# COSMOPEN: A Reverse-Engineering Tool for Complex Open-Source Architectures

François Taïani
*LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex 4, France*
*francois.taiani@laas.fr*



This article will be presented at the student forum of the *International Conference on Dependable Systems and Networks* (DSN'2003) to be held in San Francisco (CA) in June 22nd - 25th, 2003. It will be published in the supplemental volume of the proceedings of the aforementioned conference.

# COSMOPEN: A Reverse-Engineering Tool for Complex Open-Source Architectures

François Taïani
*LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex 4, France*
*francois.taiani@laas.fr*

## 1. Introduction and Motivation[1]

Third party software components (i.e. Commercials Off the Shelf (COTS) and Free and Open-Source Software (FOSS)) are now increasingly used in infrastructure systems and applications with high dependability requirements [1, 2]. Dependability in general, and the implementation of fault-tolerance in particular, requires a comprehensive analysis of the considered systems. For systems that are built out multiple components this means a thorough understanding of components internal behavior, and inter-component interactions, in particular for mastering state entangling, causality tracking, non-determinism, and state capture/ restoration.

In previous publications we have proposed a conceptual framework [3] and a design approach [4] to address this issue and help implement fault-tolerant mechanisms independently from the system nominal services (separation of concerns) in a principled and disciplined manner. We have called our approach *Multi-Layer Reflection* as it leverages the experience gathered on reflective fault-tolerant systems, and adapts it to the particular context of complex component-based software architectures.

As a support to this work, we have developed a prototype suite of reverse engineering tools named *CosmOpen* (*Comprehensive Open Source MOdeling & Patternizing ENvironement*) that explicitly focuses on the needs that we identified as necessary for Multi-Layer Reflection. We used this suite to validate our approach and understand the complex relationships of some component-based multi-layer systems.

In this paper we sketch some of the dynamic analysis capabilities of *CosmOpen* and explain their relevance for the development of generic fault-tolerant mechanisms in complex software systems.

## 2. Tool Overview

*CosmOpen* follows the architectural guidelines proposed in [5] for the manipulation of large source repositories: (i) It extracts raw information from the

```
Breakpoint 1, ClassA::callBackA (this=0xbffff987) at main-example.cpp:46
 (gdb) bt
#0  ClassA::callBackA (this=0xbffff987) at main-example.cpp:46
#1  0x08048616 in ClassB::methodB (this=0xbffff986, aClassA=0xbffff987)
    at main-example.cpp:49
#2  0x0804865c in ClassA::methodA (this=0xbffff987, aClassB=@0xbffff986)
    at main-example.cpp:45
#3  0x08048636 in main () at main-example.cpp:54
#4  0x4009ca51 in __libc_start_main () from /lib/libc.so.6
```

**Figure 1: Debugger output**

```
<traceSet threadCreation="::clone">
  <trace thread="" >
    <call entity="ClassA" method="callBackA" this="0xbffff987" />
    <call entity="ClassB" method="methodB" this="0xbffff986" />
    <call entity="ClassA" method="methodA" this="0xbffff987" />
    <call entity="" method="main" this="" />
    <call entity="" method="__libc_start_main" this="" />
  </trace>
</traceSet>
```

**Figure 2: The corresponding XML Representation**

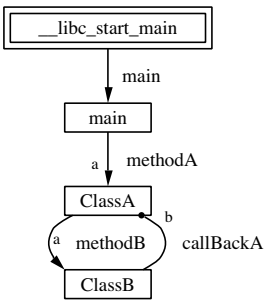


**Figure 3: Graph Representation of a Captured Trace**

[1] : This work was partially supported by the European IST Project DSoS "Dependable Systems of Systems" n°1999-11585.

observation of a program. (ii) It stores this raw information in an XML based repository. (iii) It provides abstraction operators to extract patterns and structures from the repository. (iv) The obtained information is viewed using an external viewer (in our case `dot` from AT&T [9]).

*CosmOpen* adapts these guidelines to the extraction of dynamic information: We implemented an event extractor using the method proposed in [6, 7]. We use the observation capabilities provided by common debuggers, like for instance `gdb` [8], to record the detailed execution of a program through breakpoints and stack introspection. Figure 1 shows a typical output of the `gdb` debugger for a very small C++ program with a breakpoint set on the method `ClassA::callBackA`. This output can very easily be transformed into an XML representation (Figure 2), and can be represented in a graph notation (Figure 3). (We could also obtain an UML object interaction diagram from the same XML representation.)

This method is straightforward. However, the profusion and completeness of the captured information very rapidly exceed human capabilities (as explained in [5]), even for very small programs. The need for "abstraction operators" have been acknowledged for many years now by the software engineering community, and our tool is certainly not new in that respect. The relevance of the chosen operators determines however to a large extent the usability of the tool. In the next section, we present some of the operators we have implemented using a small example.

## 3. Multi-Level Abstraction Operators

In this section we briefly describe two of the operators we have defined to obtain a more abstract representation of the brute observation of a program's execution.

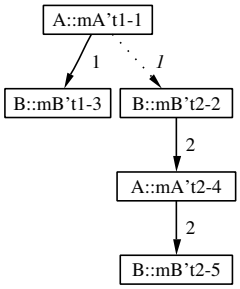


**Figure 4: Before the `condense` Operation**

The `condense` operator folds together a set of invocation trees into an interaction diagram by grouping together the invocations made on a same class. This approach leverages the object-oriented structure of the program, if present. Its application to the graph example of Figure 4 is given in Figure 5.

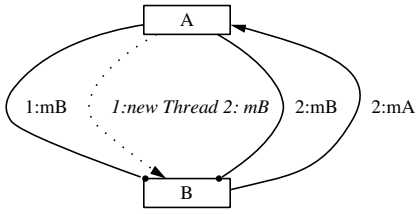


**Figure 5: After the `condense` Operation**

The `abstractAway` operator allows the removal of a set of invocations from the graph whereas keeping trace of the call sequences in which those invocations are embedded. Its application to the graph example of Figure 6 is given in Figure 7, where invocations on class B have been removed. Such an operation is very useful to hide some details of the execution flow and identify more abstract cooperation patterns between the remaining classes.

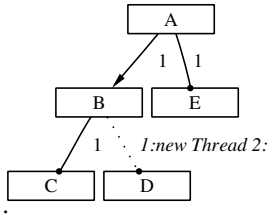


**Figure 6: Before the `abstractAway` Operation**

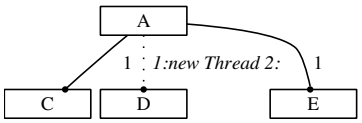


**Figure 7: After the `abstractAway` Operation**

## 4. A Small Example

We now illustrate on a small example how the use of the several transformation operators of our tool, two of which were presented in the previous section, can be used to obtain high level views of a program execution. Figure 8 shows the results of the capture of the execution of a small multi-threaded C++ program (68 lines). It includes a lot of useless details that prevent a clear understanding of the program behavior. The graph is unreadable and understandable as such.

The application of abstraction operators, among which those briefly described in Section 3, enables a new representation to be obtained, forgetting those useless details. The notion of useless detail depends on the target objectives, i.e. the notions that we want to make visible. Figure 9 shows the output of this abstraction process. This graph shows the links at runtime between classes, threads

and synchronization operations only. It is clearly more understandable and very useful for mastering objects and threads interleaving at runtime.

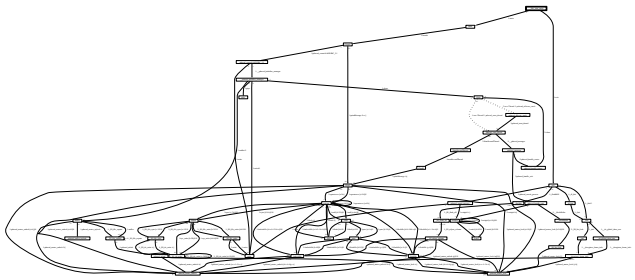

**Figure 8: Interaction Graph for a small C++ Program!!!**

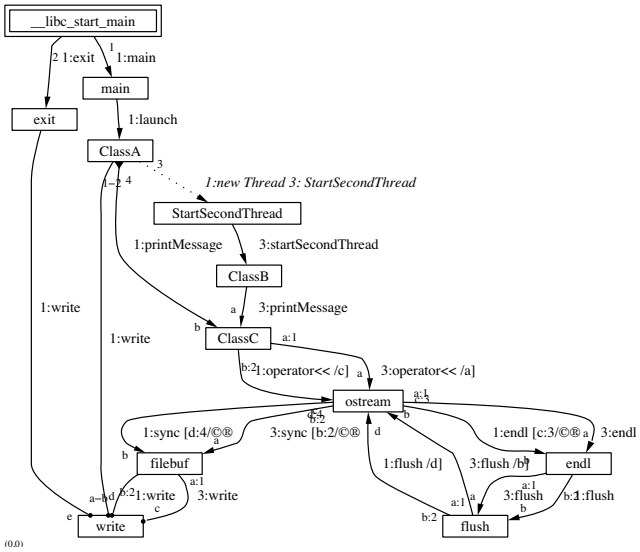


**Figure 9. Filtering out the pthread library, some of the IO file subsystem and Inter Process Communication**

## 5. Conclusion

The development of reverse engineering tools is not novel and our work was based on key developments done in the past and off-the-shelf tools. The way the raw extracted information is analyzed and presented in a comprehensive form for later use may vary according to the target objective. The aim of this work was to address the need for extracting metamodels from open source components to be able to master multi-components systems from a dependability viewpoint. This is a crucial issue (even with open-source components) due to the complexity of today systems and the need for an efficient and adaptive implementation of fault tolerance strategies.

In complex systems, reverse engineering is very valuable to establish metamodels and thus enforce separation of concerns of fault-tolerance.

To this aim, many concepts and approaches already proposed can be reused, but specific abstractors are needed to focus on inter-level relationships, causal dependencies, and state related issues. The benefits of this tool were directly used in the development a fault tolerant distributed platform developed at LAAS and based on reflective computing concepts.

## 6. References

[1] Stolper, S.A., *Streamlined Design Approach Lands Mars Pathfinder*. IEEE Software, 1999. **16**(5 (September/October)): p. 52-62.

[2] The MITRE Corporation for The Defense Information Systems Agency, *Use of Free and Open-Source Software (FOSS) in the U.S. Department of Defense*. Technical Report MP 02 W0000101 (Version 1.2.04), 2003, The MITRE Corporation. 168 pages.

[3] Taïani, F., J.-C. Fabre, and M.-O. Killijian. *Principles of Multi-Level Reflection for Fault-Tolerant Architectures*. in *2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*. 2002. Tsukuba (Japan). p. 59-66.

[4] Taïani, F., J.-C. Fabre, and M.-O. Killijian. *Towards Implementing Multi-Layer Reflection for Fault-Tolerance*. in *The International Conference on Dependable Systems and Networks (DSN-2003)*. 2003. San Francisco, CA: IEEE Computer Society.

[5] Chen, Y.-F.R., *et al. Ciao: a graphical navigator for software and document repositories*. in *International Conference on Software Maintenance*. 1995. Opio (Nice), France. p. 66-75.

[6] Systä, T., *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, Ph.D. Thesis, Tampere University, Faculty of Economics and Administration / Department of Computer Science, Tampere (Finland), 2000, 232 pages.

[7] Reiss, S.P. and M. Renieris. *Generating Java trace data*. in *Java Grande Conference (ACM 2000 conference on Java Grande)*. 2000. San Francisco, CA, USA: ACM. p. 71-77.

[8] Stallman, R. and R.H. Pesch, *Debugging with GDB*. 9th ed. 2002, Boston, MA, USA: The Free Software Foundation.

[9] Gansner, E.R. and S.C. North, *An open graph visualization system and its applications to software engineering*. Software - Practice and Experience (SPE), 2000. **30**(11): p. 1203-1233.