

Next Generation Data Interchange: Tool-to-Tool Application Program Interfaces

Susan Elliott Sim
Dept. of Computer Science
University of Toronto
simsuz@cs.utoronto.ca

Abstract

Data interchange in the form of a standard exchange format(SEF) is only a first step towards tool interoperability. Inter-tool communication using files is slow and cumbersome; a better approach would be an application program interface, or API, that allowed tools to communicate with each other directly. This paper argues such an AP is a logical next step that builds on the current drive towards an SEF. It presents high-level descriptions of three approaches to tool-to-tool APIs and illustrates how requirements for the SEF also apply to the API.

Keywords

standard exchange format, API, tool interoperability, library, protocol, framework

1. Introduction

As agreement is reached on a standard exchange format (SEF), we need to turn our attention to problems that lie beyond the current event horizon. These problems will exist independent of any single representation format and will continue to be interesting regardless of which one is chosen. Chief among these concerns is an API (Application Program Interface) for dynamic tool interoperability.

The primary object of an SEF is greater tool interoperability. Within the reverse engineering community, we have reached a level of maturity where we have recognised the benefits of making our tools work together more effectively. However, the current drive towards an SEF has focussed on *data* interchange through *files*. This interaction model essentially requires tools to read and write to disk to communicate with each other. Such delays are cumbersome and do not encourage interactive experimentation to recover high-level structures in legacy systems.

Tool interoperability can be characterised using three levels.

Level 0 Ad hoc interaction.

Level 1 Static interaction using an SEF and disk files.

Level 2 Dynamic interaction using an API to share data and to invoke other tools.

We are currently at level 0, where tools are pieced together using data converters and scripts. Furthermore, these helper programs become out-of-date as soon as one tool changes. An SEF will allow us to achieve level 1 and reduce the number of data converters needed. Adding an API¹ would take us to level 2 and reduce the brittleness of commands or scripts issued between tools. Progress towards this level depends on having a stable and operational SEF. Learning to communicate data is a necessary pre-condition to learning to communicate operations on data.

This paper looks at the problem of how to make tools communicate with each other directly. Such an API is a logical next step and results from the current SEF are also applicable to its development. Section two gives high-level descriptions of three possible approaches for the API. Having illustrated how an API might operate, section three shows that the requirements for the SEF transfer to the API as well.

2. Approaches for Dynamic Tool Interoperability

There are three basic approaches, or architectures, for dynamic tools-to-tool communication. [6] The first approach is to encapsulate each software tool and have it operate as a library. The second approach would be to add a communication protocol to an existing SEF. Finally, these two approaches could be combined to create a set of federated tools that agreed to a protocol

¹ It should be noted that an API is under discussion as part of the current SEF effort and it is primarily concerned with facilitating access to data stored in the SEF. It will be referred to as "data-API" to distinguish it from the API that is focus of this paper.

for intra- and inter-tool behaviour. Each of these approaches will be discussed in greater detail to provide a basis for discussing requirements of an API in Section 3.

Libraries

Each tool developer could make an encapsulated version of their tool available. This version could be invoked like a library from other tools, using a set of pre-defined commands or API, on a common data structure, or SEF. The API would include rules on modifications and their data requirements.

The advantage of this approach is speed. Tools calling each other in this fashion would not have to write and re-read data from disk. Nor would they even have to copy memory pages. They could simply call commands on other tools and pass memory references. The disadvantage of this approach is that some tools are not designed to be used as libraries. For instance, they might have command-line switches or a graphical user interface. Also, the resulting system would be tightly integrated and sensitive to changes. Researchers may also be reluctant to maintain another version of their tools.

Communication Protocol for SEF

This approach would allow stand-alone tools to interoperate. The underlying architecture could be peer-to-peer or client-server, and would likely use sockets to communicate. Existing technologies such as CORBA or Enterprise Java Beans could be applied here. Woods et al. have proposed a client-server architecture, CORUM, for this purpose.[7] The API would consist of a set of commands (operations on data) and a protocol for using the SEF.

The advantage of this approach is it compatible with current work. It would leverage the data-API being designed for use with the SEF and each tool would still be an independent entity. Using sockets to communicate would allow tools to be run on different operating platforms, thus reducing compatibility issues. A disadvantage of this approach is that data synchronisation between tools would be a major issue.

Hybrid: A Framework for Federated Tools

The framework approach takes elements from both of the previous approaches and combines them. Tools would have to be encapsulated in the sense that they follow a template prescribed by the framework and they would communicate using protocol. The interaction template would include basic commands (ones that every tool would be required to handle), a mechanism for accepting and returning data, and configuration

commands. It would also include standards for how each tool should behave in terms of error handling, blocking, and the like.

Tools co-operating in a framework would result in a smoother end-user experience, thus addressing the original problem of shortening the length of discovery cycles. It does require tool developers to do more work to put an interface on their own tools, but this work is repaid with the ease that they can use other tools.

While this interaction model appears complex, its implementation need not be. For example, each group make their tool available on the Internet, say using a CGI-script. This script would receive requests and data and pass it to the reverse engineering tool. The results could then be returned using another CGI script, ftp protocol, or email. In this example, the framework would consist of a set of communication rules how to send commands and data, how to return the results, and how to deal with errors. The results may be facts in the SEF, a static graphic, or tool running in a web browser.

3. Transferring Lessons Learnt from SEFs to APIs

During the consultation and design process, we are learning about the technical aspects of designing an SEF and the organisational aspects of building a consensus—lessons that will prove valuable in the development of an API. With respect to technical issues, we are eliciting functional requirements (i.e. data schema for various tools) and non-functional requirements (i.e. criteria for success). [1, 3-5]

Virtually all of the requirements are applicable to an API, including such as language-independence, flexibility, applicability to multiple levels of abstraction, and scalability. In the previous section, three possible approaches to an API were given. With this background, the most relevant requirements will be discussed here.

- The format should be simple and lightweight. As with the SEF, the API needs to be easy to understand and use in order to promote adoption. Furthermore, the time and space requirements will be minimised with a simple, lightweight standard.
- Every entity must have a unique identifier. By giving every element in the software system a unique identifier, we would be able to address each one independently. The intention for SEFs was that this unique identifier could be used in the same manner as an index key in a database or repository.

- It should be possible to incrementally add data to the repository.

Since tools would be able to build their internal repositories incrementally, they need not share the entire database up front. Instead, they can ask for only relevant facts, as they need them using unique identifiers and relations between entities.

- There needs to be a library for accessing and analysing the SEF.

While this library, or data-API is not strictly part of the SEF design, it would facilitate adoption. Furthermore, this data-API could be used within the API to manipulate and manage the SEF.

Some requirements are not directly applicable, but have analogues for the API, such as the following.

- It should be extensible, allowing users to define new schemas for the facts stored in the format as needed.

An API does not need to be extensible with respect to the schemata it represents, but the operations that it allows. It should be easy to extend the API to add new analyses to the reverse engineering tool kit as they become available. As a corollary to extensibility, it should be easy to de-couple a particular tool from the API. In other words, tools need to retain the ability to run independently.

4. Conclusion

An SEF would be a major step forward in for the reverse engineering community. The most significant benefit of an SEF is greater tool interoperability between tools from both research and industry. However, file-based data interchange should not be viewed as the pinnacle of tool interoperability. More can be done to enable tools to work together directly, such as through an API. In this paper, we illustrated three possible approaches for designing an API and we showed how the development of an API flows out of our current work with SEFs. We need not design this API simultaneously with the SEF. Nor do we need to have any approach in mind when designing the SEF. We need only be aware of the API as a desirable next step, in order to avoid closing off possibilities.

At time of writing, we have agreement that the SEF should:

- use XML to encode data;
- use typed, attributed graphs as the conceptual data model;
- have a syntax that permits multiple schemata; and
- transmit the schema along with instance data.

We have early indications that GXL (Graph eXchange Language) will become the SEF.[2] Part of the current effort includes work on a data-API for manipulating data stored in this format. These developments bode well for the design of a next generation standard, a tool-to-tool API. Progress along these lines indicates we are on a path that will lead us to greater tool interoperability beyond data interchange.

Acknowledgements

Thanks to Jeff Elliott, Jeff Turnham, and Dave McKnight for the discussions. They are responsible for the good ideas in this paper, while the mistakes are all mine.

Bibliography

- [1] I. T. Bowman, M. W. Godfrey, and R. C. Holt, "Connecting Architecture Reconstruction Frameworks," *Journal of Information and Software Technology*, vol. 42, pp. 93-104, 1999.
- [2] R. C. Holt, Andy Schürr, and A. Winter, "GXL: Toward a Standard Exchange Format," University of Koblenz, Koblenz, Germany RR-1-2000, 2000.
- [3] R. Koschke, J.-F. Girard, and M. Würthner, "An Intermediate Representation for Integrating Reverse Engineering Analyses," presented at Working Conference on Reverse Engineering, Honolulu, HI, 1998.
- [4] H. A. Müller, "Criteria for Success of an Exchange Format," presented at Workshop meeting, CASCON98, Toronto, Canada, 1998. Available at: http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/criteria_muller.html
- [5] S. Rugaber and L. Wills, "Position Paper on Research Infrastructure for Reengineering," presented at International Workshop on Program Comprehension, Dearborn, MI, 1997.
- [6] P. Wegner, "Interoperability," *ACM Computing Surveys*, vol. 28, pp. 285-287, 1996.
- [7] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici, "An Architecture for Interoperable Program Understanding Tools," presented at International Workshop on Program Comprehension, Ischia, Italy, 1998.