Migrating Bauhaus from IML to SKilL

Timm Felden Martin Wittiger

University of Stuttgart, Institute of Software Technology
Universitaetsstr. 38, 70569 Stuttgart, Germany
{feldentm,wittigmn}@informatik.uni-stuttgart.de

Abstract

In this paper we will motivate and discuss the ongoing process of migrating the Bauhaus toolchain from its home-brew intermediate representation (IR) to a generalized IR based on SKilL.

1 Introduction

Bauhaus is a suite of static analysis tools [5] developed at the University of Stuttgart. It is mainly written in Ada and analyses C, C++ and Java. The current intermediate representation of analysis results is called IML. A specification of data structures is used to generate Ada code that serializes data and provides the user with a suitable API. Bauhaus developers are dissatisfied with IML and have asked for its replacement. This paper describes our efforts to improve serialization in Bauhaus.

Requirements

Because students regularly refuse thesis projects involving languages they have never worked with before, the development of Bauhaus lost momentum after the university switched the primarily taught programming language from Ada to Java. This leads to the requirement of achieving language independence while keeping legacy code largely intact. In consequence, we have to keep our API stable. In the current setup, changing the specification often led to incompatibilities that prevented loading files created with older versions. The successor needs to avoid this problem where possible. IML is just efficient enough for the domain of static program analysis. Hence, a successor should neither increase file sizes nor decoding time.

2 Steps of Migration

Presented with these requirements, we considered different serialization technologies. After taking into account a comparison of file formats for serializing data from static analysis [3] we are disinclined to use XML. We decided to use SKilL[2], which was designed to fulfill requirements similar to ours.

Thus, we decided to create an export tool that transforms a whole IML file to semantically equivalent SKilL. In consequence, we can keep all tools unmodified and have an extra tool for the conversion between file formats. A downside of this approach is the time spent converting files. Furthermore, IML would stay as it is and the overall maintenance effort would increase because two different and, most likely diverging, intermediate representations would have to be maintained. Also, the SKilL-based intermediate representation can differ from IML in details.

2.1 Creating a Specification and Binding

About five man-days sufficed to create a total translation of all specified Bauhaus type definitions. We also added type definitions for all hand-written serialization code that we could find.

In the process of creating SKilL implementations the SKilL infrastructure saw several improvements. The specification language was extended by enums, typedefs and interfaces to simplify the specification. The code generators were extended by a back-end that generates only specification, so that the IR specification can easily be navigated. The generated implementations were improved in terms of escaping names, generated documentation, compile time and better handling of name conflicts between standard library types and specified types. The latter is an issue that almost all projects will face that move their IR from one specific language to a set of languages because there is no naming convention between programming languages for standard library types.

The export tool had to be implemented in Ada because the only complete IML API is written in Ada. Fortunately, both IML and SKilL offer a reflective API, which allows the export tool to be written in just about 550 LoC. Also, reflection could be used to effectively detect bugs in the specification and translation of unspecified parts of the IML because mismatches can be observed while matching the runtime type systems. Our completely unoptimized reflection-based approach suffices to convert our test files in about 200 seconds. In comparison, parsing these files completely requires only about 40 seconds. Profiling the export tool indicates that IML reflection is taking up most of the CPU time. While it is certainly possible to write a solution without reflection, we want to focus on our new capabilities for now.

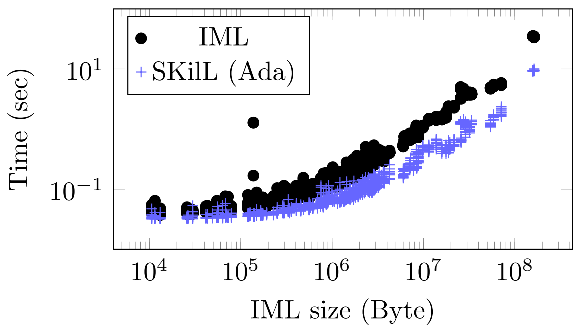

Figure 1: Execution time for finding all names of functions with implementations in an IR file. Performed on a complete specification with complete deserialization. 10 repetitions on cached data executed on an Intel i5-2540M (4 Core, 2.6 GHz).

2.2 Immediate Consequences

After finishing the export, we started immediately to profit from SKilL's benefits. Our first step was to reimplement a very simple tool using SKilL. That way, we could ensure that data is exported correctly. Also, we measured the runtime of the reimplementation to verify that our requirements have actually been met (Figure 1). To our surprise, SKilL scales better in terms of file size and serialization speeds than IML.

As the SKilL specification will remain unchanged while maintaining the existing IML implementation, the exported files are identical to files that would have been created if SKilL were used in Bauhaus directly. Thus, we already have test files to check the integrity of a future Bauhaus with IML replaced by SKilL. Furthermore, we have mitigated signs of software aging [4] in Bauhaus. SKilL generates a fully documented API for our exported IR in any implemented language. This is not only an improvement in Ada. It is also possible to attract students with the promise that they could freely choose the programming language for their thesis. For example, we could convince a student to extend Bauhaus by further pointer analysis implementations because he was allowed to use Scala.

Comparison of Code Size

Table 1 shows that the IML representation requires a significant amount of code. Given our limited human resources, it is obviously not an option to perform manual translation steps on a significant part of IML. Furthermore, there are over 2 million other LoC in Bauhaus, thus it is not an option to change the IML API in a way that would prevent compilation of existing tools.

A summary of generated code sizes for SKilL is given in table 1 to contrast the 487k LoC of IML. The *doxygen* back-end is just a rewrite of the specification that is not even type correct C++, but can be parsed by the doxygen parser to generate an interactive documentation. The *Ada* and *Java* back-ends use similar architectures. The excess code required for Ada is caused by duplication in header and source files. Additionally, Ada requires manual memory management, manual boxing, manual reflection and more verbose type conversions. The expressiveness of semi-functional Scala enables a concise architecture and relatively small implementation size.

Back-end	files	comment	code
Ada	1280	18670	288671
C++	87	8382	110417
doxygen	337	4316	3138
IML	2293	13227	487829
Java	1177	29747	131994
Scala	16	7239	53491

Table 1: Generated code for IML and SKilL back-ends

Another benefit of SKilL is that tools working only on a subset of the IR can be built against a reduced IR specification. For instance, there is a tool printing the names of implemented functions. If the specification is stripped to the minimum, the resulting amount of code is reduced to roughly a thousand lines.

3 Complete IML Replacement

Similar to the export tool one could also write an import tool building IML files from SKilL. The appealing aspect of this approach is that it would allow us to use SKilL's change-tolerance to convert between different versions of IML. On the other hand, having to convert between IML and SKilL several times on the path from the front-end to the desired analysis is reduces usability and increases total runtime.

Thus, the remaining goal is to have a total conversion of Bauhaus to SKilL. First, we will create a fully automated translation for all type definitions that are specified using the IML specification language. After that, the generator for contemporary IML serialization will be changed to create implementations that internally use the SKilL API instead of directly writing objects into a stream. That way we will soon be able to serialize instances of specified types. This strategy will allow us to keep our legacy without losing the ability to extend it effectively.

References

[1] SKilL on Github. `https://github.com/skill-lang/skill`.

[2] Timm Felden. Efficient and Change-Tolerant Serialization for Program Analysis. In: *Softwaretechnik-Trends*, 2014.

[3] Martin Kaistra. SKilL vs. XML: Performance of Serialization-Concepts, 2015.

[4] David Lorge Parnas. Software Aging. In: *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, (pages 279–287). IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. ISBN 0-8186-5855-X.

[5] Aoun Raza, Gunther Vogel and Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: *Reliable Software Technologies – Ada-Europe 2006*, *LNCS*, volume 4006, (pages 71–82), 2006.