Reverse Engineering a Reverse Engineering Format: An Experience Report and Some Modest Reflections

Holger M. Kienle
Mälardalen University, Västerås, Sweden
hkienle@acm.org

Abstract

This paper discusses my experiences in reusing an existing fact base (provided as a file encoded in the MSE exchange format and adhering to the FAMIX meta-model) for a reverse engineering effort with the Rigi graph visualization tool. This experience is an instance of a reuse scenario that is fairly typical for reverse engineering: Opportunistic grabbing of facts for subsequent transformation, analysis and visualization.

On the positive side, it was indeed feasible with relative ease to reuse the existing fact base to obtain relevant information to produce visualizations with Rigi. As such, our existing tooling and techniques have a positive effect on reuse. On the negative side, this reuse was not straightforward, effectively requiring to "reverse engineer" the fact base itself in parallel while trying to reuse it. Thus, this experience leads to the questions: Can we—as the reverse engineering community—come up with techniques that in the best case enable reuse out-of-the-box? And what are the steps that we could take towards this vision?

1. Introduction and Background

Each reverse engineering effort requires the collection of facts about the subject system. Obtaining these facts from the source code of the subject system is often a time-consuming and error-prone activity. Furthermore, from a research perspective obtaining these facts is typically only a necessary prerequisite to commence the actual, value-adding research activity.

An attractive alternative is to short-cut the fact extraction step via reusing of existing facts of the subject system that are generated and provided by third-parties (e.g., fellow researchers). In order to process such facts, they have to adhere to a meta-model (or schema) and are encoded in an exchange format. FAMIX is an example of a meta-model that can be used to describe facts from programs written in a number of object-oriented languages. MSE is an exchange format to encode FAMIX models for storage and import/export.

In this paper we report on our experiences on leveraging a pre-existing FAMIX model encoded with MSE with the goal to obtain architectural views in the Rigi graph visualizer [3]. In our particular case, the subject system is Azureus[1] version 2.5.0.0, a BitTorrent client written in Java. The reverse engineering of this system was motivated by the *Tool Demo Challenge*, which was part of the IEEE VISSOFT 2007 conference.[2]

For the tool demo challenge, we had to perform two main tasks. First, relevant facts from the Azureus system had to be encoded in Rigi's RSF format. Second, the information from the RSF file had to be manipulated in Rigi in order to produce suitable architectural views of Azureus (in the form of dependency relations at the class and package level).

Our analysis with Rigi is described elsewhere [4]; in this paper we focus on our experiences in obtaining the facts.

2. Reverse Engineering MSE

The first step of each reverse engineering effort is to obtain relevant facts from the subject system. Azureus is not a small system, composed of 4222 classes and more than 300,000 lines of Java code.

In our case, we considered several possibilities. Perhaps the most typical approach is to use Azureus' Java source code as the starting point to obtain facts. In this case, an extractor could be written from scratch, or an existing extractor could be re-used that produces facts in RSF such as FrontEndART's SourceAudit.[3] The effort of writing a Java extractor from scratch was perceived as prohibitive—even though such an effort can be significantly reduced if

[1]Azureus is now called Vuze, but still accessible at http://azureus.sourceforge.net/

[2]http://www.program-comprehension.org/vissoft07/VISSOFT2007_ToolDemo.html

[3]http://www.frontendart.com/

a suitable (compiler) front-end (e.g., EDG's JFE or Source-Navigator), or a lightweight extraction technique (e.g., island parsing) is employed. Using an existing extractor may cause problems if the extractor is difficult to install or fails to run through. As another alternative, existing facts about the system may be used if they have been made available from other researchers. In our case, the organizers of the tool demo challenge indeed provided facts of Azureus encoded as both MSE and XMI [10].

Considering the alternatives, we decided to use an existing fact base with the expectation to minimize our time and effort. As a side effect, this allowed us to explore the idea of leveraging the work of other researchers with the help of exchange formats. Indeed, enabling technology for sharing of fact bases among researches is actively discussed and pursued in the reverse engineering field (e.g., in the WCRE 2000 Data Exchange Format session [8] [6] and the Dagstuhl Seminar 01041 on Interoperability of Reengineering Tools [5]).

Since MSE is a textual format, we can manipulate and understand it easily with a standard text editor and Unix tools, as opposed to XMI. Since RSF is a textual format as well, we felt more comfortable in manipulating it than XMI. For example, it was simple for us to create a small MSE test file via copy-and-paste of fragments from the Azureus MSE file. The Azureus MSE file has 1,926,971 lines and hence is too big for effective experimentation.

With the chosen approach, we do not have to perform fact extraction per se. However, we had to write a translator instead that reads in the Azureus MSE file and encodes the facts in RSF. Perhaps ironically, this means that we have to "reverse engineer" the meaning of the Azureus MSE file first. While reverse engineering the MSE format, we started to develop a translator that transforms the information contained in the MSE file to RSF. The translator is written in Perl and has about 500 lines of code; it was written in about two days and tested only on the Azureus MSE file. Thus, we settled for the cheapest goal: Translation of one particular MSE model instance to a meaningful RSF model.

At first glance, it may seems strange that we had to "reverse engineer" the MSE file, since we have rather detailed information about the MSE file format as well as its schema. The syntax of MSE is documented with a grammar,[4] and the meaning of the encoded information is described in the FAMIX 2.x schema[5] and Java language plug-in 1.0[6] documents. However, as the subsequent discussion shows, there are still a number of unanswered questions that need to be resolved.

[4]http://smallwiki.unibe.ch/fame/msespecification/

[5]http://scg.unibe.ch/archive/famoos/FAMIX/

[6]http://scg.unibe.ch/archive/famoos/FAMIX/Plugins/JavaPlugin1.0.html

Level	Entities
1	`Class`, `InheritanceDefinition`, `Function`, `Method`, `Namespace`
2	`Attribute`
3	`Access`, `Invocation`
4	`FormalParameter`, `LocalVariable`

Figure 1. Levels of extraction in FAMIX

For example, we need to know which kinds of facts are contained within the file. We found the answer by running a simple Unix query

```
grep "(FAMIX." Azureus_2.5.0.0.mse |
sort | uniq -c
```

over the MSE file, which told us that the following entities exist: `Access` (77,872 occurrences), `Attribute` (9,383), `Class` (4,740), `FormalParameter` (21,148), `Function` (1), `InheritanceDefinition` (6,974), `Invocation` (66,666), `LocalVariable` (23,712), `Method` (24,666), and `Namespace` (418).

The single occurrence of a `Function` entity sticks out as unusual. It happens to be an `unknown_function` that is not referenced by any other entity in the file and thus can be safely ignored. The meaning of this function cannot be found in the documentation.

The Java FAMIX schema does not prescribe which facts have to be produced by an extractor; as a result, the file can contain a subset of the full schema. FAMIX defines four levels of extraction with increasing detail. Figure 1 shows which entities correspond to which level. (`Namespace` presumably is identical to `Package`, which is mentioned in the documentation.) Based on the kinds of facts that we found, our fact base adheres to level 4, but we only use the subset up to and including level 3.

We also inspected the MSE file to understand attribute values of entities. In the following we give several examples of valuable information and anomalies that we reverse engineered. To our knowledge, not all of these are documented.

- The `fileName` attribute can have the value `'_UNKNOWN_PATH_/_UNKNOWN_FILE_'`. In this case, the absence of valid `startLine` and `endLine` information is encode with `-1`.
- For anonymous inner classes (e.g., `AzureusCoreImpl$11`), the provided line number information is meaningless.

- The `name` of a `Namespace` is encoded with `::` as separator (e.g., `org::bouncycastle::crypto`).
- Examples of special `name` values are `_GLOBAL-_NAMESPACE_` for primitive Java types, and `_UNKNOWN_NAMESPACE_` for names that cannot be resolved. The latter case occurs in the MSE file because Azureus uses Eclipse SWT, which it seems was not available to the extractor when the MSE file was produced. As a result, the MSE file does not show this dependency at all.
- The `Method` entity has no line information attributes.
- If the `fileName` attribute is valid, it contains an absolute path

  ```
  (sourceAnchor
  'FILE:/home/marco/case-studies/-
  azureus/Azureus_2.5.0.0/...')
  ```

  that needs to be pruned.
- The `invokes` attribute of the `Invocation` entity can be `'<unknownConstructor>()'` (873 occurrences) and `'<unknownMethod>()'` (3116). If this is the case, the `candidate` attribute references an entity that does not exist.
- There are invalid `candidate` references for 75 invocations. These references seem to be all constructor calls with "new" and (indirectly) depend on the Eclipse SWT.
- The `Class` and `Method` attributes contain a number of attributes for metrics that are all set to `0.0`.

Note that the above findings stem from a single model instance. Further instances might reveal additional issues.

3. Discussion

As our experiences point out, reusing the existing MSE fact base required an ad-hoc reverse engineering effort to detect

- undocumented constraints and conventions of the fact base and schema. Especially, interdependencies among attributes (e.g., if an attribute has a certain value than another attribute is invalid) are often not made explicit.
- bugs or unexpected "anomalies" of the fact base. These are often the result of implementation decisions in the extractor for "gray" areas in the specifications.
- the "boundary" of the provided facts in terms of omitted libraries, etc. This boundary is often not easily determined.

It is crucial to understand that these shortcoming are not specific to MSE and FAMIX and could not have been avoided by using other tools. In fact, MSE and FAMIX are exceptionally well documented and maintained with a rigor that is far above the typical standard that can be found in (academic) reverse engineering tools.

Looking at this "reverse engineering" reverse engineering experience—which is only an instance of many others—there is this nagging question: Even though the software engineering community in general and the reverse engineering community in particular have come up with clever ideas that facilitate the exchange of information and interoperability among tools,[7] why are there still these rather mundane stumbling blocks when we do it for real?

So, what does this experience tells us? That we cannot do (much) better in enabling reuse of existing fact bases for reverse engineering research? Hopefully not. Some ideas that may help to tackle this problem are

Model-based development: If the constraints of a meta-model are formally encoded (e.g., with the help of a (graphical or textual) domain-specific language) it should be easier to reason about it for both the end-user and automatic tools. This should also enable tools for checking if constraints are met.

An extremely simple yet very effect example in our case would have been a checker that makes sure that there are not dangling references to entities in a fact base. Also, there could be tool support that allows reverse engineers to easily check hypotheses about the fact base (e.g., of the form "if attribute a_1 has property p_1 then attribute a_2 must have property p_2").

A natural-language specification of an exchange format or schema could embed formal assertions that could be checked automatically on a test corpus (see next point).

Applying basic (software engineering) principles: When developing our schemas and tools we should invest more effort into supporting infrastructure such as test cases that check the integrity of a large fact base corpus. Such test cases could be realized with the help of checkers that specify constraints in a formal manner (see previous point). This would allow, for example, meaningful regression testing.

Your thoughts and idea here! The discussed problem can only be tackled by smart people in our research community—that means: you.

[7]To give a few examples of related work in tool interoperability: [9] [11] [12] [1] [7] [2].

References

[1] L. Davis, R. F. Gamble, and J. Payton. The impact of component architecture on interoperability. *Journal of Systems and Software*, 61(1):31–45, March 2002.

[2] Alexander Egyed, Sven Johann, and Robert Balzer. Data and state synchronicity problems while integrating COTS software into systems. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 69–74, May 2004.

[3] Holger M. Kienle and Hausi A. Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247–263, April 2010.

[4] Holger M. Kienle, Hausi A. Müller, and Johannes Martin. Dependencies analysis of Azureus with Rigi: Tool demo challenge. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07)*, pages 159–160, June 2007.

[5] Timothy C. Lethbridge. Report from the Dagstuhl seminar on interoperability of reengineering tools. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, page 119, May 2001.

[6] Stephen Perelgut. The case for a single data exchange format. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 281–283, November 2000.

[7] Steven P. Reiss. Simplifying data integration: The design of the desert software development environment. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 398–407, May 1996.

[8] Susan Elliott Sim. Next generation data interchange: Tool-to-tool application program interfaces. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 278–280, November 2000.

[9] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.

[10] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX and XMI. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 296–298, November 2000.

[11] Anthony I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments: International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149. Springer-Verlag, 1990.

[12] Peter Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, March 1996.