Re-Implementing Apache Thrift using Model-Driven Engineering Technologies: An Experience Report

Sina Madani and Dimitrios S. Kolovos

Department of Computer Science, University of York, UK
{sm1748, dimitris.kolovos}@york.ac.uk

Abstract. In this paper we investigate how contemporary model-driven engineering technologies such as Xtext, EMF and Epsilon compare against mainstream techniques and tools (C++, flex and Bison) for the development of a complex textual modelling language and family of supporting code generators (Apache Thrift). Our preliminary results indicate that the MDE-based implementation delivers significant benefits in term of conciseness, coupling and cohesion.

1 Introduction

A number of frameworks and protocols have been developed over the years to facilitate service-oriented architectures, such as CORBA, Protocol Buffers, Avro and Thrift [1]. These technologies enable a cross-platform client-server model to be implemented with minimal duplication of development effort. This is made possible by the use of an Interface Definition Language (IDL); - a domain-specific language for defining the data and services provided by a server. These definitions are then used to automatically generate code to enable communication across languages and platforms. Despite most of these technologies being open-source, they are not straightforward to understand or modify; potentially due to the tools and processes used to develop them (e.g. code generation through string concatenation). We set out to investigate whether using contemporary model-driven engineering facilities can provide an improvement over traditional development methods and tools in terms of conciseness, cohesion and coupling. To achieve this we have re-implemented[1] a substantial part of Thrift using Xtext, EMF and Epsilon [2, 3], and in this paper we report on our findings.

2 Background

Apache Thrift is a software framework for defining services which work across languages and platforms. Thrift provides the necessary tooling to support a service-oriented architecture across several different languages using an Interface

[1] https://github.com/SMadani/ThriftMDE/

Definition Language (IDL). Thrift IDL files define the services, data types and functions provided by the services in a single domain-specific language. Thrift then parses these files and generates the appropriate skeleton code for both client and server in the chosen language(s).

The current implementation of Thrift's compiler is written entirely in C++ using conventional tools and a procedural approach. Since Thrift has its own syntax and type system, it requires a parser to process IDL files. To automate this process, the current implementation uses two well-established tools in order to generate the code for parsing Thrift files: flex and Bison. Flex (Fast Lexical Analyser) allows the developer to define a set of tokens using regular expressions, and when a rule is matched, a specified block of code is executed. These tokens make up the primitive building blocks of the language's grammar; which is then fed into a parser. Bison is a parser generator which allows the developer to define parser rules from a set of tokens, and specify a block of code to execute when a given rule is matched. Although the grammar specification of Thrift is defined between a flex and Bison file, the abstract syntax tree is left to the developer to configure; as is the integration between flex and Bison. These tools simply execute code when a certain set of symbols are encountered - they do not create any data types or objects. This is left to the developer to manually configure, and so the compiler has a "parse" directory solely dedicated to the declaration of classes to be used during the parsing process. This directory contains 20 files; 18 of which are class representations of Thrift types. For example, there is a 13KB file called "t_program.h" which represents an entire Thrift program. This class contains some utility methods as well as a list of all the program elements, such as services, structs, enums etc. There are other files for representing lower-level constructs too, such as functions and even fields. Again, it is worth noting that these classes must be created and populated by hand-written code in the Bison file.

The process of parsing Thrift's IDL in the current implementation is quite verbose and distributed across many files. The hand-written *flex* file is around 400 lines, and the parser is approximately 1300 lines of manually written code. Then there are the class representations of program elements; which are relatively short but numerous. One should also bear in mind that this does not include semantic validation; which is mostly performed in the main compiler program. The main compiler has four tasks: parse the command-line arguments, feed the Thrift file into the parser to obtain an object of type "t_program", validate the program and then pass it into the appropriate code generator depending on the language specified. This list of tasks occupies over 1200 lines of code; although it also includes help functionality, typical for command-line programs.

With regards to code generation, Thrift has a single C++ code file for each language, and a main file to perform any initialisation and call the top-level generation functions which are present in all languages. For example, the generator will call the "generate_service" function for every "t_service" object (i.e. a "Service" definition) in the IDL specification. All generators in all languages have these top-level functions defined. The size of the generators vary greatly depending on the language. In the Java generator, there are approximately 100 functions and the file is 5129 lines long (187KB). The Ruby generator however is much smaller at 1231 lines long (40KB). The code itself is written to a file output stream directly with string concatenation.

Thrift's compiler appears to be an ideal use case for an MDE approach. After all, code generation has been described as "the heart and soul of model-driven engineering" [4], and there are a number of tools available specifically designed for the kinds of tasks involved in creating a domain-specific language. Intuitively, it makes sense to treat an abstract syntax of a language as a model; which can then be validated and transformed with dedicated tools. The purpose of this work is twofold. Firstly, it aims to establish whether a model-driven approach is indeed superior to the more conventional approach as used by Thrift; and to what extent. Secondly, it serves as an evaluation of the maturity of MDE tools by applying them in practice to implement a sufficiently complex compiler that is currently in use.

3 Re-Implementing Thrift using Xtext and Epsilon

The Eclipse Modelling Framework (EMF) offers a platform for interoperability between MDE tools. This is primarily achieved through Ecore, which is considered to be the de-facto implementation of the Object Management Group's Essential MOF (Meta-Object Facility). Ecore offers a unified meta-modelling solution on the Eclipse platform, which can be used by various applications. Of particular significance to this project are two such tools: Xtext and Epsilon.

Xtext is a framework for creating and supporting textual domain-specific languages. Unlike conventional parser generators such as Bison, Xtext offers a complete all-in-one solution for developing a DSL. This includes a tokenizer, parser, editor and even a metamodel for the language. Xtext has its own grammar definition language which allows the user to define their language using an Extended Backus-Naur Form notation. This means that unlike older tools such as Bison, the grammar is self-documenting and more readable because it does not contain any imperative code. For instance, Listing 1.1 is a definition of Thrift's "Service" concept in Xtext:

```
//Services are interfaces.
Service:
  (comment = ML_COMMENT)? 'service' identifier = IDENTIFIER ('extends' supers =
      IDENTIFIER)? '{' (functions += Function separators += LIST_SEPARATOR)* '}';
```

Listing 1.1. Xtext definition of Thrift's Service concept

Perhaps one of Xtext's most useful features is that the grammar is backed by an Ecore metamodel, which can be automatically generated from the grammar itself. That is, Xtext automatically maps grammar rules into appropriate Ecore types. When the Xtext parser is run against a Thrift IDL file, the abstract syntax tree (as defined by the generated Ecore metamodel) is populated from matching rules. Therefore, unlike the C++ implementation, the process of obtaining an AST is largely automated.

Furthermore, Xtext and EMF have APIs which allow for a stand-alone Java application to be set up. This allows us to programmatically invoke the parser against a given Thrift IDL file and obtain the AST in the form of an Ecore model in just a few lines of Java code. Once obtained, we can then pass this into Epsilon; which we turn to next.

Epsilon is a family of model management tools and languages aimed at simplifying common MDE tasks such as model validation and model transformation. Epsilon is compatible with a variety of model sources, including EMF/Ecore. At the core of Epsilon is the Epsilon Object Language (EOL), an imperative model-oriented language that combines the procedural style of JavaScript with the powerful model querying capabilities of OCL. Epsilon also has two task-specific languages of interest: Epsilon Validation Language (EVL) and Epsilon Generation Language (EGL). The former is used to perform semantic validation, and the latter is used for code generation.

As with Xtext, Epsilon has a Java APIs, so we can pass our parsed EMF model into both EVL and EGX modules; as well as including additional parameters (for example, the current date or the specified language). Naturally, the first step is to perform semantic validation. Although Thrift is not a Turing-complete programming language, it is still sufficiently complex to warrant validation logic beyond what is feasible at the parsing stage. It supports C-like features such as defining constant values (including complex types such as maps and lists), structs, unions etc. as well as inheritance between services and exceptions – amongst other things. There is also a need to ensure that names do not clash with any of the supported languages' keywords, and to check for unique identifiers within a particular scope. For example, two functions may have the same name in different services but not if they're part of the same service. EVL provides the necessary constructs to perform all of these checks and report any errors in a concise manner. For example, an EVL constraint that checks that all of the fields in a given scope are uniquely named is illustrated in Listing 1.2.

```
context Field {
  guard: self.eContainer.isDefined()
  constraint uniqueIdentifiers {
    guard: self.eContainer.hasProperty("fields")
    check: self.eContainer.fields.one
      (field | field.identifier == self.identifier)
    message: self.eContainer.identifier+" has multiple fields with the name '"+self
        .identifier+"'."
  }
}
```

Listing 1.2. EVL constraint that checks name uniqueness

Moving on to code generation, we use two languages: EGX and EGL. EGL is a template-based model-to-text language that compiles down to EOL. As such, we can make full use of EOL's features, such as operations and extended properties. In order to improve readability, re-use and cohesion, we can move commonly used functionality to separate EOL and/or EGL files (since operations may also contain static sections).

Unlike many interface definition languages, Thrift supports container types such as maps and sets (which may have arbitrary levels of nesting), as well as "structs" and "typedefs". Even with a neatly structured grammar, the sheer number of different types that Thrift supports requires an extensive number of operations to process. For each language, there is also a separate EGL file which contains further language-specific operations which are used by other templates. These may include operations which are sufficiently complex to warrant separation from the main template to improve readability, or operations in a more generic context (such as "Field"; which is used by most top-level elements) that serve as utility functions - for example, converting from a Thrift type to a language-specific type.

EGX is a rule-based co-ordination language for co-ordinating the execution of EGL templates. For instance, we can create a rule which will invoke the "enum" template for every "Enum" type in our model. We can pass the relevant parameters to the template, which may have been declared in the "pre" block or from Java. The invoked template's output will be written to the specified target file as illustrated in Listing 1.3.

```
rule Enums transform enumeration : Enum {
  parameters {
    var params : new Map;
    params.put("enumeration", enumeration);
    params.put("package", pkgId);
    params.put("date", date);
    return params;
  }
  template: "thrift-java-enum.egl"
  target: outDir+enumeration.identifier+".java"
}
```

Listing 1.3. EGX rule for generating Enumerations

4 Evaluation

We have re-implemented Thrift's compiler with support for two output languages: Java and Ruby (though without additional generator option flags). We will compare our implementation with the existing one using lines of (handwritten) code as a metric. Clearly, this is by no means an ideal qualitative measure for comparing two different approaches due to potentially differing coding styles. However, if the two implementations vary significantly in terms of lines of code, then we may be able to draw some conclusions with regards to the general effort required to achieve the same functionality.

It should be noted that we tried to replicate the existing implementation's functionality as closely as possible. With regards to the generator(s), this involved closely inspecting the source code as well as the output for a number of different Thrift IDL files - specifically from Thrift's repository of complex test files designed for such purposes. We used Eclipse's text comparison editor to highlight any differences in the output of the two implementations, and iteratively worked to ensure that no (major) differences were found between their respective outputs. The results are presented in Table 1.

Table 1: LOCs in the components of the two implementations

Implementation	**C++**	**Model-Driven**
Language definition (parsing & validation)	3419 (105 KB)	447 (14 KB)
Language-neutral code	712 (22KB)	1036 (26 KB)
Java generator	5129 (187 KB)	2224 (73 KB)
Ruby generator	1231 (40 KB)	422 (14 KB)
Total	**>10491 (395 KB)**	**4149 (128 KB)**

Firstly, we begin by comparing the parsing process. We shall separate this into two parts: the grammar and semantic validation. With regards to the grammar, this includes all hand-written code required to obtain the IDL abstract syntax tree for a given input file. This also includes any code required to read in the input file specified in the command-line arguments. For our implementation, this is around 7 lines of Java code to obtain an "EmfModel" from a specified input file. The grammar definition in Xtext is 146 lines long. Meanwhile, the C++ implementation is approximately 400 lines for the lexer, and 1300 lines for the parser. On top of that, there are the additional parse types; which add another 1560 lines (approximately). There is also the header file for the main program at around 80 lines, as well as the "parse" function in the main program; which is 70 lines long. This totals well over 3000 lines of manually written code just to obtain the AST, compared to the MDE implementation's 150 or so lines.

Moving on to validation, the existing implementation appears to be quite concise, with approximately 250 lines of code. By contrast, our EVL constraints file combined with some of the helper functions it uses is approximately 360 lines. However, it is worth noting that the two implementations do perform certain checks at different stages. For example, there is a set of illegal identifiers which are banned because they clash with language keywords. In the existing implementation, they are declared during the parsing process, whereas we check for these during validation. Validation is also performed at ï£¡runtimeï£¡ (i.e. during code generation) in the existing implementation by throwing exceptions. Some of the validation logic in the current implementation is also built in to some of the parse types (classes), so it is difficult to separate the validation from parsing to some degree. Nevertheless, there seems to be little effort saved in validation by using EVL over a general-purpose language.

Finally, we turn to code generation. The current implementation has a separate language-independent generator file which is over 150 lines, and is roughly similar in function to our EGX coordination rules. Our Ruby EGX is 65 lines of code. Meanwhile, our Ruby generator is 422 lines of code when adding up all of the templates. However, this does not include the generic EOL file (1036 lines) which contains many commonly used operations that are language-independent, but still used by the Ruby generator. Our generators also do not support any additional option flags. Based on these results, it appears that although a template-based approach to code generation has the potential to be more concise (by up to threefold in the case of Ruby) and, arguably, more readable, there is insufficient evidence to suggest that the benefits are overwhelming compared to traditional approaches - particularly for the dynamic code generation sections. This is perhaps because the Thrift generator contains a lot of complex logic which can be comfortably expressed using a general-purpose language such as C++, whilst there are relatively fewer "static" sections; where a template-based language would be better suited especially for complex languages like Java; where the savings in terms of code are relatively smaller (5129 vs 2224). That is, a template-based model-to-text transformation can reduce the accidental complexity, but has marginal benefits in expressing inherent complexity more concisely.

We appreciate that fewer lines of code is not necessarily an improvement as this can be achieved by simply using an obscure coding style. Of greater interest are metrics such as readability and modifiability; which are arguably related to cohesion and coupling. Although these are subjective to some extent, we can make some comparisons based on the structure of the two implementations. We have seen that the existing compiler has many files used for defining the IDL and parsing it; all of which vary greatly in length but contribute piecewise to the process. For instance, it would be difficult to argue that the existing implementation's grammar is easier to deduce from flex and Bison source compared to our singular Xtext file (which arguably has self-documenting syntax). If one wanted to make a change to the language, it would simply be a case of modifying the Xtext grammar, whereas the equivalent change in the existing implementation may be much more complicated due to the low cohesion and high coupling between various files.

Furthermore, we argue that a template-based approach is more cohesive and traceable than using one huge file because it is easier to find which part of the generator contributes to a specific part of the output. For instance, by having an EGX file, we can see how many files will be produced and what they will be called. By splitting up the generation into multiple templates, each template file reads similarly to the expected output file; albeit with dynamic sections to express some logic. With regards to validation, we argue that using a domain-specific language such as EVL makes for a much more cohesive expression of language semantics - not only because of the syntax but also because it's a single file; as opposed to the relatively sparse approach of the existing implementation. Overall, we make the case that a model-driven approach is far "cleaner" in its workflow compared to a C++-based implementation.

5 Conclusions

We set out to discover whether the contemporary MDE tools and techniques offer a significant advantage over conventional approaches in creating a compiler for an intermediate language. Our findings suggest that frameworks such as Xtext offer a much better alternative to defining and parsing a language than traditional tools such as flex and Bison. This is mainly because Xtext enables developers to define a grammar with relative ease and conciseness without extensive knowledge of the concepts involved in the lexing and parsing process, whilst also providing an abstract syntax tree without requiring the developer to explicitly specify how to do this. However, with respect to code generation, it appears that although a template-based model-to-text transformation provides a significant reduction in code required to achieve the same output, the benefits are less pronounced for complex dynamic sections.

We have focused on an objective measure - in this case, the number of lines of code. However, this metric is arguably less useful to a prospective developer than more subjective aspects, such as readability and modifiability. Given the specialised nature of the tools used in our model-driven approach, we argue that an MDE-based implementation is more cohesive because each stage of the compilation process has its own dedicated tools and semantics. For instance, the grammar definition and overall parsing process is far more concise (and arguably more readable) because Xtext allows us to focus solely on language specification rather than dealing with relatively low-level lexing and parsing constructs. With respect to validation, although there aren't any gains to be made in terms of lines of code, the specialised nature of a language like EVL makes for a more idiomatic approach to specifying constraints. Finally, code generation is also more specialised, with a template-based approach allowing for improved readability; since each template describes a single file, and it is easy to distinguish between static and dynamic sections. However, for logic-intensive generators such as Thrift, there is not a significant enough difference in terms of lines of code to recommend an MDE approach solely on the basis of conciseness. However, we recommend the MDE approach overall for its superior cohesion.

Further research on this topic should focus on evaluating the two approaches more formally - perhaps through developer surveys or elaborate experiments - across a wider range of more important criteria, such as ease of development, readability, testability and modifiability. In doing so, further insights can be gained on both the adequacy of MDE tools and the logistical benefits of a model-driven architecture.

Acknowledgements This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125).

References

1. Apache Foundation. Apache Thrift, 2016. `https://thrift.apache.org`.
2. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.
3. Eclipse Foundation. Epsilon, 2016. `https://eclipse.org/epsilon`.
4. Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September/October 2003.