Evolution of a Program Analysis Toolchain

Timm Felden Felix Krause

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

Abstract

In this paper we report experiences made in the mostly finished process of modernization of the Bauhaus toolchain. We discuss the replacement of a hard-to-change intermediate representation that is only accessible in Ada by a modern one that is both language-independent and easy-to-change. Furthermore, we describe how to set up a process such that even academic research projects can perform an affordable and safe migration.

1 Introduction

Bauhaus is a toolchain with a focus on static program analysis[8]. New analyses are contributed in the form of tools resulting from master and doctoral theses. Most tools are written in Ada and analyse C, C++ and Java source code. Tools communicate over a file-based intermediate representation called IML.

After about 20 years of development, two main issues emerged that are addressed by this work. The first is that it gets increasingly harder to recruit new students for master theses. This trend started when the primarily taught programming language was switched from Ada to Java. In consequence, the average student is unable to write Ada code of sufficient quality without prior introduction. Using two months of a six months thesis for a sufficient introduction to a programming language is not only unproductive. It is also a disadvantage when competing with other projects for the best students. Therefore, we had to find a way of making the existing toolchain accessible in a language that students know well, and more importantly, like. Migrating the whole code base to Java, for instance, is not an option because it consists of roughly two million lines of Ada code. We know of no tool that can convert this code base automatically. Manual conversion is also not an option, as we do not have enough resources to convert all tools. Removing tools from the toolchain would likely deprive us of a significant part of existing functionality.

The second issue is that the existing IML infrastructure has no practicable way of changing its central specification. Though it is possible to add new type definitions to the specification, once added definitions can no longer be changed without breaking compatibility of the intermediate representation. The reason for this behaviour is the design of IML. An API and a binary file format is generated out of a specification. A change of an existing type definition is directly reflected in both. Hence, the resulting version of IML is incompatible to the prior version. In consequence, tools using that API have to be changed. Also, all existing data sets can no longer be read. Therefore, the central specification has experienced almost no changes over the last years. This leads to a serious evolutionary pressure that makes addressing this issue increasingly necessary.

Unsurprisingly, this issue is completely independent of Bauhaus which is only kept as a concrete example. A similar situation, for example, is reported by Strittmatter and Heinrich in the context of the Palladio Component Model[10].

In Bauhaus, almost all access to the intermediate representation is encapsulated in a library called libIML. Its source code is mostly generated out of a machine-readable specification. Its serialization format is based on said specification but can also contain hand-written parts. The basic architecture of a tool is shown in figure 1. Migration extends mostly to the replacement of nodes connected by dotted edges.

Now, one could simply try to replace the existing Ada source code generator by a Java counterpart. This would immediately solve the issue of accessibility of the primarily used programming language. In fact, this has been done some time ago. The problem of this approach is that the IML serialization format is mostly specified but not entirely. Actually, there are data structures whose semantics are only given by handwritten Ada code. As such, it is not possible[1] to generate equivalent serialization code for any programming language other than Ada. Hence, the generated API was realized basically by delegating API calls to the existing Ada implementation. This is achieved by enriching the Ada version of libIML with compiler directives that make all functions accessible from C. Then, code generators could easily be implemented that, based on the C API, implement APIs for C++ and Java. Consequently, a Java API for IML exists already. But it remains unused because some IML data structures cannot be expressed directly in C and, hence, are not exported to C++ and Java either. This

[1] At least not in any somehow maintainable way.

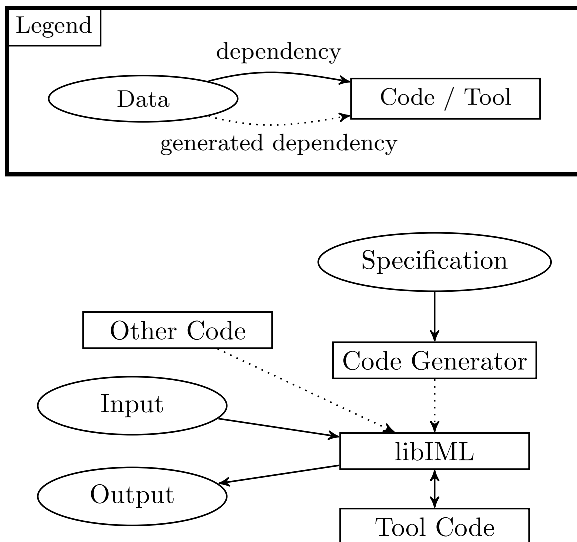

Figure 1: Abstract architecture of a tool.

is the case even though a direct translation of those data structures from Ada to Java would be possible. Having some parts of the intermediate representation inaccessible makes the development of new tools ultimately impossible.

A different approach is to replace the serialization format as a whole. Enough sufficiently powerful formats exist already that can be accessed in multiple relevant programming languages. Also, one could try to get an understanding of libIML that is so far abstracted that it makes generation for both Ada and Java possible without the requirement of hand-written code except for the realization of the original API.

In both cases, a specification for all hand-written code has to be created. Further, both solutions require an Ada binding to be implemented such that existing data sets can be converted to the new format. But there are further requirements that have to be fulfilled in order to create a practicable solution.

2 Goals

The necessity of presenting students with a programming language they have learned in the course of their education was mentioned already. Using native interfaces to export an API has not only the drawback of not allowing any evolutionary improvements of our intermediate representation. Also, it yields a remarkably poor performance. Hence, we have to consider additional side-goals when designing our migration process as a whole.

Feasibility

A set of conditions has to be considered in order to create a feasible solution. Today, tools can effortlessly analyse programs of sizes ranging from several hundred to even a million lines of C code. This property has to be maintained at all cost. Between tool executions, data is repeatedly read and written. In consequence, read and write operations have to be fast so that the whole execution time is not affected notably. The abstract process of performing an analysis is depicted in figure 2.

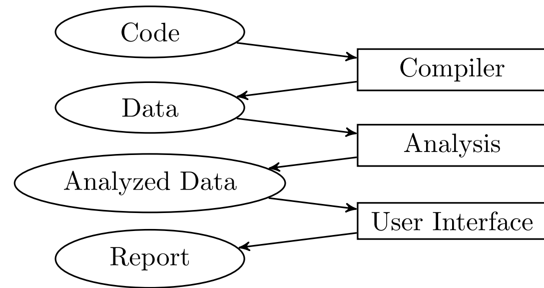

Figure 2: Abstract process of tool execution.

Furthermore, the IML specification contains around 300 type definitions. It is an object-oriented specification language. Having grown over a long time, the average level of inheritance is unusually high. Slightly simplified, one could imagine the specification to be some sort of huge UML class diagram. But, as the specification describes pure data-objects, all fields are publicly accessible and no methods exist at all. For those tools that locally define methods in the IML class-hierarchy, a generated abstract visitor (see [5]) can be extended. Also, there are some peculiarities that have to be addressed correctly in each target language. For instance, there is multiple inheritance using interfaces, which, in contrast to Java, can inherit from a regular class. Further, fields can not only be pointers to other objects or ordinary values like strings or integers. It is also required to specify arrays, lists, sets and maps. Choosing the correct type is required for a correct representation of the Ada API. The base type of any such container specification can be anything from a polymorphic pointer to a regular integer type.

In order to use libIML or an equivalent in a language like Ada without a native interface, it has to be possible to generate an implementation conforming to the specification. The generated implementation should be almost identical to the existing one to minimize the effort of adapting to the existing API. Given the sheer amount of generated code, it is also not an option to perform any manual manipulation after its generation. Such manipulations are likely to be repeated after every generation, thus hindering adaptation of specification changes. Furthermore, the generated code should be as compact as possible to keep compile time at a manageable level. Finally, the generated implementation shall export documentation contained in the specification in their public API. This will reduce the amount of training required by students, as they no longer have to deal with the original IML specification. Allowing students to mod-

ify the central specification is undesired anyway.

The graphs resulting from analyses can be used for testing. For instance, if they are given to students, they should on the one hand consist of one file only and on the other hand be rather compact. It would be particularly nice if they could be sent by an email, at least in compressed form. This would simplify the communication between team members and students.

Change Tolerance

Over the last 20 years, several design decisions were made that turned out to be wrong. Because wrong decisions cannot be prevented in general, it would be nice if it were possible to correct them at little cost. However, maintaining compatibility with data sets generated with old tools is an absolute requirement. At least if the specification change does not interfere with the part of the specification used by an old tool. This is important for two reasons. The first is that the specification of the front-end is almost never changed. Secondly, there are very valuable data stets that cannot be reproduced easily. Those are in part results of very expensive analyses, such as a precise pointer analysis which can run for several days. Another source of expensive data sets are ones whose source code became inaccessible for whatever reason. Furthermore, the amount of time required for the compilation of a million lines of C code must not be underestimated. Recompiling regularly is wasting a lot of time and will likely cause larger programs to be removed from the test data set. This can, in consequence, lead to slight degradation of scalability over time to a point where analysing the largest examples is no longer affordable.

Additionally, a change in specification must not result in a change of the public API of libIML. Changing this API may require a manual adaptation of a hundred tools which is simply too expensive. Therefore, changes have to be either API compatible or the serialization format itself has to be compatible to that one resulting from a changed specification. The latter case causes more than one specification of the intermediate representation in the long run. This will itself cause some maintenance effort that has to be anticipated.

Bearing in mind the never ending change of any technology, we should not only consider a migration strategy from Ada to Java, but one that would work for any target language. It would be naive to expect Java to live forever.

3 Realization

At first glance, one would expect that an XML-based solution exists. In fact, EMF[9] comes close to our expectations on a conceptual level. But, the lack of a code generator for Ada is already enough to prevent a thorough evaluation. The SKilL serialization system[1], on the other hand, seems to be a sufficient foundation for our intermediate representation. SKilL offers a sufficiently rich and easy-to-learn specification language. Also, there are code generators for the programming languages Ada, C++, Java and Scala readily available. In order to use this system, the existing specification has to be transformed into an equivalent SKilL specification. After this, an Ada API can be generated that can be combined with the existing libIML to create a format converter. Luckily, this task is rather easy because both APIs offer reflection that can be used to map objects between both representations almost automatically. Some data structures in IML bypass the specification and, therefore, require some hundred lines of glue code.

Now, we can start to examine important properties that this approach will have when the process of migration will have finished. Amongst them are file size, accessibility of logical data over the generated API and coding speeds. But most important is that existing test data can be converted to the new format. This does not only keep those data sets accessible. It is also the source of a valuable set of test cases for the migration process itself. For these tests, a sufficient amount of tools is required, whose output is deterministic. These tools can be run against all data sets and their output can be compared to their old variants automatically. In Bauhaus, a sufficient amount of such tools exists. Furthermore, we plan to keep the Ada implementation of libIML completely API compatible during migration. Hence, tools don't have to be changed in most[2] cases.

One can be pretty confident in the success of the overall migration process if the test set is sufficiently large which is the case for Bauhaus. Furthermore, it is completely irrelevant whether tools produce correct output. The only thing we are interested in is that they produce the same output as an indication of equivalent behaviour. This observation is very important in the context of a 20 year old toolchain because looking at individual tools' outputs reveals immediately that some tools had already required maintenance some time ago.

At this point, one could simply revert the direction of the format conversion tool. This would create a way back into the old IML world. But this leads to serious problems. Not only are further conversion steps between the two representations time-consuming. Also, a maintenance nightmare would be created because two completely different technologies were to be used in a central place of the toolchain that had to be maintained in parallel. Furthermore, converting from the old IML to a change-tolerant format will not make the old IML any more change-tolerant. In consequence, multiple versions of the old IML would need to be created that reflect the various possible contents of a data set which had to be changed with every extension to the intermediate representation. Therefore, the old IML is to be replaced completely.

[2] Unfortunately, some tools bypass libIML.

In parallel to the creation of a converter, a new generator for libIML has to be developed. This new generator should map its input specification to SKilL automatically. In consequence, the original specification can be kept although it can now be modified. This is, because the new underlying file format will automatically map the content to a new specification if the inheritance hierarchy is kept compatible and fields do not change type. Furthermore, the new generator has to ensure that the generated libIML API stays compatible to the existing API as intended even in the face of changed specifications. As soon as this generator can create a libIML that tools can be compiled against, fully automated testing can start. Our original goal is accomplished when all tests succeed.

4 Evaluation

A short demonstration of the effectiveness of this approach can be found in [2]. In order to test the applicability of the development process with the new API, a master thesis with the goal of implementing three pointer analyses has been performed (see [6] for details). Two out of these three analyses have already been implemented in similar theses (see [3] [4]). The third analysis has also been implemented in Bauhaus. Of course, the student had no access to the source code of those existing implementations. Furthermore, open source implementations of these analyses can be found on the internet which are not based on IML. But, in the course of the thesis it turned out that understanding the analyses in great detail and finding a way through the vast IML specification and its intricacies were the most time consuming tasks. The most important point is that he was in fact successful in implementing these analyses against the generated API. Furthermore, an interesting observation is that it is in fact possible to change the specification to fit a tool's needs without breaking compatibility to other tools. Therefore, we conclude that students can develop new prototypes with sufficient efficiency. Hence, the first of our initial problems can indeed be solved using this migration strategy.

Now, the question remains whether the costs of migrating the original IML to SKilL are acceptable. Przytarski [7] demonstrates in his master's thesis that this is in fact the case. His strategy splits into three key aspects. The first is to reengineer syntax and semantics of the specification language. Although some specification exists, the details required to create an automated mapping to a different specification language were no longer available. For instance, there is a mechanism to instruct the IML code generator to create a set type for a type declaration that can be used later on. This mechanism may appear rather strange to a Java programmer or even to somebody experienced in code generation. But this sort of information has to be addressed somehow. Also, the semantics of IML and SKilL types are sometimes quite different even though constructs share a name. This is for instance the case with enums which had to be mapped to SKilL integers instead. Similar obstacles are to be expected when considering the migration of other toolchains to SKilL as it is rather unlikely that semantics of specification languages turn out to fit perfectly in every detail. Furthermore, Przytarski describes and implements a code generator that uses the SKilL API for the implementation of the preexisting Ada libIML. Afterwards, he demonstrates that at least three different tools behave as before when compiled against his newly created libIML. Also, he shows that the overall performance of tool implementations stays at about the same level.

5 Conclusion

We have given an example demonstrating that a large toolchain can be migrated with SKilL to become extensible and language-independent. We consider the migration process to be rather inexpensive as it could be performed by just two master theses. Therefore, the approach can be applied easily by other academic researchers as long as their toolchain is communicating over a central intermediate representation that is generated out of a central specification.

References

[1] Timm Felden. The SKilL Language. Technischer Bericht, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2013.

[2] Timm Felden and Martin Wittiger. Migrating Bauhaus from IML to SKilL. In: *Softwaretechnik-Trends*, volume 36:2, 2016.

[3] Simon Frohn. *Implementierung einer unifizierenden Zeigeranalyse mit gerichteten Zuweisungen*. Students thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2004.

[4] Simon Frohn. *Konzeption und Implementierung einer Zeigeranalyse für C und C++*. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2006.

[5] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[6] Matthias Harrer. *Prototypenentwicklung mit Bauhaus und SKilL*. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2016.

[7] Dennis Przytarski. *SKilLed Bauhaus*. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2016.

[8] 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.

[9] David Steinberg, Frank Budinsky, Marcelo Paternostro and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009. ISBN 0321331885.

[10] Misha Strittmatter and Robert Heinrich. Challenges in the Evolution of Metamodels. In: *3rd Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems*, *Softwaretechnik-Trends*, volume 36(1), (pages 12–15), 2016.