolved library for 212 programs of FreeBSD 4.3-Release, it took nearly 12 hours by a FreeBSD PC with a single Pentium III 1GHz processor and 256M byte memory. In this report, we have been focusing on the evolved library generation. It is essential to use more elaborate graph-matching algorithm in order to maintain the scalability.

We showed the idiom in the previous version appear as the substructure of the new idiom in the later version. This means that our method promote the software reuse without decreasing the reliability in the evolving process.

For the future work, firstly it is needed to analyse the cross-function dependency. Our dependency analyser deals with only a single function. By analysing the cross-function dependency it is able to capture some undesirable idiom. For instance, a file descriptor is passed to a non-library function and in the function it is closed. In many cases such a cross-function dependency is created by the ad-hoc coding and violates the modularity. It should be useful to point out such unwanted dependency if such usages appear frequently. By distinguishing the cross-function dependency, it is expected to have more reliability.

In our method, the size and quantity parameters are the crucial keys in generating idiom. It is important to set appropriate parameters to get better reusability and less maintainance cost. By generating evolved libraries for many kinds of software, we continue to investigate the correlation between the software scale and those parameters.

6. REFERENCES