

Final thesis

**Ada code generation support for  
Google Protocol Buffers**

by

**Niklas Ekendahl**

LITH-IDA-EX-PUBYEAR/THESISNUMBER

DATEOFPUBLICATION



Final thesis

# Ada code generation support for Google Protocol Buffers

by

**Niklas Ekendahl**

LITH-IDA-EX-PUBYEAR/THESISNUMBER

DATEOFPUBLICATION

Supervisors: Ulf Kargén  
Joakim Strandberg (Saab SDS)

Examiner: Nahid Shahmehri



# Abstract

We now live in an information society where increasingly large volumes of data are exchanged between networked nodes in distributed systems. Recent years have seen a multitude of different serialization frameworks released to efficiently handle all this information while minimizing developer effort. One such format is Google Protocol Buffers, which has gained additional code generation support for a wide variety of programming languages from third-party developers.

Ada is a widely used programming language in safety-critical systems today. However, it lacks support for Protocol Buffers. This limits the use of Protocol Buffers at companies like Saab, where Ada is the language of choice for many systems. To amend this situation Ada code generation support for Protocol Buffers has been developed. The developed solution supports a majority of Protocol Buffers' language constructs, extensions being a notable exception.

To evaluate the developed solution, an artificial benchmark was constructed and a comparison was made with GNATColl.JSON. Although the benchmark was artificial, data used by the benchmark followed the same format as an existing radar system. The benchmark showed that if serialization performance is a limiting factor for the radar system, it could potentially receive a significant speed boost from a substitution of serialization framework. Results from the benchmark reveal that Protocol Buffers is about 6 to 8 times faster in a combined serialization/deserialization performance comparison. In addition, the change of serialization format has the added benefit of reducing size of serialized objects by approximately 45%.



# Acknowledgements

First of all I would like to thank my examiner professor Nahid Shahmehri for giving me the opportunity to work on this thesis. I would also like to thank my supervisors Ulf Kargén and Joakim Strandberg for their invaluable feedback and encouragement, without them this thesis could never have been completed.





# Contents

<b>List of tables</b>	<b>xi</b>
<b>List of figures</b>	<b>xiii</b>
<b>Listings</b>	<b>xvi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Purpose and goals . . . . .	2
1.3 Scope . . . . .	2
1.4 Audience . . . . .	3
1.5 Limitations . . . . .	3
1.6 Methodology . . . . .	4
1.6.1 Ada code generation for Protocol Buffers . . . . .	4
1.6.2 Performance comparison . . . . .	5
1.7 Typographical conventions . . . . .	5
1.8 Thesis overview . . . . .	5
<b>2 Data serialization formats</b>	<b>7</b>
2.1 History . . . . .	8
2.2 Google Protocol Buffers . . . . .	9
2.2.1 Overview . . . . .	9
2.2.2 Techniques for solving common problems . . . . .	10
2.2.3 Protocol Buffers binary encoding format . . . . .	11
2.2.4 Example . . . . .	14
2.3 JavaScript Object Notation (JSON) . . . . .	15
2.3.1 GNATColl.JSON . . . . .	16
<b>3 Ada code generation for Google Protocol Buffers</b>	<b>19</b>
3.1 Ada . . . . .	19
3.1.1 Packages and types . . . . .	19
3.1.2 Predefined types . . . . .	20
3.1.3 Memory management . . . . .	21
3.2 Software requirements . . . . .	21

3.3	Plug-in or standalone application? . . . . .	21
3.4	Implementing a plug-in . . . . .	22
3.5	Deciding how to implement the plug-in . . . . .	22
3.6	Challenges faced during development . . . . .	23
3.7	Structure of the developed software solution . . . . .	24
3.7.1	Supporting libraries . . . . .	24
3.7.2	Code generator design . . . . .	26
3.7.3	Code generation for Ada . . . . .	27
3.8	Testing . . . . .	29
3.8.1	Unit testing framework . . . . .	29
3.9	API . . . . .	29
3.9.1	Messages . . . . .	30
3.9.2	Fields . . . . .	31
3.10	Installation and use . . . . .	37
3.10.1	From .proto file to executable application . . . . .	37
<b>4</b>	<b>Evaluation</b>	<b>41</b>
4.1	Requirements . . . . .	41
4.2	Performance comparison . . . . .	41
4.2.1	Benchmark environment . . . . .	42
4.2.2	Measurement techniques and results . . . . .	42
4.2.3	Analysis of results . . . . .	43
4.3	Future investigations and improvements . . . . .	44
4.3.1	Performance . . . . .	44
4.3.2	Features . . . . .	45
<b>5</b>	<b>Conclusions</b>	<b>49</b>
	<b>Appendices</b>	<b>51</b>
<b>A</b>	<b>System requirements</b>	<b>53</b>
A.1	Definitions and conventions . . . . .	53
A.2	Non-functional requirements . . . . .	53
A.3	Functional requirements . . . . .	54
<b>B</b>	<b>Address book example</b>	<b>63</b>
B.1	Adding contacts to the addressbook . . . . .	63
B.2	Listing contacts from the address book . . . . .	65
<b>C</b>	<b>Unit tests</b>	<b>67</b>
C.1	Coded_Output_Stream test suite . . . . .	67
C.2	Coded_Input_Stream test suite . . . . .	68
C.3	Message test suite . . . . .	69

<b>D</b>	<b>Performance comparison</b>	<b>71</b>
D.1	Radar system data format . . . . .	71
D.2	GNATColl.JSON . . . . .	73
D.3	Protocol Buffers . . . . .	75



# List of tables

2.1	Wire types Protocol Buffers . . . . .	12
2.2	Type . . . . .	12
2.3	Examples of sint32 encoded numbers . . . . .	13
3.1	Scalar value types and their representation . . . . .	31
A.1	Non-functional requirements . . . . .	54
A.2	Functional requirements . . . . .	54
C.1	Coded_Output_Stream test cases . . . . .	67
C.2	Coded_Input_Stream test cases . . . . .	68
C.3	Message test cases . . . . .	69



# List of figures

3.1	Compilation of example program . . . . .	25
3.2	Class diagram . . . . .	28
4.1	Performance comparison. . . . .	47





# Listings

1.1	Code listing example . . . . .	5
2.1	<b>person.proto</b> . . . . .	14
2.2	Text format serialization . . . . .	15
2.3	Binary serialization . . . . .	15
2.4	GNATColl.JSON example . . . . .	16
2.5	Output from GNATColl.JSON example . . . . .	17
3.1	Circular dependency example . . . . .	24
3.2	Serializing procedures . . . . .	30
3.3	Parsing procedures . . . . .	30
3.4	Merging procedures . . . . .	30
3.5	Miscellaneous procedures and functions . . . . .	30
3.6	Singular numeric field definition . . . . .	31
3.7	Generated functions for singular numeric fields . . . . .	32
3.8	Singular string/bytes field definition . . . . .	32
3.9	Generated functions for singular string/bytes fields . . . . .	32
3.10	Enum definition . . . . .	33
3.11	Singular enum field definition . . . . .	33
3.12	Generated functions for singular enum fields . . . . .	33
3.13	Message definition . . . . .	34
3.14	Singular embedded message field definition . . . . .	34
3.15	Generated functions for singular embedded message fields . . . . .	34
3.16	Repeated numeric field definition . . . . .	34
3.17	Generated functions for repeated numeric fields . . . . .	35
3.18	Repeated string/bytes field definition . . . . .	35
3.19	Generated functions for repeated string/bytes fields . . . . .	35
3.20	Repeated enum field definition . . . . .	36
3.21	Generated functions for repeated enum fields . . . . .	36
3.22	Repeated embedded message field definition . . . . .	36
3.23	Generated functions for repeated embedded message fields . . . . .	37
3.24	<b>addressbook.proto</b> . . . . .	38
3.25	Address book demonstration . . . . .	39
B.1	<b>add_person.adb</b> . . . . .	63

B.2	<code>list_people.adb</code> . . . . .	65
D.1	Description of radar system data . . . . .	71
D.2	Proto definition file for the radar system ( <code>radar.proto</code> ) . . .	72
D.3	Benchmark application source code for GNATColl.JSON . . .	73
D.4	Benchmark application source code for Protocol Buffers . . .	75

# Chapter 1

## Introduction

This final thesis report is written as part of a master's degree in computer engineering at Linköping University. The thesis work has been performed at *Saab Security and Defence Solutions (Saab SDS)* in Järfälla.

Saab is an international company that provides services and products for both military defense and civil security. Saab SDS is a business area at Saab, with a focus on defense reconnaissance systems, airborne early warning systems, training and simulation, air traffic management, maritime security, security and monitoring systems, and solutions for safe, robust communications. (Saab Group, n.d.)

### 1.1 Background

Saab SDS has a product portfolio, which contains several distributed systems, one such distributed system is a radar system, which tracks and displays targets using a web interface. Somewhat simplified, the system can be said to consist of two separate components, the radar that gathers information about targets and a web server that displays information about targets. To communicate target information to the web server the radar uses *JavaScript Object Notation (JSON)*, a standard text-based data serialization format. The radar system handles large data quantities and, because of this, serialization performance is an important concern. Saab SDS therefore wants to investigate if the system's performance can be increased by substituting the text-based serialization format with a binary serialization format. Saab SDS has previous experience using the binary serialization format *Protocol Buffers* in other projects, and because of this sees it as a good candidate format for use in serializing target data in the radar system. (J. Strandberg, personal communication, May, 2013)

Protocol Buffers are an open-source implementation of a protocol for interchanging information that was internally developed at Google. The following is a description from Google's Protocol Buffers website.

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the ‘old’ format. (Google, 2012b)

Official C++, Java and Python compilers for Protocol Buffers are available from Google, and in addition to that, compilers for other programming languages have been developed by third-parties.

However, the web server used by the radar system is written in Ada. Unfortunately, Protocol Buffers lack support, both official and third party, for Ada. At Saab SDS, components written in Ada are common, and for that reason, development of a compiler for Protocol Buffers, with code generation support for Ada, would be of great benefit to Saab SDS.

## 1.2 Purpose and goals

The primary goal of the thesis is to develop a compiler for Protocol Buffers, with code generation support for Ada, and to evaluate the developed solution. Evaluation will be done using data representative of that from an existing radar system. At present the radar system uses JSON for serialization purposes; serialization with Protocol Buffers will therefore be compared to serialization with JSON. A more general comparison of JSON and Protocol Buffers will also be conducted to serve as an aid in future decisions regarding data serialization formats.

To establish if Protocol Buffers is a possible candidate for replacing JSON in the radar system, this thesis aims to answer the following questions for the radar system:

- Can Protocol Buffers be used to represent the same information as JSON?
- What performance advantage, if any, does the developed Protocol Buffers solution provide over JSON? Is performance at least as good as with JSON?

## 1.3 Scope

The intent of the work, done as part of this thesis, is to produce a software solution that will provide Ada code generation for Protocol Buffers. The time constraints, which are inherent to a master thesis, dictate the amount of the work that can be done as part of a thesis. Because of this, it is not

feasible to offer the same level of functionality as that provided by the official implementations. Support for dynamic messages<sup>1</sup>, which makes it possible to manipulate unknown protocol types, has for example not been considered, or included into the software solution produced.

The performance comparison includes only GNAT Component Collection's JSON implementation and the developed Protocol Buffers solution. A more thorough investigation into the performance of data serialization formats would entail comparing Protocol Buffers to other data interchange formats as well.

## 1.4 Audience

Most of the material in this thesis is presented in form that should be approachable to anyone who has a basic familiarity with software development. The reader is assumed to have an elementary understanding of object-oriented programming, but any terminology that is specific to either Ada or C++ should be explained in the text or in the glossary.

The material in this thesis touches upon, albeit very briefly in some cases, the following subjects:

- Ada software development
- Benchmarking
- Open-source software development
- Serialization
- Google Protocol Buffers
- JSON

It is the author's hope that a reader, who is interested in any of these subjects, will find something of interest when reading this thesis.

## 1.5 Limitations

In the thesis, the performance of GNAT Component Collection's implementation of JSON is taken as indicative of JSON performance, when using *Ada*. Other implementations exist, such as the one provided in the *serialize* package that is part of *Ada Util*<sup>2</sup>, but it is the author's opinion that performance will be comparable between different implementations of JSON.

Protocol Buffers is a binary format and the logical course of action would therefore be to compare it to not only a text-based format, but also to

---

<sup>1</sup>[https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.dynamic\\_message](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.dynamic_message)

<sup>2</sup><http://code.google.com/p/ada-util>

other binary formats. Ada software support for data serialization formats is, regrettably, limited. Because of this, JSON was chosen as a comparison format, even though it is text-based.

All written Ada code has been compiled using *GNAT* which at the time of this writing is the only compiler available with support for Ada 2012. Unfortunately, this means that it has not been possible to test any of the code written (or generated) for Ada using another compiler.

The developed solution has not been written or verified to work on big-endian architectures, but it should not entail too much work making the changes necessary to support Ada code generation for big-endian architectures.

## 1.6 Methodology

Work on the thesis has been divided into two phases, development of Protocol Buffers code generation support for Ada and conducting a performance comparison between the developed solution and JSON. An outline of the approach taken in each of these phases is described below.

### 1.6.1 Ada code generation for Protocol Buffers

Implementing Ada code generation for Protocol Buffers required a thorough understanding of Protocol Buffers, both the underlying format and the supporting software. Rather than starting ‘from-scratch’ and developing a solution based only on the underlying format, it was deemed that studying already existing solutions would provide beneficial clues to the implementation. The structure of both official and third-party solutions was studied, to gain an idea on how best to implement Ada software support for Protocol Buffers. Furthermore, knowledge of Ada was, of course, essential to the development.

After having studied existing implementations of Protocol Buffers for other languages, a throwaway prototype was constructed. The prototype was constructed to help with elicitation of requirements and to study alternative ways of implementing Ada code generation for Protocol Buffers. The prototype also provided much needed Ada development experience.

Functional and non-functional requirements were then gathered. The requirements were elicited from the prototype, the Protocol Buffer language, and from official and third-party implementations of Protocol Buffers. Requirements were deliberately written to capture what the system should be able to do and not exactly how it should do it because of the difficulties associated with specifying an *Application Programming Interface (API)* in advance.

## 1.6.2 Performance comparison

To make the performance comparison as general as possible a decision was made to isolate the serialization part of the radar system. The performance comparison will thus only compare serialization performance with representative data and not the radar system as whole. A consequence of this is that test results are easier to reproduce, and furthermore, it has the added bonus of reducing the complexity of the benchmark setup.

## 1.7 Typographical conventions

The following is a list of typographical conventions used throughout the document.

- Terms defined in the glossary will be displayed in an italic font the first time they are used, e.g. *term*.
- Filenames and will be displayed in a fixed-width font commonly referred to as a monospaced font, e.g. `filename.suffix`.
- Commands intended to be entered on the command line will be displayed in a monospaced font with a greater-than symbol as a prefix, e.g. `> date`.
- Protocol buffer definition files will be displayed in a monospaced font with keywords show in purple, see listing 1.1.

Listing 1.1: Code listing example

```
message Person {  
  required string name = 1;  
}
```

## 1.8 Thesis overview

The thesis is divided into five chapters, the contents of which are briefly described here.

**Chapter 1** provides the reader with an introduction and a background to the thesis.

**Chapter 2** is divided into three parts:

- The first part gives a more in-depth description, than the one provided in the introduction, of data serialization formats in general.

- The second part is dedicated to describing Protocol Buffers. It provides a detailed description of Protocol Buffers binary encoding and the interface description language used to describe serialized data. A simple example illustrating the use of Protocol Buffers is also included at the end of the second part.
- The third part introduces JSON. It begins with an overview of JSON, which is then followed by a brief description of GNAT Component Collection's implementation of JSON.

**Chapter 3** contains a description of the developed Ada code generation software, implemented as part of this thesis. A brief overview of Ada is given in the beginning of the chapter for readers unfamiliar with the language. Elicited software requirements are then presented and motivations for major design decisions are explained. An outline of the developed software solution is given and the resulting Protocol Buffers API for Ada is described. The chapter ends with a section that explains installation and use of the developed Ada Protocol Buffers API.

**Chapter 4** compares the performance of GNAT Component Collection's implementation of JSON with the performance of Ada code generation for Protocol Buffers, developed as part of the thesis. The chapter starts with a description of how the benchmarks were conducted, followed by the actual benchmark results and ending with an analysis of the results.

**Chapter 5** presents conclusions and summarize the thesis.



## Chapter 2

# Data serialization formats

Serialization describes a process whereby objects or data structures stored in main memory are saved for later reconstruction. Serialization encodes data stored in a data structure in such way that it can later be stored on disk or transferred over a network medium. The opposite of serialization is deserialization, where objects or data structures are recreated from previously serialized data.

Many programming languages have standard library routines that provide support for serialization, for example Java, Python and PHP. What is common among these languages, and a majority of the programming languages with built-in support for serialization, is that they use a format that is programming language specific. Data serialized using one programming language can therefore not easily be read by software developed in another programming language and vice versa.

There are also data serialization formats that are intended to work independently of the programming language used and the deployment architecture. Architectural independence relates to the computer architecture a program executes on, when serializing and deserializing data. For instance, different computer architectures might use different representations for storing integers.

As has been previously mentioned, data can be serialized using either a text-based encoding or a binary encoding depending on serialization format, although some formats support both binary and text-based encodings. Formats that use a mix of binary and text-based encoding, where some types of values are encoded as text and others types are encoded in a binary encoding also exist. The obvious advantage to using a text-based format is that it is easier for humans to understand. Serialized objects can be inspected without difficulty and values determined by simply reading the serialized description of the object. A downside to a text-based encoding is that the encoding of data might be less space efficient than the equivalent binary encoding.

Another factor that differentiates different serialization formats from each other is their support for serializing an object in memory. Some serialization formats have tools that can generate code that allows the programmer to construct objects, which can be serialized directly, whereas others require that the programmer serialize objects manually. The word manually in this case refers to the process of serializing and deserializing every primitive data type an object consists of separately, a process that can be both cumbersome and error prone. Generating code, that support serialization of objects represented in memory, is highly dependent on the programming language being used. This means that for code generation to work for a programming language it needs to be implemented specifically for that programming language. Code generation support for serialization formats is for that reason often not available for programming languages with a smaller user base.

To serialize objects or data structures, some serialization formats require the use of *Interface Description Language (IDL)* files. An IDL file provides a description of the serialized information in a programming language independent way. IDL files are often used by tools to generate code that support direct serialization of objects.

## 2.1 History

A precursor to the many data serialization formats used today is *Comma-Separated Values (CSV)*. As the name suggest data or values are written as text, which are separated by commas. The format or variants of the format have been in use since the early 1970s (IBM, 1972). Although the format has existed for nearly 40 years, no formal documentation of it existed until recently (Shafranovich, 2005).

In 1984, the International Telegraph and Telephone Consultative Committee (CCITT), now known as the Telecommunication Standardization Sector (ITU-T), released a draft recommendation<sup>1</sup> for an information encoding to be used for communication in distributed systems (Pope, 1984). The standard was originally intended for use in e-mail systems, but the format is today used in a wide range of applications<sup>2</sup>.

*Abstract Syntax Notation One (ASN.1)*, as the format is called nowadays, uses an IDL to describe how data is to be represented. The IDL used by ASN.1 is called *Abstract Syntax Notation (ASN)* and was developed to provide a platform independent language that is able to describe data structures in any programming language. ASN does not specify how information is to be encoded, instead, that is left to the transfer syntax. This separation of concerns makes it possible to associate different encodings of data to a single abstract syntax definition by merely choosing or specifying a new transfer syntax. The abstract syntax definition and the transfer syntax specified are

---

<sup>1</sup>Encoding CCITT X.409 Presentation Transfer Syntax

<sup>2</sup><http://www.itu.int/ITU-T/asn1/uses/index.htm>

then used by an ASN.1 compiler to generate code, supporting serialization for a *target language*. (Dubuisson, 2000)

Another major milestone in the history of data serialization formats came with the introduction of *Extensible Markup Language (XML)*. Development of XML started in the late 1990s with the goal of creating a simple format that could be used for communication over the internet (Bray, Paoli, Sperberg-McQueen & Yergeau, 2013). XML, in its original design, is a text-based self-describing format (Bray, Paoli & Sperberg-McQueen, 1998). A self-describing format mixes data with the description of said data, which makes the use of *schema* files unnecessary. Use of XML today is widespread and it has been deployed in a wide variety of applications.<sup>3</sup>

XML and ASN.1 had at the time of their creation very different design goals in mind and they are therefore widely different. However, the formats have evolved and received various extensions since their initial releases and the descriptions here are therefore simplified to make the descriptions fit into the limited space available.

## 2.2 Google Protocol Buffers

Google Protocol Buffers is a data serialization format, which uses IDL files to specify and generate code to facilitate serialization of data.

### 2.2.1 Overview

The IDL files used by Protocol Buffers, from here on referred to as **.proto** files because of their filename suffix, employs a construct known as a message to describe data. A message can be seen as a record that contains a set of fields, which can be of either a scalar type<sup>4</sup> or a message type. Fields inside a message can be declared as either optional, repeated or required. A required field inside a message must be set before serialization, while no such requirement is imposed on optional fields. A repeated field can be seen as a vector containing an arbitrary number of optional fields. Unlike XML the serialized data contains no self-describing information and a **.proto** file is needed to interpret the data. All data that is serialized by Protocol Buffers is serialized using a binary encoding; a text format is also available, but it is only useful for debugging purposes. (Google, 2013b)

Having described the data-structure in a **.proto** file, the next step is to compile the **.proto** file using a protocol buffer compiler to generate accessors so that data can easily be read and written from and to raw bytes. The official compiler from Google supports code generation for C++, Java and Python. However, the feature set differs a bit between different languages. For instance, the code generation for Java and C++ recognize the option

---

<sup>3</sup><http://xml.coverpages.org/xmlApplications.html>

<sup>4</sup>The following scalar types are available: double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64, bool, string and bytes.

`optimize_for` that can be used to optimize the generated code for different use cases, such as speed or size. An added benefit with the Java compiler is that it generates code that supports the builder design pattern, which greatly simplifies the construction of messages. (Google, 2013b)

To make development with Protocol Buffers easier Google provides a plug-in for editing `.proto` files in Eclipse (3.7) that includes features such as syntax highlighting, content assist, outline view, automatic field number generation etc. Google also maintains a list<sup>5</sup> of third-party add-ons, included in the list among other things are compilers for other languages and RPC implementations, which support Protocol Buffers. Although support for several languages exists from third-parties Ada is currently not one of the supported languages. (Google, 2012f)

A major advantage with Protocol Buffers is that optional and repeated fields can be added and removed from a message specification without breaking any backwards compatibility. A `.proto` file can for instance be extended with an optional address field. Applications compiled with the previous message specification will simply ignore the new field. A benefit with this is that no special code is needed to inspect the data to handle messages of different versions, which simplifies the introduction of new message protocols. (Google, 2013b)

Protocol Buffers also support something called extensions. Extensions allow the reservation of tags inside a message for future use. The reserved tags can then later be used to define new fields inside a message. The definition of new fields can also be done in a separate `.proto` file that includes the original `.proto` file the message was defined in. To read and write an extended field special getters and setters are used, and when using Java a registry of extensions needs to be explicitly built to parse the extensions. (Google, 2013b)

The extensions mechanism has many uses. It could, for instance, be used to reserve fields for third-party use.

Serialization is an important part of systems that use *Remote Procedure Call (RPC)*, because of this Protocol Buffers provides built-in support for specifying so called services inside a `.proto` file. The official support for generating RPC services from `.proto` files is limited as it only generates an interface from the `.proto` file. Part of the reason for not providing a complete implementation is that it would be impossible to provide a solution that would work across all RPC implementations. The recommendation is therefore instead to use a third-party plug-in developed for a specific RPC implementation. (Google, 2013b)

### 2.2.2 Techniques for solving common problems

Google wants to keep Protocol Buffers simple to use and, as a consequence of that, it is has decided to only include features needed by a majority of the

---

<sup>5</sup><http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns>

users. Google has on their techniques page<sup>6</sup> some suggestions of how certain features can be added to Protocol Buffers. The following bulleted items are a summary of the information available on that page. (Google, 2012e)

- Messages are not self-delimiting, which can be a problem when streaming multiple messages. To overcome this issue Google suggests that every message is prefixed with the length of the message.
- Protocol Buffers was not designed with large data sets in mind, and if the size of messages reaches into the megabytes, it is recommended that they be split into smaller pieces.
- Protocol Buffers cannot determine the type of a message based on the contents alone. Not knowing the type of a message might pose a problem in a scenario where the type of the message could be one of several. As a way to determine the message type Google suggests that all possible message types be wrapped as optional fields inside a container message. Accessors can then be used to determine the type of a message. Another possible solution is to introduce a required field, which specifies the message type in the container message. If the number of message types is large a solution that uses extensions might be preferred to avoid specifying all possible message types.
- Although a message is not self-describing out of the box it is possible to achieve through the use of reflection, which is supported both on C++ and Java. It is not fully implemented by Google and therefore requires that the user write tools to manipulate self-describing messages. Google states that the reason for not fully implementing it is that they have not had any use for it themselves.

Inside Google, Protocol Buffers are used for persistent storage of data and in RPC systems (Google, 2012b). According to Google (2012b) ‘Protocol buffers are now ... [their] lingua franca for data ...’.

### 2.2.3 Protocol Buffers binary encoding format

This section describes the encoding used by Protocol Buffers to serialize data. All of the material in this section is based on information provided by Google (2012c, 2013b).

Fields in a serialized message are stored as key-value pairs. The key in the key-value pair consists of a field number, also known as a tag, and a wire type. A tag is an identification number assigned to each field in the `.proto` file and the wire type is used to indicate how the binary bits of a serialized field should be interpreted. It should not be confused with the type of a field previously described. Every field type has a wire type associated with it, see table 2.1 for a description of available wire types and their associated

---

<sup>6</sup><https://developers.google.com/protocol-buffers/docs/techniques>

field types. The attentive reader might be wondering what purpose the wire type serves, since it is possible to deduce the wire type from a field type. Protocol Buffers was designed to be able to handle unknown fields i.e. fields not specified in the `.proto` file guiding the interpretation of a serialized message. The wire type is added to the key to make it possible to skip unknown fields encountered during parsing of a serialized message. To compose a key from a field number and a wire type the following formula is used  $(field\_number) \ll 3 \mid wire\_type^\dagger$ . A key is then serialized in the same way as a 32-bit unsigned integer (uint32) value.

Table 2.1: Wire types Protocol Buffers

Type	Meaning	Used for
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

Table 2.2 describes the scalar field types.

Table 2.2: Type

Type	Explanation
double	Double-precision floating-point
float	Single-precision floating-point
int32	32-bit integer
int64	64-bit integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
sint32	32-bit signed integer
sint64	64-bit signed integer
fixed32	32-bit unsigned fixed integer
fixed64	64-bit unsigned fixed integer
sfixed32	32-bit fixed integer
sfixed64	64-bit fixed integer
bool	Boolean <sup>7</sup>
string	UTF-8 encoded string
bytes	Arbitrary sequence of bytes

<sup>†</sup> *field\_number* is shifted left 3 bit positions and then combined with *wire\_type* using bitwise OR.

<sup>7</sup>Stored as int32. False when 0 otherwise true.

Varint is a variable length integer encoding. The first bit in each byte of a varint indicates if the next byte is also a part of the varint, in that case, the first bit is 1, otherwise it is 0. The remaining 7 bits in each byte are used store the value in a two's complement format with the least significant group first. The following example illustrates how a how to encode  $396_{10}$  as a varint.

$$396_{10} = 0001\ 1000\ 1100_2$$

$0001\ 1000\ 1100_2$  is clearly too big to fit inside one varint encoded byte. It must therefore be split into groups of 7 bits.

$$0001\ 1000\ 1100_2 \xrightarrow{\text{split}} \underbrace{000\ 0011}_{\text{group 0}} \underbrace{000\ 1100}_{\text{group 1}}$$

Groups must now be reordered so that the least significant group comes first.

$$\underbrace{000\ 0011}_{\text{group 0}} \underbrace{000\ 1100}_{\text{group 1}} \xrightarrow{\text{reorder}} \underbrace{000\ 1100}_{\text{group 1}} \underbrace{000\ 0011}_{\text{group 0}}$$

And as a final step a bit is added to each group to indicate if more bytes follow.

$$\underbrace{000\ 1100}_{\text{group 1}} \underbrace{000\ 0011}_{\text{group 0}} \xrightarrow{\text{MSB}} \underbrace{1000\ 1100}_{\text{byte 0}} \underbrace{0000\ 0011}_{\text{byte 1}}$$

The difference between signed integers and ordinary integers is that ordinary integers store numbers in two's complement, whereas signed integers store numbers using a ZigZag encoding<sup>8</sup>, see table 2.3 for examples of sint32 encoded values. The ZigZag encoding is used to minimize the number of bytes needed to represent negative numbers, since negative numbers in two's complement form needs to be represented with a varint using the maximum amount of bytes required to represent a type.

Table 2.3: Examples of sint32 encoded numbers

Signed original	Encoded as
0	0
-1	1
1	2
-2	3
2	4
...	...
2147483647	4294967294
-2147483648	4294967295

<sup>8</sup>ZigZag encoding is named after the way it 'zig-zags' between negative and positive values. The following formula illustrates how to convert *value* to sint32 ( $value \ll 1$ )<sup>^</sup> ( $value \gg 31$ ). *value* is shifted left one bit position and then combined with *value* shifted right 31 bit positions using bitwise XOR.

32-bit and 64-bit wire types encode numbers in a little-endian byte order using a fixed size length (32-bit uses 4 bytes and 64-bit uses 8 bytes).

Enum fields storing enumeration values are stored in the same way as 32-bit integer (int32) fields. This means that enumeration literals can take on negative values, although it is not recommended since negative values are stored inefficiently using 32-bit integer (int32) fields.

String, bytes, embedded messages and packed repeated fields are stored as length-delimited fields. A length-delimited field has a varint encoded length prefix to indicate the length of the field.

To understand what a packed repeated field is an understanding of how Protocol Buffers orders the fields in a message is needed. Fields in a message are not guaranteed to be ordered sequentially by the field numbers in the message, even though the implementations provided by Google adhere to this rule. It is even possible for repeated fields to be interleaved with other fields, although the order among the repeated fields is preserved. To avoid this, packed repeated fields, specified by setting the option `packed` to `true`, can be used. Packed repeated fields are simply packed into a length-delimited field with repeated elements encoded in the field as normal, except the tag that is not repeated. A limitation of packed fields is that only 32-bit and 64-bit varint formats can be declared packed.

### 2.2.4 Example

Listing 2.1 contains a short example, based on `addressbook.proto`<sup>9</sup>, to exemplify the use of `.proto` files.

Listing 2.1: `person.proto`

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

The `Person` message contains two required fields; `name` and `email`. Data that is to be serialized must contain all required fields. The optional fields, `type` and `email`, are optional and can be present but are not required inside a message. There is also another message, `PhoneNumber`, contained inside

<sup>9</sup><https://developers.google.com/protocol-buffers/docs/cpptutorial>



Person. The PhoneNumber message is declared repeated and can therefore appear zero or more times inside a message. Every field inside a message also has a tag associated with it, a unique number used to identify the field when it has been serialized. The example also shows how enum fields can be declared; in this case, type is declared to be of type PhoneType and has a default value of HOME.

From this description a compiler, such as *protoc*, can generate code for a target language, which the developer can then use to serialize and deserialize objects of type Person.

The best way to illustrate how data serialized by Protocol Buffers can look like is by showing an example. Listing 2.2 shows a message of type Person formatted according to Protocol Buffers text format. In this example a person with a name, id and a phone number has been created.

Listing 2.2: Text format serialization

```
name: "John Doe"
id: 2
email: "john@doe.com"
phone {
  number: "31415926"
  type: MOBILE
}
```

The binary encoded version of the same message is shown in hexadecimal notation in listing 2.3.

Listing 2.3: Binary serialization

```
0A 08 4A 6F 68 6E 20 44 6F 65 10 02 1A 0C 6A 6F 68 6E 40
64 6F 65 2E 63 6F 6D 22 0C 0A 08 33 31 34 32 35 39 32 36
10 00
```

## 2.3 JavaScript Object Notation (JSON)

Crockford (2006) describes JSON as a ‘... lightweight, text-based, language-independent data interchange format’ (p. 1). JSON, as the name implies, uses an object notation that is based on JavaScript’s notation for object literals. Although the notation is based on JavaScript, the format is truly language-independent and support for a wide variety of languages exists (Crockford, n.d.).

To encode data JSON uses only two types of structures: objects and arrays.

An object consists of an unordered set of name/value pairs. A left brace indicates the beginning of an object and a right brace the end of an object. In between the braces an arbitrary number of name/value pairs can be present, separated by commas. A name/value pair consists of a name enclosed in double quotes (a string) followed by a colon and a value.

An array consists of an ordered collection of values. A left bracket indicates the beginning of an array and a right bracket indicates the end. In between

the brackets an arbitrary number of values can be present, separated by commas.

A value is either a string, number, object, array or one of the predefined values true, false or null. A string consists of an arbitrary number of Unicode characters enclosed in double quotes, except \ and " which must be escaped. A number is a sequence of digits and can be positive, negative, contain a decimal point and have an exponent.

The information above is a summary of the format used by json to encode data, a complete definition can be found at <http://www.json.org>. However, the format is simple and there is not much to add that has not already been covered by the summary. As Crockford (n.d.) states, JSON's simple syntax makes data formatted according to it easy to interpret by both human and computer.

JSON is made available with a permissive license, which allow the user to copy, modify, publish, sublicense and sell software based on JSON as long as a copyright notice is included with any distributed copies. Having said that, the license includes the following clause 'The software shall be used for Good, not Evil'. (Crockford, n.d.)

### 2.3.1 GNATColl.JSON

*GNAT Component Collection JSON (GNATColl.JSON)* is a software library package, which provides Ada constructs for easy creation and parsing of JSON encoded data (AdaCore, n.d.).

Listing 2.4 shows a simple example illustrating the use of GNATColl.JSON.

Listing 2.4: GNATColl.JSON example

```
pragma Ada_05;
with GNATCOLL.JSON;    use GNATCOLL.JSON;
with Ada.Text_IO;      use Ada.Text_IO;

procedure Main is
  Person : JSON_Value := Create_Object;
  Phone_Numbers : JSON_Array := Empty_Array;
  Home_Phone : JSON_Value := Create_Object;
  Mobile_Phone : JSON_Value := Create_Object;

  MOBILE : constant := 0;
  HOME : constant := 1;
begin
  -- Set name
  Person.Set_Field("name", "John Doe");

  -- Set id
  Person.Set_Field("id", Create(23));

  -- Set home phone
  Home_Phone.Set_Field("number", "314 159 26");
  Home_Phone.Set_Field("type", HOME);

  -- Set mobile phone
  Mobile_Phone.Set_Field("number", "271 82 81");
  Mobile_Phone.Set_Field("type", MOBILE);

  -- Add home phone and mobile phone to phone numbers
```

```
Append(Phone_Numbers, Home_Phone);
Append(Phone_Numbers, Mobile_Phone);

-- Add phone numbers to person
Person.Set_Field("phone numbers", Phone_Numbers);

-- Print person
Put_Line (Person.Write);
end Main;
```

The example shown in listing 2.4 is the JSON equivalent of the Protocol Buffers example shown in listing 2.1. To begin with a JSON object named `Person` is setup to store information about an individual. Fields are then created inside the `Person` object to store the individual's name and id. Two JSON objects are then setup to store a home and mobile phone number. The phone numbers are then stored inside a JSON array named `Phone_Numbers`. Finally the array is stored inside the `Person` object and written to *standard output (stdout)*. Output of the example program is shown in listing 2.5.

Listing 2.5: Output from GNATColl.JSON example  
(output has been formatted to make it easier on the eye)

```
{
  "id": 23,
  "name": "John Doe",
  "phone numbers": [
    {
      "number": "314 159 26",
      "type": 1
    },
    {
      "number": "271 82 81",
      "type": 0
    }
  ]
}
```

AdaCore, the developers of GNATColl.JSON, has made the software library available under the *GNU General Public License (GPL)* version 3 with an added exception. The exception, known as the *GCC Runtime Library Exception*, allows redistribution of GPL software under a different license as long as set of conditions are fulfilled (Free Software Foundation, 2013).



## Chapter 3

# Ada code generation for Google Protocol Buffers

Ada code generation for Google Protocol Buffers has been developed as part of this thesis. In this chapter a description of the developed software solution is provided along with backgrounds and motivations for major design decisions.

### 3.1 Ada

The history of Ada goes back to the 1970s when the United States Department of Defense sponsored the development of a new programming language that later would become known as Ada 83, after the year it was certified by the American National Standards Institute (Barnes, 2006). Since then several major versions have been released; as of this writing the most recent stable release is Ada 2012 (Ada Resource Association, n.d.).

Ada is a general-purpose, statically typed, imperative, object oriented programming language designed from the beginning to be used in the development of large scale applications. Programmers used to other high-level languages might find Ada programs a bit verbose, due the language being designed with an emphasis placed on readability instead of the writing of succinct code. (Cohen, 1995)

The remaining part of this section is dedicated to a comparison of Ada to Java and C++. Focus in the comparison is placed on differences that impact design of Ada code generation support for Ada.

#### 3.1.1 Packages and types

To group related source code entities, Ada uses a module system construct called a package. Packages are divided into two parts: body and specification. A package specification provides the interface for a package and is commonly

placed inside a separate file with filename suffix `.ads`. A package body provides the implementation for a package and is commonly placed inside a separate file with the filename suffix `.adb`. Packages can also be defined inside other packages, which is referred to as nesting of packages. Another way to organize packages hierarchically is through the use of child packages. A child package has another package as a parent and can be used as way to separate interface from implementation or simply partition a system into a tree-like structure. (Barnes, 2006)

Packages in Ada does not automatically specify a data type, instead it is up to the programmer to declare types inside a package. A declared type's name is independent of package name and must be marked as tagged to provide support for object oriented constructs such as inheritance and dynamic polymorphism<sup>1</sup>. The so-called *tagged type* has a tag that can be used to determine the type of the object and also the functions and procedure that belong to the type. (Barnes, 2006)

The class construct familiar from C++ and java is not present in Ada. However, the definition of a tagged type inside a package is often used to model the behaviour of class. (Brosgol, 2008)

### 3.1.2 Predefined types

The predefined type system available in Ada provides types similar to those available in C++ and Java, but a few differences exists that are worth mentioning.

- A variable of type Integer, which at first glance appears to be similar to the int type defined in C++ and Java, is only guaranteed to support values in the range  $-2^{15} + 1 \dots 2^{15} - 1$  as specified by Taft, Duff, Brukardt, Ploedereder and Leroy (2006a) in section 3 of the Ada Language Reference Manual.
- Standard String variables are represented by Character arrays. A consequence of this is that they have a fixed length that is determined by the declaration or an initial assignment. A String can therefore not change its size after its declaration, which makes them somewhat inflexible. To allow for more flexibility Ada has introduced packages that support bounded and unbounded strings of varying length. (Barnes, 2006)
- String variables of all types are indexed using the type Positive, a subtype of Integer that only includes the positive values in the Integer range (Barnes, 2006). A string variable can therefore only be guaranteed to hold up to  $2^{15}$  characters, which for those not comfortable with powers of two is 32 768 characters.

---

<sup>1</sup>Barnes (2006) uses the term 'dynamic polymorphism' to describe a situation where a function or procedure call is determined at run-time based on the type of the object.

- Ada has a powerful type system, which supports the definition of new types (scalar and composite). New types can be defined with only range and precision explicitly specified, thus leaving the job of determining in-memory representation to the compiler. Run-time checks for violations of range constraints are automatically generated by the Ada compiler for a new type, but can be selectively suppressed by the programmer. Operators can also be redefined for existing types and defined for new types. (Barnes, 2006)

### 3.1.3 Memory management

Objects in Ada can be allocated on the stack or on the heap. An object declared inside a function or procedure is automatically allocated on the stack, whereas the keyword `new` is used to allocate objects on the heap. (Barnes, 2006)

There are two or three different techniques, depending upon how you look at it, that can be used in Ada to reclaim previously allocated storage (Brosgol, 2008). Ada has a package called `Ada.Finalization` that gives the user the ability to control what happens when objects are initialized and when they are finalized (Barnes, 2006). This is similar to how memory is handled in C++ with constructors and destructors. Storage can also be reclaimed at specific point in execution using the package `Ada.Unchecked_Deallocation` (Barnes, 2006). Finally some Ada implementations have built-in automatic garbage collection like Java (Brosgol, 2008).

## 3.2 Software requirements

Appendix A contains software requirements elicited during the planning stages of development. Inspiration for most of the requirements came from the development of a throwaway prototype, as well as from existing compilers with support for other target languages.

## 3.3 Plug-in or standalone application?

Version 2.3.0 of `protoc`, Google's official Protocol Buffers compiler, was released in January 2010. The release introduced a new plug-in system that allows third-party developers to extend `protoc` with support for, among other things, new programming languages. The new plug-in system simplifies the process of developing support for new languages, since plug-ins can take advantage of `protoc`'s existing parser. The sharing of a single parser implementation and the fact that it allows code generators to provide a consistent interface makes Google recommend that all third-party code generators be written as plug-ins. (Google, 2012d)

A decision to implement code generation as a plug-in instead of as a standalone application was thus made based on Google's recommendations.

## 3.4 Implementing a plug-in

Google provides two alternatives for developing plug-ins, developers can either write a plug-in for code generation using a C++ API or interface with `protoc` at the protocol buffer level. (Google, 2013c)

To interface with `protoc` at the protocol buffer level an executable plug-in must first be created. `protoc` can then be told to invoke the executable plug-in and write the parsed `.proto` files to the plug-ins standard input. The parsed `.proto` files that are written to standard input are formatted according to `plugin.proto`<sup>2</sup> and `descriptor.proto`<sup>3</sup>, the plug-in must thus deal with data that is in binary protocol buffer wire format. (Google, 2013c)

Writing a plug-in for code generation using the C++ API does not require the developer to handle data written in a binary protocol buffer wire format, but it does require that the plug-in interface with C++ code. A plug-in using the C++ API can thus be written in C++ or in an arbitrary language of choice using a C++ language binding.

Developing a plug-in in C++ can be done by writing an implementation of `CodeGenerator` and linking against `libprotobuf` and `libprotoc`. `protoc` can then be told to invoke the main function inside the `CodeGenerator` class. (Google, 2013c)

## 3.5 Deciding how to implement the plug-in

Deciding whether to implement the plug-in using the C++ API or by interfacing with `protoc` at the protocol buffer level, required a bit of investigation to make an informed decision. Early on in the development process, a throwaway prototype with basic functionality was constructed, as a means to explore the problem domain and aid in elicitation of requirements. The prototype was written in Ada and constructed as a plug-in that interfaced with `protoc` at the protocol buffer level.

The reason behind Ada as a choice of programming language for the prototype was that it would give a thorough understanding of a user's experience interfacing with the generated Ada code. Another factor, that played a role in the decision to use Ada for code generation, was that a single programming language and development environment could be used throughout development.

---

<sup>2</sup><https://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/plugin.proto>

<sup>3</sup><https://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/descriptor.proto>



Further research into the C++ API provided by Google for code generation and experiences gained by writing the prototype made it clear that the benefits of using the C++ API for code generation outweighed the drawbacks. The C++ API provides benefits such as easy indentation, variable substitutions inside character strings, logging of error message etc. Dealing with the raw binary message format, when writing the prototype, also proved to be a bit more cumbersome than expected.

A decision was made to implement the Ada code generation using C++ and not through an Ada language binding for the C++ API. An Ada language binding would have required generating or writing Ada specifications for C++ header files. This approach would have entailed more work than simply writing a C++ implementation, without any substantial benefits, and it was therefore not considered.

### 3.6 Challenges faced during development

As described in section 3.1.2, the type system used by Ada differs from the type system used by C/C++, which is similar to the type system used by Protocol Buffers. Fortunately, Ada has special types defined inside the package Interfaces, which are provided to directly interface with C/C++ and other foreign languages (Barnes, 2006). Another reason to use these types is that the standard types defined by Ada lack support for bit-shifts, which is offered for the unsigned types defined by the Interfaces package.

GNAT represents the Integer type using a 32-bit value, even though the standard only requires it to support a range provided by a 16-bit value (AdaCore, n.d.). Because of this the size limitation on the String type becomes a nonissue and it can be used as a type for storing `.proto` defined string fields. Other modern Ada compilers will probably use a similar representation. However, if one is unsure about the implementations details of an Ada implementation it should be included in the documentation accompanying the compiler. Every standard compliant Ada implementation is required to document implementation-defined characteristics, such as size of the predefined integer types (Taft, Duff, Brukardt, Ploedereder & Leroy, 2006b). An Ada compiler, which does not represent the Integer type using at least 32-bits, would suffer from string length limitations compared to other implementations. A possible workaround would be to make modifications to the type system used by generated Ada code.

Circular dependencies between messages are allowed by Protocol Buffers, see listing 3.1 for a contrived example illustrating such a dependency between two messages. Circular dependencies complicate Ada code generation, since messages correspond to tagged types defined in separate packages in the generated code.

Ada does not permit circular dependencies between types in different

packages using regular with clauses<sup>4</sup>. However, Ada 2005 introduced a new construct called a limited with clause with the intention of simplifying use of mutually dependent types. A limited with clause gives an incomplete view of the types in the package being with'ed, which restricts usage of types in the with'ed package. An incomplete view of a type forbids declaring variables of that type, instead access types (pointers) must be used. (Barnes, 2008)

As a result package A can't simply use a regular with clause to refer to Package B and vice versa it must be a limited with clause<sup>5</sup>. The restrictions imposed by limited with clauses have far reaching implications, which impacts code generation for messages and enumerations declared inside messages.

Listing 3.1: Circular dependency example

```
message A {  
  optional B message_b = 1;  
}  
  
message B {  
  optional A message_a = 1;  
}
```

## 3.7 Structure of the developed software solution

The overall structure of the software solution closely resembles the structure of Google's implementation of Protocol Buffers. This is no coincidence, but a conscious decision since there is no point in reinventing the wheel. Adaptations have of course been made to fit the Ada language, where such decisions have had to be made. It should be noted that the close resemblance between the official implementation of Protocol Buffers and the developed plug-in has been somewhat diminished by the scaled back feature set of the developed plug-in.

### 3.7.1 Supporting libraries

Generated Ada code files rely on a library of Ada code that is independent of the definitions made inside a `.proto` file. Figure 3.1 shows the compilation phases for a simple example program, which illustrates the use of the Ada code library. The dotted arrows indicate usage or rather inclusion of files, which provides functionality to the including package.

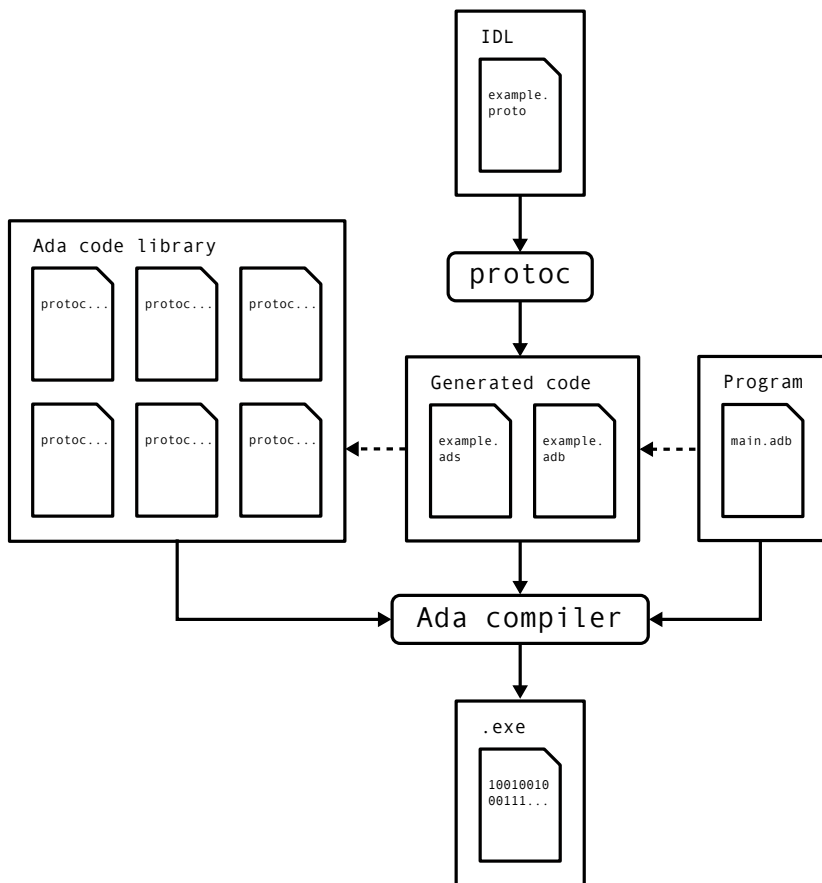
The following is a list of packages included in the library:

---

<sup>4</sup>A with clause is the Ada equivalent of a C++ include directive.

<sup>5</sup>Only one of the with clauses needs to be changed to a limited with clause to break the cyclic dependency chain. Generating code for messages makes this distinction irrelevant, since it would be nigh on impossible to distinguish the relationship between two messages defined in `.proto` file. Limited with clauses are therefore used for all packages that needs to be with'ed because of embedded message fields.

Figure 3.1: Compilation of example program



### Message

This package contains an abstract data type that every generated message type inherits from. It provides functionality to read a message from a stream, write a message to a stream, merge a message with another message, copy a message, get the full message name, determine if the message has been initialized etc.

### Coded\_Input\_Stream

This package is used by generated message types to read serialized data from an input stream. The package provides helper functions for reading binary encoded protocol buffer fields from a stream. This package is not intended to be used directly by a user of the API, except in cases where stream behaviour needs to be customized.

### Coded\_Output\_Stream

This package is used by generated message types to write data to an output stream in the binary protocol buffer format. The package provides helper functions for writing protocol buffer fields encoded in the binary protocol buffer format. This package is not intended to be used directly by a user of the API.

**Wire\_Format**

This package specifies a list of Ada types corresponding to protocol buffer types.

**Generated\_Message\_Utilities**

This package provides utilities for handling default values to generated message types.

### 3.7.2 Code generator design

Code generation for C++, Java and Python, inside the official implementation of Protocol Buffers, uses Google's C++ API plug-in system (Google, 2012a). Overall, the design of the code generation is very similar for Java and the C++, whereas Python differs quite a bit. The reason for this is that the Python implementation uses reflection to generate message classes at runtime (Robinson, n.d.). The generation of message classes at runtime can generate a significant overhead to the serialization and deserialization of messages (Robinson, n.d.). This alone is enough of a reason for not considering it as a possible design, since performance is such an important factor. The choice was thus made to do the code generation at compile time, similar to how the code generation is done for C++ and Java.

The general structure of the official code generation for Java and C++ is largely language independent, and as consequence of this, it lends itself well to Ada code generation. The main classes are:

**AdaGenerator**

Inherits from `CodeGenerator` and overrides the member function `Generate` that is used for code generation. `Generate` is called by `protoc` after the `.proto` files given as input have been parsed.

**MessageGenerator**

Class used by `FileGenerator` to generate package bodies and specifications for messages. It is also used to generate type independent code for fields.

**FieldGenerator**

An abstract class that defines an interface for derived `FieldGenerators`. It includes a factory method<sup>6</sup> that can be used to construct specialized `FieldGenerators` (`PrimitiveFieldGenerator`, `StringFieldGenerator` etc.).

---

<sup>6</sup>A factory method is an example of a creational design pattern described by Gamma, Helm, Johnson and Vlissides (1994) in their definitive work on design patterns.

**PrimitiveFieldGenerator**

Generates code for primitive numeric field types, except enum. Primitive fields are fields of type `int32`, `uint32`, `sint32`, `fixed32`, `sfixed32`, `int64`, `uint64`, `sint64`, `fixed64`, `sfixed64`, `float`, `double`, `bool`, `bytes` and `enum`.

**StringFieldGenerator**

Generates code for string fields.

**EnumGenerator**

Generates supporting code for enumeration fields.

**EnumFieldGenerator**

Generates code for enum fields.

**MessageFieldGenerator**

Generates code for nested messages.

**ExtensionGenerator**

Generates code for extensions.

**FileGenerator**

Generates package body and specification for a `.proto` file.

Not mentioned in the list above are classes related to the code generation for repeated fields. All classes in list inheriting from `FieldGenerator` also have a corresponding class with the same name except for the prefix `Repeated`. A detailed description of repeated fields can be found in section 2.2.3.

Figure 3.2 shows a class diagram that displays how the main classes used for code generation relate to one another. `CodeGenerator` is provided by Google and technically not included in the developed code generation solution for Ada, but is shown the diagram because of its importance in code generation process that is described in the next section.

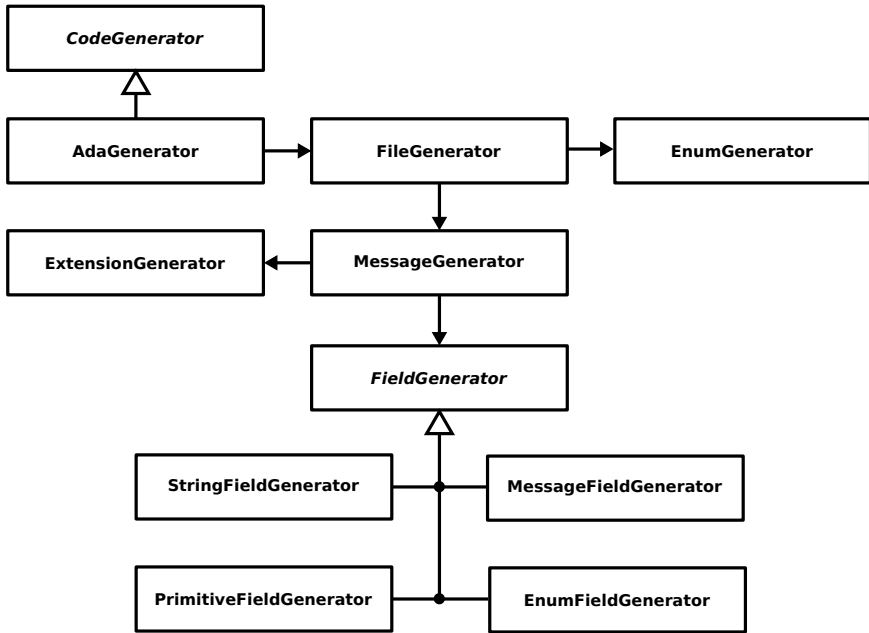
The code generation solution also includes utility functions that are Ada specific, which are defined inside `ada_helpers.h`. These so-called utility functions are mostly concerned with the translation of names, types, and values defined in `.proto` files to the Ada equivalent used in generated code.

### 3.7.3 Code generation for Ada

The first step in the code generation process is the parsing of `.proto` files done by `protoc`. To describe a parsed `.proto` file a `FileDescriptor` object is built by `protoc`. A `FileDescriptor` contains information about a `.proto` file and any message defined inside the `.proto` file. After a `FileDescriptor` object has been built by `protoc` it is then passed to a `CodeGenerator`. It is at this point a code generation plug-in takes over responsibility for generating code that is specific to the target language.

The next step is to generate code for the target language, in this case Ada. As the code generator plug-in for Ada takes over it creates a `FileGenerator`

Figure 3.2: Class diagram



object to generate body and specification files for a package with the same name as the parsed `.proto` file. The generated package is then used as a parent for all child packages generated from message definitions inside the `.proto` file. The same child package structure is also used for messages defined inside other messages, so called nested messages.

MessageGenerator objects are then created by the FileGenerator object to generate body and specification code for every message specified inside the `.proto` file. MessageGenerator objects are also used to generate boilerplate code for fields, along with code needed to implement inherited functions and procedures from the abstract message type, described in section 3.7.1.

For every field defined inside a message, a FieldGenerator object of appropriate type is created by the MessageGenerator tasked with generating code for said message. This is accomplished with the help the FieldGeneratorMap class, which is used to create objects based on information stored in a FieldDescriptor object. Through the use of virtual member functions<sup>7</sup> code specific to a certain field type is then generated by a suitable FieldGenerator.

The process as it has been described in this section provides a rudimentary explanation of how code generation for Ada is done using the developed

<sup>7</sup>A virtual member function is C++ terminology for a function, marked by the keyword `virtual`, that can be overridden to provide dynamic polymorphism (Lippman, Lajoie & Moo, 2005).

plug-in. Some details have obviously been left out in the description of the code generation process, but the description should provide details necessary to give the reader a rough idea of how code generation is carried out by the developed plug-in.

## 3.8 Testing

Unit testing has been partitioned into three test suites with a focus on testing serialization, deserialization and generated message functionality. The test suites focusing on testing serialization and deserialization test the underlying Ada code library, while the test suite focusing on testing generated message functionality tests generated API code. All test suites implicitly test code generation since they rely on packages generated from messages defined in `.proto` files. Appendix C provides a description of the test suites.

### 3.8.1 Unit testing framework

The supporting library code, which is written in Ada, has been unit tested using *AUnit*<sup>8</sup>. Ahven<sup>9</sup> and VectorCAST/Ada<sup>10</sup> were also considered when deciding on a unit testing platform. VectorCAST/Ada was decided against, because of the proprietary software license that would require obtaining a license for use. Ahven and AUnit on the other hand are both available under open-source licenses. AUnit was chosen instead of Ahven for two reason:

- AUnit is integrated into *GNAT Programming Studio (GPS)*, the integrated development environment used to develop supporting libraries for Ada code generation.
- AUnit is developed and maintained by AdaCore, the company behind GPS, whereas Ahven is the work of single developer.

## 3.9 API

As has been previously mentioned, a message declared inside a `.proto` file corresponds to a package containing a type inheriting from a base message type declared inside the Message package. Functions and procedures that are common among generated message types are specified by this base message type. The interface provided by the base message type is similar to the official C++ API for generated message classes.<sup>11</sup> It was a conscious decision to model the interface on the C++ API, since developers already familiar with the C++ API can start using the Ada message API almost immediately.

---

<sup>8</sup><http://libre.adacore.com/tools/aunit>

<sup>9</sup><http://ahven.stronglytyped.org>

<sup>10</sup><https://www.vectorcast.com/software-testing-products/ada-unit-testing>

<sup>11</sup><https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf>. message contains detailed information about the official C++ message class API.

### 3.9.1 Messages

Serialization procedures are provided for Ada library streams as well as for Coded\_Output\_Stream and Coded\_Input\_Stream.<sup>12</sup> Coded\_Output\_Stream and Coded\_Input\_Stream are library stream wrappers that add features specific to Protocol Buffers. Serialization procedures for library streams are therefore provided as a convenience to the user, as they do little more than call the Coded\_Input\_Stream or Coded\_Output\_Stream equivalent. So called partial procedures allow serialization of message missing required fields.

Listing 3.2 shows procedures provided for serializing a message. The procedure Serialize\_With\_Cached\_Sizes is called by the other procedures to do the actual serialization, but can also be called by the user directly to avoid calculating the message size every time a message is serialized.

Listing 3.2: Serializing procedures

```
procedure Serialize_To_Output_Stream
procedure Serialize_To_Coded_Output_Stream
procedure Serialize_Partial_To_Output_Stream
procedure Serialize_Partial_To_Coded_Output_Stream
procedure Serialize_With_Cached_Sizes
```

Listing 3.3 shows procedures provided for parsing a message.

Listing 3.3: Parsing procedures

```
procedure Parse_From_Input_Stream
procedure Parse_From_Coded_Input_Stream
procedure Parse_Partial_From_Input_Stream
procedure Parse_Partial_From_Coded_Input_Stream
```

Listing 3.4 shows procedures provided for merging a message with a message read from a stream.

Listing 3.4: Merging procedures

```
procedure Merge_From_Input_Stream
procedure Merge_From_Coded_Input_Stream
procedure Merge_Partial_From_Input_Stream
procedure Merge_Partial_From_Coded_Input_Stream
```

Listing 3.5 shows the remaining procedures provided by the message interface. Merge and Copy do exactly what their name suggests. Clear is used to clear a message's fields and restores default values specified in the .proto files. Get\_Type\_Name returns a message's full type name. Byte\_Size is used to recursively calculate a message's serialized size. Get\_Cached\_Size returns the message size previously calculated by Byte\_Size. Is\_Initialized determines if all required fields have been set.

Listing 3.5: Miscellaneous procedures and functions

<sup>12</sup>Library stream types offer an implementation independent way of sequentially accessing elements of different types. A stream can for instance be implemented to read and write from a file, internal buffer or a network channel. (Taft, Duff, Brukardt, Ploedereder & Leroy, 2006c)



```

procedure Merge
procedure Copy
procedure Clear
function Get_Type_Name
function Byte_Size
function Get_Cached_Size
function Is_Initialized

```

### 3.9.2 Fields

For each field defined inside a `.proto` file a set of accessors and mutators are generated. This section describes the set of generated functions for all field types.<sup>13</sup> Table 3.1 lists the scalar value types, described in section 2.2.3, and their Ada representation.

Table 3.1: Scalar value types and their representation

Type	Name	Representation
double	PB_Double	Interfaces.IEEE_Float_64
float	PB_Float	Interfaces.IEEE_Float_32
int32	PB_Int32	Interfaces.Integer_32
int64	PB_Int64	Interfaces.Integer_64
uint32	PB_UInt32	Interfaces.Unsigned_32
uint64	PB_UInt64	Interfaces.Unsigned_64
sint32	PB_Int32	Interfaces.Integer_32
sint64	PB_Int64	Interfaces.Integer_64
fixed32	PB_UInt32	Interfaces.Unsigned_32
fixed64	PB_UInt64	Interfaces.Unsigned_64
sfixed32	PB_Int32	Interfaces.Integer_32
sfixed64	PB_Int64	Interfaces.Integer_64
bool	PB_Bool	Boolean
string	PB_String	String
bytes	PB_String	String

#### Singular numeric fields

A singular numeric<sup>14</sup> field can be defined as in listing 3.6.

Listing 3.6: Singular numeric field definition

```

optional int32 foo = 1;
required int32 foo = 1;

```

<sup>13</sup>Portions of the material presented in this section are modifications based on work created and shared by Google and used according to the terms described in Creative Commons 3.0 Attribution License. See <https://developers.google.com/protocol-buffers/docs/reference/cpp-generated> for the original source material.

<sup>14</sup>the following numeric types are available: double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32 and sfixed64

The set of procedures and functions that would be generated from either one of the field definitions in listing 3.6 is shown in listing 3.7. Singular numeric fields of other types would have PB\_Int32 replaced with the corresponding Ada type shown in table 3.1.

Listing 3.7: Generated functions for singular numeric fields

```
function Has_Foo
  (The_Message : in ...)
  return Boolean;
-- Returns true if field has been set.

procedure Clear_Foo
  (The_Message : in out ...);
-- Clears the value of the field. After calling this, Has_Foo will
-- return False and Get_Foo will return the default value.

function Get_Foo
  (The_Message : in ...)
  return Protocol_Buffers.Wire_Format.PB_Int32;
-- Returns the current value of the field. If the field is not
-- set, it returns the default value.

procedure Set_Foo
  (The_Message : in out ...;
   Value : in Protocol_Buffers.Wire_Format.PB_Int32);
-- Sets the value of the field. After calling this, Has_Foo will
-- return True and Get_Foo will return Value.
```

### Singular string/bytes fields

Singular string and bytes fields can be defined as in listing 3.8.

Listing 3.8: Singular string/bytes field definition

```
optional string foo = 1;
required string foo = 1;
optional bytes foo = 1;
required bytes foo = 1;
```

Listing 3.9 shows the set of functions and procedures that would be generated from any one of the field definitions in listing 3.8.

Listing 3.9: Generated functions for singular string/bytes fields

```
function Has_Foo
  (The_Message : in ...)
  return Boolean;
-- Returns True if the field is set.

procedure Clear_Foo
  (The_Message : in out ...);
-- Clears the value of the field. After calling this, Has_Foo will
-- return False and Get_Foo will return the default value.

function Get_Foo
  (The_Message : in ...)
  return Protocol_Buffers.Wire_Format.PB_String;
-- Returns the current value of the field. If the field is not
-- set, returns the default value.

procedure Set_Foo
```

```
(The_Message : in out ...;
  Value : in Protocol_Buffers.Wire_Format.PB_String);
-- Sets the value of the field. After calling this, Has_Foo will
-- return True and Foo will return Value.

function Release_Foo
  (The_Message : in out ...)
  return Protocol_Buffers.Wire_Format.PB_String_Access;
-- Releases the ownership of the field and returns a pointer to
-- the allocated string. After calling this, caller takes the
-- ownership of the allocated string, Has_Foo will return False,
-- and Get_Foo will return the default value.
```

## Singular enum fields

Enumerations can be defined as in listing 3.10.

Listing 3.10: Enum definition

```
enum Bar {
  BAR_VALUE = 1;
}
```

Using the enum type defined in listing 3.10 a singular enum field can be defined as in listing 3.11.

Listing 3.11: Singular enum field definition

```
optional Bar foo = 1;
required Bar foo = 1;
```

Listing 3.12 shows the set of functions and procedures that would be generated from either one of the field definitions in listing 3.11.

Listing 3.12: Generated functions for singular enum fields

```
function Has_Foo
  (The_Message : in ...)
  return Boolean;
-- Returns True if the field is set.

procedure Clear_Foo
  (The_Message : in out ...);
-- Clears the value of the field. After calling this, Has_Foo will
-- return False and Get_Foo will return the default value.

function Get_Foo
  (The_Message : in ...)
  return Bar;
-- Returns the current value of the field. If the field is not
-- set, returns the default value.

procedure Set_Foo
  (The_Message : in out ...;
   Value : in Bar);
-- Sets the value of the field. After calling this, Has_Foo will
-- return True and Get_Foo will return Value.
```

## Singular embedded message fields

A message can be defined as in listing 3.13.

Listing 3.13: Message definition

```
message Bar {}
```

Using the message type defined in listing 3.13 a singular embedded message field can be defined as in listing 3.14.

Listing 3.14: Singular embedded message field definition

```
optional Bar foo = 1;
required Bar foo = 1;
```

Listing 3.15 shows the set of functions and procedures that would be generated from either of the one field definitions in listing 3.14.

Listing 3.15: Generated functions for singular embedded message fields

```
function Has_Foo
  (The_Message : in ...)
  return Boolean;
-- Returns True if the field is set.

procedure Clear_Foo
  (The_Message : in out ...);
-- Clears the value of the field. After calling this, Has_Foo will
-- return False and Get_Foo will return the default value.

function Get_Foo
  (The_Message : in out ....)
  return access Bar.Instance;
-- Returns the current value of the field. If the field is not
-- set, returns a Bar with none of its fields set.

function Release_Foo
  (The_Message : in out ...)
  return access Bar.Instance;
-- Releases the ownership of the field and returns a pointer to
-- the Bar object. After calling this, caller takes the ownership
-- of the allocated Bar object, Has_Foo will return False, and
-- Get_Foo will return the default value.

procedure Set_Foo
  (The_Message : in out ...;
   Value : in Bar.Bar_Access);
-- Sets the Bar object to the field and frees the previous field
-- value if it exists. If the Bar pointer is not null, the message
-- takes ownership of the allocated Bar object and Has_Foo will
-- return True. Otherwise, if the Bar pointer is null, the
-- behavior is the same as calling Clear_Foo.
```

## Repeated numeric fields

A repeated numeric field can be defined as in listing 3.16.

Listing 3.16: Repeated numeric field definition

```
repeated int32 foo = 1;
```

The set of procedures and functions that would be generated from the field definition in listing 3.16 is shown in listing 3.17.

Listing 3.17: Generated functions for repeated numeric fields

```
function Foo_Size
  (The_Message : in ...)
  return Protocol_Buffers.Wire_Format.PB_Object_Size;
-- Returns the number of elements currently in the field.

procedure Clear_Foo
  (The_Message : in out TestAllTypes.Instance);
-- Removes all elements from the field. After calling this,
-- Foo_Size will return zero.

function Get_Foo
  (The_Message : in ...;
   Index : in Protocol_Buffers.Wire_Format.PB_Object_Size)
  return Protocol_Buffers.Wire_Format.PB_Int32;
-- Returns the element at the given zero-based index.

procedure Set_Foo
  (The_Message : in out ...;
   Index : in Protocol_Buffers.Wire_Format.PB_Object_Size;
   Value : in Protocol_Buffers.Wire_Format.PB_Int32);
-- Sets the value of the element at the given zero-based index.

procedure Add_Foo
  (The_Message : in out ...;
   Value : in Protocol_Buffers.Wire_Format.PB_Int32);
-- Appends a new element to the field with the given value.
```

## Repeated string/bytes fields

Repeated string and bytes fields can be defined as in listing 3.18.

Listing 3.18: Repeated string/bytes field definition

```
repeated string numeric = 1;
repeated bytes  numeric = 1;
```

The set of procedures and functions that would be generated from either one of the field definitions in listing 3.18 is shown in listing 3.19.

Listing 3.19: Generated functions for repeated string/bytes fields

```
function Foo_Size
  (The_Message : in ...)
  return Protocol_Buffers.Wire_Format.PB_Object_Size;
-- Returns the number of elements currently in the field.

procedure Clear_Foo
  (The_Message : in out ...);
-- Removes all elements from the field. After calling this,
-- Foo_Size will return zero.

function Get_Foo
  (The_Message : in ...;
   Index : in Protocol_Buffers.Wire_Format.PB_Object_Size)
  return Protocol_Buffers.Wire_Format.PB_String;
-- Returns the element at the given zero-based index.

procedure Set_Foo
```

```

(The_Message : in out ...;
Index : in Protocol_Buffers.Wire_Format.PB_Object_Size;
Value : in Protocol_Buffers.Wire_Format.PB_String);
-- Sets the value of the element at the given zero-based index.

procedure Add_Foo
(The_Message : in out ...;
Value : in Protocol_Buffers.Wire_Format.PB_String);
-- Appends a new element to the field with the given value.

```

## Repeated enum fields

Using the enum type defined in listing 3.10 a repeated enum field can be defined as in listing 3.20.

Listing 3.20: Repeated enum field definition

```
repeated Bar foo = 1;
```

The set of procedures and functions that would be generated from the field definition in listing 3.20 is shown in listing 3.21.

Listing 3.21: Generated functions for repeated enum fields

```

function Foo_Size
(The_Message : in ...)
return Protocol_Buffers.Wire_Format.PB_Object_Size;
-- Returns the number of elements currently in the field.

procedure Clear_Foo
(The_Message : in out ...);
-- Removes all elements from the field. After calling this,
-- Foo_Size will return zero.

function Get_Foo
(The_Message : in ...;
Index : in Protocol_Buffers.Wire_Format.PB_Object_Size)
return Bar;
-- Returns the element at the given zero-based index.

procedure Set_Foo
(The_Message : in out ...;
Index : in Protocol_Buffers.Wire_Format.PB_Object_Size;
Value : in Bar);
-- Sets the value of the element at the given zero-based index.

procedure Add_Foo
(The_Message : in out ...;
Value : in Bar);
-- Appends a new element to the field with the given value.

```

## Repeated embedded message fields

Using the message type defined in listing 3.13 a repeated embedded message field can be defined as in listing 3.22.

Listing 3.22: Repeated embedded message field definition

```
repeated Bar foo = 1;
```

The set of procedures and functions that would be generated from the field definition in listing 3.22 is shown in listing 3.23.

Listing 3.23: Generated functions for repeated embedded message fields

```
function Foo_Size
  (The_Message : in ...)
  return Protocol_Buffers.Wire_Format.PB_Object_Size;
-- Returns the number of elements currently in the field.

procedure Clear_Foo
  (The_Message : in out ...);
-- Removes all elements from the field. After calling this,
-- Foo_Size will return zero.

function Get_Foo
  (The_Message : in ...;
   Index : in Protocol_Buffers.Wire_Format.PB_Object_Size)
  return access Bar.Instance;
-- Returns the element at the given zero-based index.

function Add_Foo
  (The_Message : in out ...)
  return access Bar.Instance;
-- Appends a new element to the field with the given value.
```

## 3.10 Installation and use

There are two options available when it comes to installing the Ada code generation plug-in for Protocol Buffers.

The first option is to build and install it along with the official compiler from Google. Installation in this case is as simple as following the installation instructions for `protoc`, with the exception that installation using Microsoft Visual Studio is not supported.

The second option is to install it separately by compiling and linking the sources with `libprotobuf` and `libprotoc` that are built when installing the official compiler. A tool called `pkg-config` can be used to determine which flags needs to be passed to the compiler and linker on the target platform. More information about compilation using `pkg-config` can be found in `README.txt`.

### 3.10.1 From .proto file to executable application

This section describes the process of going from a `.proto` file to an executable application by demonstrating how to build an example application from files located in the directory `examples/`. The address book example described here is the same as the one used by Google to illustrate the use of Protocol Buffers with officially supported programming languages. It uses the Protocol Buffers format to store contact information in a file, i.e. store information about contacts like an address book. Assumptions made in this tutorial are that `protoc` has been installed with code generation support for Ada and that a version of GNAT has been installed with support for Ada 2012.

Listing 3.24 shows `addressbook.proto`, which is very similar to the example `.proto` file described in section 2.2.4. The only difference of importance here is that a message `AddressBook` is used to store information about persons.

Listing 3.24: `addressbook.proto`

```
// See README.txt for information and build instructions.

package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2; // Unique ID number for this person.
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person person = 1;
}
```

Having described the data structure we now need to generate Ada code from `addressbook.proto`. If `protoc` was correctly installed using the first method, described above, it can be invoked directly from the command line, provided that it is located in a directory specified by the `PATH` variable.

```
> protoc --ada_out=. addressbook.proto
```

This will generate Ada source code files in the current directory that can be used by the example application to serialize and deserialize contact information. Source code for the example application is listed in appendix B.

To build the example executable(s) we now only need to compile and link the application with the generated source code and the library used by generated source code.

```
> gnatmake -I../ada/src/main add_person.adb && gnatmake
-I../ada/src/main list_people.adb
```

If the above command was successful `list_people` and `add_person` should now appear in the current working directory. A demonstration run is displayed in listing 3.25. It should also be possible to list and add people using the



address book applications developed in other languages that are present in the `examples/` directory.

Listing 3.25: Address book demonstration

```
> ./add_person address_book
address_book: File not found. Creating a new file.
Enter person ID number: 1
Enter name: John Doe
Enter email address (blank for none): john.doe@nowhere.com
Enter a phone number (or leave blank to finish):

> ./list_people address_book
Person ID: 1
  Name: John Doe
  E-mail address: john.doe@nowhere.com
```



# Chapter 4

## Evaluation

In this chapter the implementation of Ada code generation for Protocol Buffers is evaluated and a performance comparison is made with GNAT-Coll.JSON.

### 4.1 Requirements

The requirements detailed in appendix A provided a good starting point for implementing the code generation plug-in. However, the scope proved to be great for a master's thesis, and as a result a decision was made to scale back the feature set. Requirements had already been prioritized for situations such as this and it was decided that focus should be placed on implementing priority 1 requirements. A notable exception is extensions, which, while being a priority 1 requirement<sup>1</sup>, was deemed too time consuming to implement. Google (2013a) writes the following about their C++ implementation ‘... The exact implementation of extension identifiers is complicated and involves magical use of templates ...’.

The written unit test cases provide confirmation that all priority 1 requirements, with the exception of support for extensions, have been fulfilled.

### 4.2 Performance comparison

A benchmark comparison was made to evaluate the potential performance advantage realised by substituting JSON with Protocol Buffers as a serialization framework for the radar system. Unfortunately for several different reasons, unrelated to technology, it was not possible to replace the existing serialization framework in the radar system. Instead, an artificial serialization benchmark was constructed with the sole focus of isolating serialization performance. Without having access to a running system and only limited

---

<sup>1</sup>See requirement 51 appendix A.

access to radar system data, it should be noted that the benchmark relies on randomly generated data. Data is therefore not necessarily representative of real-world usage where, for instance object size might fluctuate less. Any performance advantage shown by the benchmark comparison would therefore not necessarily translate into a real world performance gain for the radar system, because of a multitude of other different factors affecting performance in a distributed system.

### 4.2.1 Benchmark environment

Performance benchmarks were conducted on a MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of DDR3 memory clocked at 1067 MHz. The computer was running Mac OS X Lion 10.7.5 in 64-bit mode and all binaries were built using gnatmake<sup>2</sup>. Version 1.5w of GNATColl.JSON was used for all JSON related benchmarks. A newer version (1.6w) is available but performance was found to be comparable to that of version 1.5w.

### 4.2.2 Measurement techniques and results

To mimic the radar system, random data of the same format as that used by the radar system was generated. A description of said data can be seen in appendix D along with a `.proto` file constructed from the description.<sup>3</sup> In addition to that, appendix D also contains the Ada source code for the performance comparison.

The benchmark presents result for the following three stages:

#### Object creation

Reads generated random data from a file and into an array of records containing every field described by the JSON format. The array of records is then used to set every field inside a newly created object to the corresponding record value.

#### Serialization

Serializes previously created objects to a so called `Memory_Stream`, which is used to stream objects to a buffer in memory. A message with a repeated embedded message field is used to store each object, which makes it easier to deserialize objects later on, since Protocol Buffers does not use any message delimiters. Every JSON object's length is stored in an array to make parsing easy. An alternative would have been to store each JSON object inside an array, but this is a much costlier operation.

---

<sup>2</sup>Gnatmake GPL 2012 (20120509) was compiled from source on the target machine and invoked with the following flags `-O2 -funroll-loops -gnatn`.

<sup>3</sup>All field names have been obfuscated to hide implementation details. Field name length affects the size of serialized JSON objects, but the obfuscated names are of comparable length to the actual field names and have no considerable effect on serialized object size.

### Deserialization

Deserializes previously serialized objects.

Figure 4.1 shows the results from the performance comparison. Combined time refers to the total time it takes to create, serialize and deserialize an object. All time measurements were taken with the Clock function declared inside the standard Calendar package. Individual tests were run 500 times and all time measurements are based on an average of the measured time for each individual test run. Also shown in the figure is the total serialized size of all objects.

### 4.2.3 Analysis of results

The time taken to serialize and deserialize objects scale roughly linearly with the number of objects for both GNATColl.JSON and Protocol Buffers. Although not shown in the graphs it continues to do so for well above 100,000 objects. Results show that Protocol Buffers is about 6 to 8 times faster in the combined serialization/deserialization performance comparison, with the difference increasing as the number of objects grow. Looking at object creation, serialization and deserialization independently paints a similar picture with Protocol Buffers approximately 5 to 9 times faster. This outcome was largely expected because of the differences between the two formats. JSON not being optimized for performance, but rather for readability and ease of use. Protocol Buffers performance advantage can be attributed to a more compact representation, an almost native binary representation of data types and use of length-prefixing for embedded message fields and strings.

Serialized objects see a reduction in size by around 45% when going from JSON to Protocol Buffers, but this is of course highly dependent on the object being serialized. The JSON format adopted by the radar system uses string values for a majority of its fields. The reliance on string fields reduces Protocol Buffers' size advantage, since serialized string values are encoded no differently than they are using JSON. Some of these string fields could quite possibly be represented using another more space efficient field type by Protocol Buffers, because of its richer type system. Default values could also be used to further reduce the size of serialized objects. Size is an important aspect when dealing with large data quantities, especially if that information must be transferred over a network. If size is more important than serialization speed there is always the possibility of compressing data, which should produce serialized objects closer in size to Protocol Buffers because of the information redundancy introduced by JSON. A quick check shows that compressing 1,000 radar system objects using gzip results in a compression ratio of 2.9 for JSON and 1.8 for Protocol Buffers.<sup>4</sup> Another factor to consider is the impact a format has on executable file size. Protocol

---

<sup>4</sup>gzip is a file compression utility implementing the DEFLATE algorithm described by Deutsch (1996). File size before compression for JSON was 546,899 bytes and 188,295 bytes after, and for Protocol Buffers 296,644 bytes before and 165,224 bytes after.

Buffers generate code for every message defined inside a `.proto` file, which of course affects the executable file size. Take for example the benchmark application which produces a binary of 1.5 MiB (1,589 KiB) for Protocol Buffers and 876 KiB for GNATColl.JSON. However, the size of the executable is not all due to generated code. Most of it is due to the library used by the generated code. This is evident if the library is dynamically linked as the size then shrinks to 280 KiB.

Performance is an important aspect, albeit not the only one to consider when contrasting the two serialization formats. For instance, Protocol Buffers require the use of `.proto` files, which must be compiled to generate code for object creation, serialization and deserialization, adding another step to the build process. However, it relieves the programmer of the need to write handcrafted serialization and deserialization functions for objects, which can be an error prone activity when working with complex objects and data structures. JSON's text based format on the other hand means that it is easier to interpret serialized objects.

## 4.3 Future investigations and improvements

This section contains suggestions on improvements to the code generation plug-in.

### 4.3.1 Performance

Although performance is crucial to a serialization framework, it has not been prioritized when developing code generation for Ada. Instead, work has been primarily focused on implementing as many of Protocol Buffers' features as possible. As a consequence, the code generation plug-in could benefit from the following enhancements:

- Thread safety has not been considered, but is something that would obviously be important for multi-threaded applications.
- Inlining of procedures and functions. This would require some investigation into what should be inlined and how it would affect code size.
- Deciding which field to parse when reading from a `Coded_Input_Stream` is done with the help of a case statement<sup>5</sup>. Serialized message fields are prefixed with a tag, which is what is used by the case statement to determine how binary data that follows the tag should be parsed.

Code generated for C++ by `protoc` uses a similar approach but tries to optimize the process by guessing the next field. If the current field is a repeated field (not packed) the next field is probably another instance of this field, otherwise the next field is probably the field

---

<sup>5</sup>A case statement is the Ada equivalent to a switch statement in C/C++.

with a tag value increased by one from the current field's tag value. This is accomplished in generated code through the use of if and goto statements.

- A message declared inside a `.proto` file generates a record type with fields belonging to the message stored as record components. Alignment of these record components could possibly be optimized to minimize padding in memory without having a considerable negative affect on spatial cache locality. Ada has support for so called record representation clauses that can be used to specify the order, position and size of components. It should therefore be a relatively simple matter to rearrange fields using a record representation clause. A similar approach is taken by `protoc` when generating code for C++.

#### 4.3.2 Features

Code generation for Ada still has some ways to go before achieving feature parity with code generation for the officially supported languages as is evident from reading earlier sections. Nevertheless, it already includes support for most of the core features present in the protocol buffer language such as default values, optional fields, required fields, repeated fields, nested messages, packed repeated fields, etc.

The system requirements listed in appendix A gives the reader some appreciation of avenues worthy of exploration, when it comes to adding additional features. Here is a list of some of the main features and enhancements code generation would benefit from supporting:

- Extensions, as described earlier, is probably the most important feature not yet implemented, but potentially also the most time consuming to implement.
- So called unknown fields that would allow applications created with an earlier version of a `.proto` file to talk to applications created with a newer version of the same `.proto` file, as long as only optional or repeated fields were added.
- A reflection interface for message types that would allow runtime manipulation of messages.
- The API could benefit from becoming a bit more user friendly. Anonymous access types are for instance mixed with general access types, due to circular dependencies that can arise among packages. Fields and messages declared inside a `.proto` file can have names that would lead to illegal identifiers in generated Ada code.<sup>6</sup> Bloch (2005) provides some tips on how to design a good API and why the design of an API

---

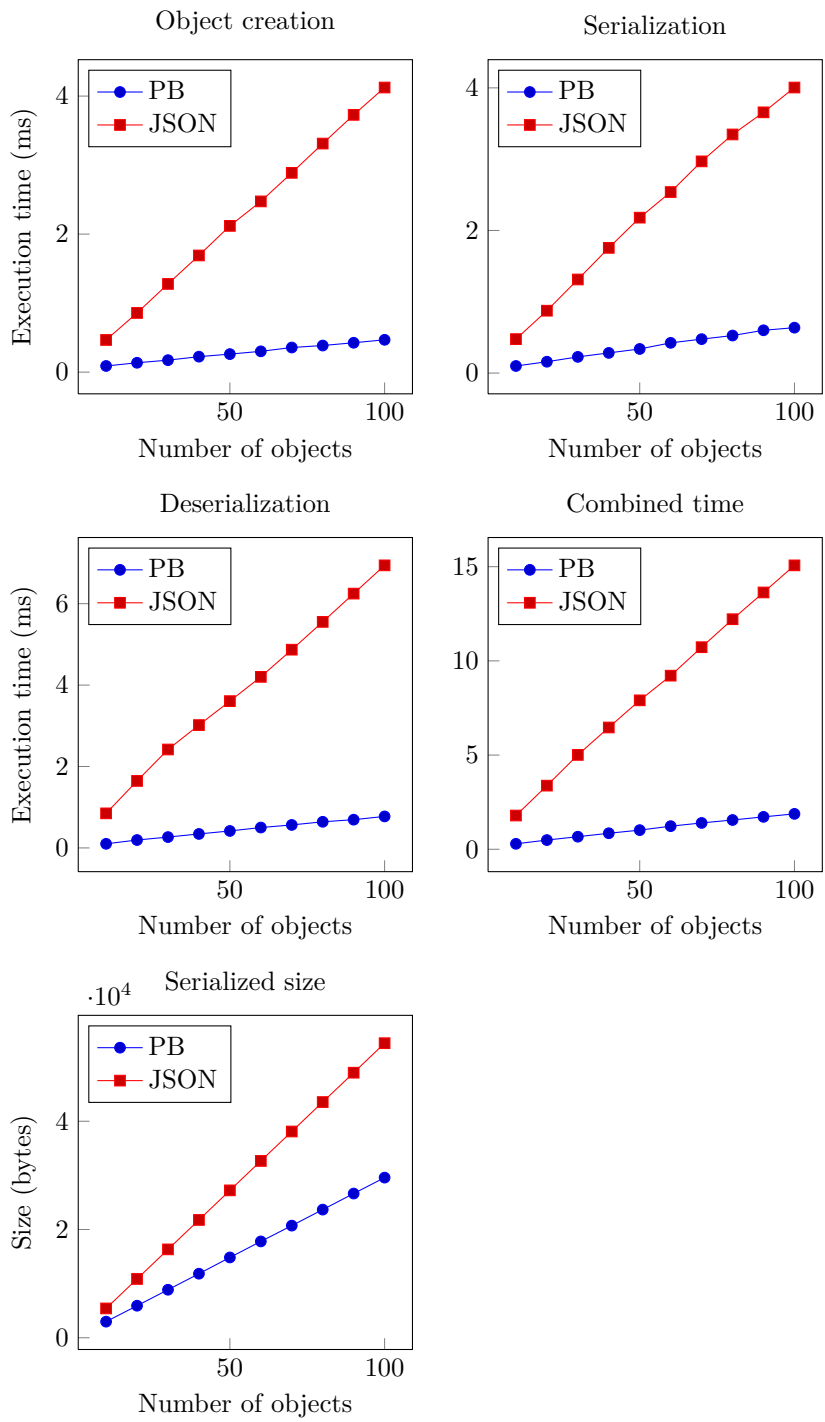
<sup>6</sup>A message named `Foo_` would for instance generate a package named `Foo_`, but an identifier in Ada is not allowed to end with underline.

is so important, which could come in handy when considering making changes to the API.

- Support for embedded messages allocated on the stack. At present the message API is designed for embedded messages allocated on the heap with automatic finalization.



Figure 4.1: Performance comparison.





# Chapter 5

## Conclusions

The history of data serialization frameworks can be traced back to the 1980s with the introduction of ASN.1. Since then various other frameworks and formats have appeared that aim to facilitate cross-platform language independent information exchange. Protocol Buffers is one such framework that was developed in-house by Google and later released to the public under an open-source license. As part of this thesis Ada code generation support for Protocol Buffers have been developed to extend the number of supported languages.

Code generation for Ada uses protoc's plug-in system, which makes it possible to reuse the official compiler for parsing of proto definition files. The developed plug-in was written in C++, but code generated from IDL files also rely on a library of Ada code.

The purpose of this thesis has been to develop and evaluate Ada code generation for Protocol Buffers. To evaluate the developed solution an existing radar system was chosen as a candidate for replacement of serialization format. As part of the evaluation, this thesis aims to answer the following questions for the radar system:

- **Can Protocol Buffers be used to represent the same information as JSON?**

Using the developed solution it has been shown that the radar system, which is described in the introductory chapter, can be rewritten to accommodate Protocol Buffers instead of JSON as a serialization format. Protocol Buffers offer support for language constructs that not only matches those provided by JSON, such as object and array, but it also provides a much richer type system than JSON. It should therefore come as no surprise that Protocol Buffers can be used to represent the same information as JSON.

- **What performance advantage, if any, does the developed Protocol Buffers solution provide over JSON? Is performance at**

---

## least as good as with JSON?

An artificial benchmark, based on the radar system's data format, indicates that the developed solution could offer substantial performance improvements in comparison to JSON. Protocol Buffers offer an advantage both in terms of the size of serialized objects and the time taken to deserialize and serialize objects when compared to GNATColl.JSON. Results from the benchmark reveal that Protocol Buffers is about 6 to 8 times faster in a combined serialization/deserialization performance comparison. In addition, the change of serialization format has the added benefit of reducing size of serialized objects by approximately 45%. A majority of radar system data is encoded as text, which diminishes Protocol Buffers advantage over JSON when it comes to the size of serialized objects. Protocol Buffers performance advantage can be attributed to a more compact representation, an almost native binary representation of data types and use of length-prefixing for embedded message fields and strings.

There are some caveats to be aware of when interpreting the results from the benchmark. Without having access to a running system and only limited access to radar system data, it should be noted that the benchmark relies on randomly generated data. Also, factors related to the distributed nature of the system might affect performance and the results from the benchmark should, as a consequence of this, only be seen as indicative of possible performance gains.

Code generation for Ada does not achieve feature parity with languages that are officially supported by Google, but the main functionality needed to handle serialization with Protocol Buffers has been developed. The developed solution provides Saab SDS with the tools necessary to handle Protocol Buffers on the Ada side in other projects as an added bonus. Support for extensions might have to be implemented as it is a fairly common language construct, but other than that there is no major obstacle hindering the adoption of Protocol Buffers for use with Ada.

# Appendices



# Appendix A

## System requirements

Functional and non-functional system requirements elicited as part of the development of Ada code generation support for Protocol Buffers are listed in the following sections.

### A.1 Definitions and conventions

This is list of definitions and conventions used throughout this document.

- An unknown field is a field encountered during the parsing of a message not declared inside the `.proto` file that was used for compilation.
- Scalar value types refer to the protocol buffer field types defined in Protocol Buffers language guide.<sup>1</sup> The scalar value types are double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64, bool, string and bytes.
- Numeric fields are fields of type double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64 or bool.
- All messages parsed from or written to a stream shall be encoded according to the binary wire format for protocol buffer messages, unless otherwise specified.
- optional, required, repeated, enum and message are keywords defined in the protocol buffer language guide.

### A.2 Non-functional requirements

Table A.1 lists the systems non-functional requirements.

---

<sup>1</sup><https://developers.google.com/protocol-buffers/docs/encoding>

Table A.1: Non-functional requirements

ID	Requirement description	Priority
1	All requirements shall be uniquely identifiable using an identification number.	1
2	All requirements shall be assigned a priority from 1 to 3, where 1 is the highest priority and 3 is the lowest.	1
3	The application shall work on the following platforms Mac OS X 10.8.X, Windows XP SPX and CentOS 6.X running on an x86-32 or x86-64 processor architecture.	1
4	Generated Ada code shall work with an Ada 2012 certified compiler.	1
5	All code produced shall be in a source code repository that is managed by version control software.	1
6	All functions and procedures shall be unit tested.	1
7	The application shall be written as a plug-in to protoc, Google's official Protocol Buffers compiler.	1

### A.3 Functional requirements

Table A.2 lists the systems functional requirements.

Table A.2: Functional requirements

ID	Requirement description	Priority
8	The application shall generate Ada source code files meeting all listed requirements when the following command is run from the command line.  <pre>&gt; protoc --ada_out=dir file.proto</pre> <p>Where <code>dir</code> is a relative path and <code>file.proto</code> is a valid <code>.proto</code> definition file</p>	1
9	A message declared inside <code>file.proto</code> shall generate a package specification when <code>protoc --ada_out=dir file.proto</code> is run from the command line.	1
10	A message declared inside <code>file.proto</code> shall generate a package body when <code>protoc --ada_out=dir file.proto</code> is run from the command line.	1
11	The package specification generated for a message shall include a tagged type to support dot notation.	1

*Continued on next page*



Table A.2 – *Continued from previous page*

ID	Requirement description	Priority
12	A declaration option <code>package=ancestor.parent</code> inside a proto file shall generate an empty package named <code>Ancestor</code> inside the file <code>ancestor.ads</code> and an empty child package named <code>Ancestor.Parent</code> inside the file <code>ancestor-parent.ads</code> etc. A generated message shall then result in a package specification named <code>Ancestor.Parent.Message</code> inside the file <code>ancestor-parent-message.ads</code> and a package body <code>Ancestor.Parent.Message</code> inside the file <code>ancestor-parent-message.adb</code> .	1
13	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to copy the message.	1
14	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to clear the contents of the message, any optional field values are reset to their default, if one was specified when the field was declared. Any fields previously set inside the message will no longer be considered as set.	1
15	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to determine if all required fields inside the message has been set.	1
16	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to merge a message with a message read from an input stream. If required fields are missing from the message read from the input stream an error shall be indicated.	1
81	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to merge a message with a message read from an input stream, without requiring all required fields to be set in the message read from the input stream.	1
17	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to get the message's full name.	1
18	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to write the message to an output stream if all required fields have been set, otherwise an error shall be indicated.	1
19	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to write the message to an output stream without requiring all required fields to be set.	1

---

*Continued on next page*

Table A.2 – Continued from previous page

ID	Requirement description	Priority
20	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to parse a message from an input stream if all required fields have been set, otherwise an error shall be indicated.	1
21	For a message declared inside a <code>.proto</code> file the generated Ada code shall provide functionality to parse a message from an input stream without requiring all required fields to be set.	1
22	For a singular field of a scalar value type, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to determine if the field has been set.	1
23	For a singular field of a scalar value type, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to get the field's value. If the field is optional and has not been set, the default value is returned.	1
24	For a singular field of a scalar value type, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to set the field's value.	1
25	For a singular field of a scalar value type, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to clear the field. If the field was declared as optional then the field's value is reset to its default value, if one was specified when the field was declared. A previously set field will no longer be considered as set.	1
26	For a singular enum field, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to determine if the field has been set.	1
27	For a singular enum field, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to get the field's value. If the field was declared as optional and has not been set, the default value is returned.	1

*Continued on next page*

Table A.2 – *Continued from previous page*

ID	Requirement description	Priority
28	For a singular enum field, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to set the field's value. If the specified value is not a valid enum value, as defined inside the <code>.proto</code> file, an error shall be indicated.	1
29	For a singular enum field, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to clear the field. Resets the field's value to its default value, if one was specified when the field was declared. A previously set field will no longer be considered as set.	1
30	For a repeated field of a scalar value type, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to get the current number of elements in the field.	1
31	For a repeated field of a scalar value type, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to get an element at a specified index in the field.	1
32	For a repeated field of a scalar value type, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to set the value of an element at a specified index in the field.	1
33	For a repeated field of a scalar value type, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to add an element with a specified value to the field.	1
34	For a repeated field of a scalar value type, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to remove all elements in the field.	1
35	For a repeated enum field, declared inside a message, the generated Ada code shall provide functionality to get the current number of elements in the field.	1
36	For a repeated enum field, declared inside a message, the generated Ada code shall provide functionality to get an element at a specified index in the field.	1

*Continued on next page*

Table A.2 – Continued from previous page

ID	Requirement description	Priority
37	For a repeated enum field, declared inside a message, the generated Ada code shall provide functionality to set the value of an element at a specified index in the field. If the specified value is not a valid enum value, as defined inside the <code>.proto</code> file, an error shall be indicated.	1
38	For a repeated enum field, declared inside a message, the generated Ada code shall provide functionality to add an element with a specified value to the field. If the specified value is not a valid enum value, as defined inside the <code>.proto</code> file, an error shall be indicated.	1
39	For a repeated enum field, declared inside a message, the generated Ada code shall provide functionality to remove all elements in the field.	1
40	For a singular embedded message field, declared inside a message as either optional or required in a <code>.proto</code> file, the generated Ada code shall provide functionality to determine if the field has been set.	1
41	For a singular embedded message field, declared inside a message as either optional or required in <code>.proto</code> file, the generated Ada code shall provide functionality to get the field's value. If the field has not been set, a message with none of its fields set, is returned.	1
42	For a singular embedded message field, declared inside a message as either optional or required in <code>.proto</code> file, the generated Ada code shall provide functionality to set the field's value.	1
43	For a singular embedded message field, declared inside a message as either optional or required in <code>.proto</code> file, the generated Ada code shall provide functionality to clear the field.	1
44	For a repeated embedded message field, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to get the current number of elements in the field.	1
45	For a repeated embedded message field, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to get an element at a specified index in the field.	1
46	For a repeated embedded message field, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to set the value of an element at a specified index in the field.	1

*Continued on next page*

Table A.2 – Continued from previous page

ID	Requirement description	Priority
47	For a repeated embedded message field, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to add an element with a specified value to the field.	1
48	For a repeated embedded message field, declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality to remove all elements from the field.	1
49	Trying to set the value of a repeated field with an index that refers to no element shall be indicated with an error.	1
50	Trying to get the value of a repeated field with an index that refers to no element shall be indicated with an error.	1
51	For an extension range declared inside a message in a <code>.proto</code> file, the generated Ada code shall provide functionality corresponding to a regular field of that type.	1
52	If an optional field is declared with a default value inside a message in a <code>.proto</code> file, as described in the protocol buffers language guide, the generated Ada code shall set that field to the default value.	1
53	An optional string field declared inside a message shall have the empty string as a start value, if no default value has been specified.	1
54	An optional bool field declared inside a message shall have False as a start value, if no default value has been specified.	1
55	An optional numeric field (except bool) declared inside a message shall have zero as start value, if no default value has been specified.	1
56	An optional enum field declared inside a message shall have the first value listed in the enum definition as a start value, if no default value has been specified.	1
57	The application shall support reading and writing of fields declared with option packed set to true inside a <code>.proto</code> file.	1
59	The application shall support import statements inside a <code>.proto</code> file.	1
60	The application shall support multiple messages declared inside a single <code>.proto</code> file.	1

---

*Continued on next page*

Table A.2 – Continued from previous page

ID	Requirement description	Priority
61	The application shall ignore any comments inside a <code>.proto</code> file. A comment is denoted by <code>//</code> and continues to the end of the line.	1
62	The application shall support nested extensions i.e. extensions declared inside another message.	2
63	The application shall impose a default limit on the size of a message.	2
64	The application shall allow the user to specify a new limit on the size of a message.	2
65	The application shall indicate an error when trying to write a message with a size larger than the message size limit.	2
66	The application shall indicate an error when trying to parse a message with a size larger than the message size limit.	2
67	The application shall output any error encountered during the parsing of a <code>.proto</code> file to standard error with a descriptive message. If it is a fatal error, the application shall exit and a non-zero exit code shall be returned.	2
68	The application shall generate code formatted according to the following standard (add standard here).	2
69	The application shall provide insertion points for context clause directives outside a message's package specification to allow other plug-ins to extend the generated Ada code.	3
70	The application shall provide insertion points inside a message's package specification to allow other plug-ins to extend the generated Ada code.	3
71	The application shall provide insertion points inside a message's package body to allow other plug-ins to extend the generated Ada code.	3
72	Functions or procedures used to parse a message shall store all unknown fields encountered during the parsing of a message.	3
73	Functions or procedures used to write a message shall write all unknown fields encountered during the parsing of a message.	3
74	The application shall include functionality to remove all unknown fields encountered during parsing of a message.	3
75	The application shall include functionality to determine if a parsed message contains any unknown fields.	3

*Continued on next page*

Table A.2 – *Continued from previous page*

ID	Requirement description	Priority
76	The application shall include functionality to get current number of elements in an unknown repeated field.	3
77	The application shall include functionality to get an element at a specified index in the unknown repeated field. If no element exists at the specified index, an error shall be indicated.	3
78	The application shall allow the user to generate Ada code optimized for minimum code size.	3
79	The application shall allow the user to generate Ada code optimized for speed.	3
80	The application shall provide a text format, which can be used for debugging and editing purposes. Formatting rules for the various language constructs are described below:	3

- Message
    - A message starts with the message name followed by an opening curly brace.
    - The message's fields follow the message's opening curly brace.
    - The message ends with a closing curly brace.
  - Field
    - If the field is an embedded message, see the rules for messages, otherwise a colon follows the field's name.
    - The colon is then followed by the field's value, if it is a string the field's value is enclosed in double quotes.
-





# Appendix B

## Address book example

Listed in this appendix is the source code for the address book example mentioned in section 3.10.1.

### B.1 Adding contacts to the addressbook

Listing B.1 displays `add_person.adb`, which is used to add contacts to an address book, i.e a file, when invoked from the command line. It reads or creates an address book specified as an argument to the program and then prompts the user to enter information for a new contact, which is then stored in the specified file.

Listing B.1: `add_person.adb`

```
-- See README.txt for information and build instructions.
pragma Ada_2012;

with AddressBookProto.AddressBook;
with AddressBookProto.Person.PhoneNumber;

with Ada.Command_Line;
with Ada.Streams.Stream_IO;
with Ada.IO_Exceptions;
with Ada.Long_Integer_Text_IO;
with Ada.Text_IO;

procedure Add_Person is
  package CLI renames Ada.Command_Line;
  package TIO renames Ada.Text_IO;
  package IIO renames Ada.Long_Integer_Text_IO;
  package SIO renames Ada.Streams.Stream_IO;

  procedure Prompt_For_Address
    (Person : AddressBookProto.Person.Person_Access) is
  begin
    TIO.Put ("Enter person ID number: ");
    declare
      Id : Integer;
    begin
      IIO.GET (Id);
      Person.Set_Id (Id);
    end;
  end;
```

```
end;

TIO.Skip_Line;

TIO.Put ("Enter name: ");
Person.Set_Name (TIO.Get_Line);

TIO.Put ("Enter email address (blank for none): ");
declare
    Email : constant String := TIO.Get_Line;
begin
    if Email'Length > 0 then
        Person.Set_Email (Email);
    end if;
end;

loop
    TIO.Put ("Enter a phone number (or leave blank to finish): ");
    declare
        Number : constant String := TIO.Get_Line;
        Phone_Number : AddressBookProto.Person.PhoneNumber :=
            PhoneNumber_Access;
    begin
        if Number'Length = 0 then
            exit;
        end if;

        Phone_Number := Person.Add_Phone;
        Phone_Number.Set_Number (Number);

        TIO.Put ("Is this a mobile, home, or work phone? ");
        declare
            Phone_Type : constant String := TIO.Get_Line;
        begin
            if Phone_Type = "mobile" then
                Phone_Number.Set_Type_Pb (Addressbookproto.Person.MOBILE);
            elsif Phone_Type = "home" then
                Phone_Number.Set_Type_Pb (Addressbookproto.Person.HOME);
            elsif Phone_Type = "work" then
                Phone_Number.Set_Type_Pb (Addressbookproto.Person.WORK);
            else
                TIO.Put_Line ("Unknown phone type. Using default.");
            end if;
        end;
    end;
end loop;
end Prompt_For_Address;

Address_Book : AddressBookProto.AddressBook.Instance;
begin
    if CLI.Argument_Count /= 1 then
        TIO.Put_Line (File => TIO.Standard_Error,
            Item => "Usage: " & CLI.Command_Name & "
                ADDRESS_BOOK_FILE");
        CLI.Set_Exit_Status (Code => CLI.Failure);
        return;
    end if;

    -- Try to read the existing address book.
    declare
        Input : SIO.Stream_Access;
        File : SIO.File_Type;
    begin
        SIO.Open (File => File,
            Mode => SIO.In_File,
            Name => CLI.Argument (1));
        Input := SIO.Stream (File => File);
        Address_Book.Merge_From_Input_Stream (Input_Stream => Input);
```

```
SIO.Close (File => File);
exception
when Ada.IO_Exceptions.Name_Error =>
    TIO.Put_Line (CLI.Argument (1) & ": File not found. Creating a new
        file.");
end;

-- Add an address.
Prompt_For_Address (Address_Book.Add_Person);

-- Write the new address book back to disk.
declare
    Output : SIO.Stream_Access;
    File   : SIO.File_Type;
begin
    SIO.Create (File => File,
                Mode => SIO.Out_File,
                Name => CLI.Argument (1));
    Output := SIO.Stream (File);

    Address_Book.Serialize_To_Output_Stream (Output);
end;

end Add_Person;
```

## B.2 Listing contacts from the address book

Listing B.2 displays `list_people.adb`, which is used to list the contacts in an address book. It reads data stored in an address book and prints the contact information stored inside to `stdout`.

Listing B.2: `list_people.adb`

```
-- See README.txt for information and build instructions.
pragma Ada_2012;

with AddressBookProto.AddressBook;
with AddressBookProto.Person.PhoneNumber;

with Ada.Command_Line;
with Ada.Streams.Stream_IO;
with Ada.IO_Exceptions;
with Ada.Long_Integer_Text_IO;
with Ada.Text_IO;

-- Reads the entire address book from file and prints all the information
-- inside.
procedure List_People is
    package CLI renames Ada.Command_Line;
    package TIO renames Ada.Text_IO;
    package SIO renames Ada.Streams.Stream_IO;

    -- Iterates through all people in Address_Book and prints info about
    -- them.
    procedure Print
        (Address_Book : in AddressBookProto.AddressBook.Instance)
    is
    begin
        for K in 0 .. Address_Book.Person_Size - 1 loop
            declare
                Person : AddressBookProto.Person.Person_Access := Address_Book.
                    Get_Person (K);
            begin

```

```

TI0.Put_Line ("Person ID:" & Integer'Image (Person.Get_Id));
TI0.Put_Line ("  Name: " & Person.Get_Name);
if Person.Has_Email then
  TI0.Put_Line ("  E-mail address: " & Person.Get_Email);
end if;

for M in 0 .. Person.Phone_Size - 1 loop
  declare
    Phone_Number : AddressBookProto.Person.PhoneNumber;
    PhoneNumber_Access := Person.Get_Phone (M);
  begin
    case Phone_Number.Get_Type_Pb is
      when AddressBookProto.Person.MOBILE =>
        TI0.Put ("  Mobile phone #: ");
      when AddressBookProto.Person.HOME =>
        TI0.Put ("  Home phone #: ");
      when AddressBookProto.Person.WORK =>
        TI0.Put ("  Work phone #: ");
    end case;

    TI0.Put_Line (Phone_Number.Get_Number);
  end;
end loop;

end;
end loop;
end Print;

begin
  if CLI.Argument_Count /= 1 then
    TI0.Put_Line (File => TI0.Standard_Error,
      Item => "Usage: " & CLI.Command_Name & "
        ADDRESS_BOOK_FILE");
    CLI.Set_Exit_Status (Code => CLI.Failure);
    return;
  end if;

  declare
    Input : SI0.Stream_Access;
    File : SI0.File_Type;
    Address_Book : AddressBookProto.AddressBook.Instance;
  begin
    SI0.Open (File => File,
      Mode => SI0.In_File,
      Name => CLI.Argument (1));
    Input := SI0.Stream (File => File);

    -- Read existing address book.
    Address_Book.Parse_From_Input_Stream (Input);

    Print (Address_Book);
  end;
end List_People;

```

# Appendix C

## Unit tests

Unit test cases have been partitioned into three test suites, which are described in the following sections.

### C.1 Coded\_Output\_Stream test suite

Unit test cases listed in table C.1 verify the behaviour of package Coded\_Output\_Stream.

Table C.1: Coded\_Output\_Stream test cases

Test
<b>Test_Encode_Zig_Zag_32</b> Encodes 32-bit integers using the ZigZag encoding and compares the result to known values. Checks that decoding and then encoding a ZigZag encoded number does not affect the ZigZag encoded number.
<b>Test_Encode_Zig_Zag_64</b> Same as above but for 64-bit integers.
<b>Test_Write_Raw_Little_Endian_32</b> Writes 32-bit little-endian integers to stream and compares them to the expected result.
<b>Test_Write_Raw_Little_Endian_64</b> Same as above but for 64-bit little-endian integers.
<b>Test_Write_Raw_Varint_32</b> Writes 32-bit varint numbers to stream and compares them to the expected result.
<b>Test_Write_Raw_Varint_64</b> Same as above but for 64-bit varint numbers.

## C.2 Coded\_Input\_Stream test suite

Table C.2 lists unit test cases used to verify the behaviour of package Coded\_Input\_Stream.

Table C.2: Coded\_Input\_Stream test cases

Test
<b>Test_Decode_Zig_Zag_32</b> Decodes 32-bit ZigZag encoded numbers and compares them to known values.
<b>Test_Decode_Zig_Zag_64</b> Same as above but also tries 64-bit values.
<b>Test_Read_Raw_Little_Endian_32</b> Reads 32-bit little-endian integers from a stream and compares them to the expected result.
<b>Test_Read_Raw_Little_Endian_64</b> Same as above but for 64-bit little-endian integers.
<b>Test_Read_Raw_Varint_32</b> Reads 32-bit varint numbers from a stream and compares them to the expected result.
<b>Test_Read_Raw_Varint_64</b> Same as above but reads 64-bit varint numbers.
<b>Test_Invalid_Tag</b> Tests that reading an invalid tag from a stream raises an exception.
<b>Test_Size_Limit</b> Checks that reading past the allowed size limit for a message raises an exception.
<b>Test_Malicious_Recursion</b> Tests that reading a maliciously crafted recursive message from a stream raises an exception. <sup>1</sup>
<b>Test_Reset_Size_Counter</b> Checks that resetting the size limit works as expected when reading from a stream.
<b>Test_Write_And_Read_Huge_Blob</b> Writes a message containing a bytes field with 1 MB of data to file and reads it back from file. Checks that message size is reported correctly. Checks that data read matches the original bytes field.
<b>Test_Read_Maliciously_Large_Blob</b> Constructs a large malicious blob by writing fake a length that is much bigger than the maximum message length.

*Continued on next page*

<sup>1</sup>Recursion depth is increased when encountering an embedded message field inside a message and decreased when the length-delimited field has been parsed. A malicious recursion refers to a message that has a recursion depth greater than the recursion limit.

Table C.2 – *Continued from previous page*

Test
<b>Test_Write_And_Read_Whole_Message</b> Tests writing a whole message to file and then reading it back from file. Checks that message size is reported correctly. Checks that all field values are the same as they were before serialization.

## C.3 Message test suite

This test suite focuses on testing code generated for messages declared inside `.proto` files. Table C.3 lists all unit test cases in the test suite.

Table C.3: Message test cases

Test name
<b>Test_Floating_Point_Defaults</b> Tests that floating point default values specified in a <code>.proto</code> file are set correctly in generated code. Checks default values such as positive infinity, negative infinity, exponentiation and NaN.
<b>Test_Extreme_Small_Integer_Defaults</b> Tests that small integer defaults specified in a <code>.proto</code> file are set correctly in generated code.
<b>Test_String_Defaults</b> Tests string defaults specified in a <code>.proto</code> file are set correctly in generated code.
<b>Test_Required</b> Test that required fields in a message are really required. Checks this by incrementally setting all fields inside a message and checking that messages is not initialized until all fields have been set.
<b>Test_Required_Foreign</b> Tests that required fields inside optional and repeated embedded messages works as expected. <sup>2</sup>
<b>Test_Mutual_Recursion</b> Tests mutually recursive messages i.e. messages which refer to each other by containing an embedded message field of the other type.
<b>Test_Really_Large_Tag_Number</b> Checks that using really large tag numbers does not cause any issues.
<b>Test_Enumeration_Values_In_Case_Statement</b> Checks that using enumeration values in a case statement work as expected.

*Continued on next page*

---

<sup>2</sup>A message containing optional or repeated embedded message fields is considered initialized if it does not contain any embedded messages or if all message inside are initialized.

Table C.3 – *Continued from previous page*

Test
<b>Test_Accessors</b> Tests that accessors work for every field type. Checks fields declared as either optional or repeated.
<b>Test_Clear</b> Tests that clearing a message clears all field values and restores them to their default values.
<b>Test_Clear_One_Field</b> Tests clearing individual fields

---



# Appendix D

## Performance comparison

Source code and a description of the data representation for the performance comparison is presented in this appendix.

### D.1 Radar system data format

Listing D.1 provides a description of the JSON format used by the radar system. Random data generated as part of the benchmark is based on this format.

Listing D.1: Description of radar system data

```
{
  "F0": <Integer 32>,
  "F1": <String 32 characters>,
  "F2": [{ "F3": <String 34 characters> }],
  "F4":
  {
    "F5": <String 32 characters>,
    "F6": <String 34 characters>,
    "F7":
    {
      "F8":
      {
        "F9": <String 1-15 characters>,
        "F10": <String 1 character>,
      },
      "F11":
      {
        "F12": <String 5-10 characters>,
        "F13": <String 1 character>,
        "F14": <Float 32>,
        "F15": <Integer 32>,
        "F16": <Integer 32>,
        "F17": <Float 32>,
        "F18": <Integer 32>,
        "F19": <String 1-15 characters>,
        "F20": <String 1-2 characters>,
        "F21": <String 5-10 characters>,
        "F22": <Float 32>,
        "F23": <Integer 32>,
        "F24": <Integer 32>
      }
    }
  }
}
```

```

        "F25":
        {
            "F26": <String 1 character>,
            "F27": <String 1 character>,
        },
        "F28": <String 4 characters>,
        "F29": <Integer 32>,
        "F30": <Integer 32>,
    }
}
}
}

```

The .proto file shown in listing D.2 is a conversion from the JSON format used by the radar system, where objects are represented as messages and the array as a repeated message field. A message (ContainerMessage) has also been added to simplify deserialization of multiple messages from a stream.

Listing D.2: Proto definition file for the radar system (radar.proto)

```

message ObjectMessage {
    message F3Message {
        required string f3 = 1;
    }

    message F2Message {
        repeated F3Message f3_array = 1;
    }

    message F25Message {
        required string f26 = 1;
        required string f27 = 2;
    }

    message F8Message {
        required string f9 = 1;
        required string f10 = 2;
    }

    message F11Message {
        required string f12 = 1;
        required string f13 = 2;
        required float f14 = 3;
        required int32 f15 = 4;
        required int32 f16 = 5;
        required float f17 = 6;
        required int32 f18 = 7;
        required string f19 = 8;
        required string f20 = 9;
        required string f21 = 10;
        required float f22 = 11;
        required int32 f23 = 12;
        required int32 f24 = 13;
        required F25Message f25 = 14;
        required string f28 = 15;
        required int32 f29 = 16;
        required int32 f30 = 17;
    }

    message F7Message {
        required F8Message f8 = 1;
        required F11Message f11 = 2;
    }

    message F4Message {
        required string f5 = 1;
    }
}

```

```
        required string f6 = 2;
        required F7Message f7 = 3;
    }

    required int32 f0 = 3;
    required string f1 = 2;
    required F4Message f4 = 1;
    required F2Message f2 = 4;
}

message ContainerMessage {
    repeated ObjectMessage object = 1;
}
```

## D.2 GNATColl.JSON

Source code for the GNATColl.JSON benchmark application is shown in listing D.3.

Listing D.3: Benchmark application source code for GNATColl.JSON

```
pragma Ada_2012;

with Ada.Text_IO;
with Ada.Calendar;
with Generated_Data;
with Memory_Stream;

with GNATCOLL.JSON; use GNATCOLL.JSON;

procedure Main is
    Object : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F3      : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F4      : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F11     : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F25     : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F7      : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F8      : array (1 .. $NUMBER_OF_TIMES) of JSON_Value;
    F2      : array (1 .. $NUMBER_OF_TIMES) of JSON_Array;

    BUFFER_SIZE      : constant := $NUMBER_OF_TIMES * 700;
    My_Memory_Stream : Memory_Stream.Stream_Access := null;

    Length : array (1 .. $NUMBER_OF_TIMES) of Integer;

    Gen_data : Generated_Data.Array_Data_Type :=
        Generated_Data.Read_Data ($NUMBER_OF_TIMES);

    use type Ada.Calendar.Time;
begin
    -- Create
    declare
        Start_Create : Ada.Calendar.Time;
        Finish_Create : Duration;
    begin
        Start_Create := Ada.Calendar.Clock;

        for K in 1 .. $NUMBER_OF_TIMES loop
            Object(K) := Create_Object;
            F3(K) := Create_Object;
            F4(K) := Create_Object;
```

```

F11(K) := Create_Object;
F25(K) := Create_Object;
F7(K) := Create_Object;
F8(K) := Create_Object;
F2(K) := Empty_Array;

F3(K).Set_Field ("f3", Gen_data(K).F3.all);

Append (F2(K), F3(K));

F25(K).Set_Field ("f26", Gen_data(K).F26.all);
F25(K).Set_Field ("f27", Gen_data(K).F27.all);

F11(K).Set_Field ("f12", Gen_data(K).F12.all);
F11(K).Set_Field ("f13", Gen_data(K).F13.all);
F11(K).Set_Field ("f14", Gen_data(K).F14);
F11(K).Set_Field ("f15", Gen_data(K).F15);
F11(K).Set_Field ("f16", Gen_data(K).F16);
F11(K).Set_Field ("f17", Gen_data(K).F17);
F11(K).Set_Field ("f18", Gen_data(K).F18);
F11(K).Set_Field ("f19", Gen_data(K).F19.all);
F11(K).Set_Field ("f20", Gen_data(K).F20.all);
F11(K).Set_Field ("f21", Gen_data(K).F21.all);
F11(K).Set_Field ("f22", Gen_data(K).F22);
F11(K).Set_Field ("f23", Gen_data(K).F23);
F11(K).Set_Field ("f24", Gen_data(K).F24);
F11(K).Set_Field ("f25", F25(K));
F11(K).Set_Field ("f28", Gen_data(K).F28.all);
F11(K).Set_Field ("f29", Gen_data(K).F29);
F11(K).Set_Field ("f30", Gen_data(K).F30);

F8(K).Set_Field ("f9", Gen_data(K).F9.all);
F8(K).Set_Field ("f10", Gen_data(K).F10.all);

F7(K).Set_Field ("f8", F8(K));
F7(K).Set_Field ("f11", F11(K));

F4(K).Set_Field ("f5", Gen_data(K).F5.all);
F4(K).Set_Field ("f6", Gen_data(K).F6.all);
F4(K).Set_Field ("f7", F7(K));

Object(K).Set_Field ("f0", Gen_data(K).F0);
Object(K).Set_Field ("f1", Gen_data(K).F1.all);
Object(K).Set_Field ("f2", F2(K));
Object(K).Set_Field ("f4", F4(K));
end loop;

Finish_Create := Ada.Calendar.Clock - Start_Create;
Ada.Text_IO.Put (Finish_Create'Img);
Ada.Text_IO.Set_Col (15);
end;

-- Serialize
declare
  Start_Serialize : Ada.Calendar.Time;
  Finish_Serialize : Duration;
begin
  My_Memory_Stream := new Memory_Stream.Memory_Buffer_Stream (
    BUFFER_SIZE);

  Start_Serialize := Ada.Calendar.Clock;

  for K in 1 .. $NUMBER_OF_TIMES loop
    declare
      A_String : String := Object(K).Write;
    begin
      String'Write (My_Memory_Stream, A_String);
    end;
  end loop;
end;

```

```
end loop;

Finish_Serialize := Ada.Calendar.Clock - Start_Serialize;
Ada.Text_IO.Put (Finish_Serialize'Img);
Ada.Text_IO.Set_Col (29);
end;

-- Size
declare
Size : Integer := 0;
begin
for K in 1 .. $NUMBER_OF_TIMES loop
declare
A_String : String := Object(K).Write;
begin
Length(K) := A_String'Length;
Size := Size + Length(K);
end;
end loop;

Ada.Text_IO.Put (Size'Img);
Ada.Text_IO.Set_Col (43);
end;

-- Deserialize
declare
Start_Deserialize : Ada.Calendar.Time;
Finish_Deserialize : Duration;
begin
for L in 1 .. $NUMBER_OF_TIMES loop
Object(L) := JSON_Null;
end loop;

Start_Deserialize := Ada.Calendar.Clock;
for K in 1 .. $NUMBER_OF_TIMES loop
declare
A_String : String(1 .. Length(K));
begin
String'Read (My_Memory_Stream, A_String);
Object(K) := GNATCOLL.JSON.Read (A_String, "");
end;
end loop;
Finish_Deserialize := Ada.Calendar.Clock - Start_Deserialize;

Ada.Text_IO.Put_Line (Finish_Deserialize'Img);
end;

Memory_Stream.Free (My_Memory_Stream);

end Main;
```

## D.3 Protocol Buffers

Benchmark application source code for Protocol Buffers is shown in listing D.4.

Listing D.4: Benchmark application source code for Protocol Buffers

```
pragma Ada_2012;

with Radar.ObjectMessage; use Radar.ObjectMessage;
with Radar.ObjectMessage.F2Message;
with Radar.ObjectMessage.F3Message;
```

```

with Radar.ObjectMessage.F4Message;
with Radar.ObjectMessage.F8Message;
with Radar.ObjectMessage.F7Message;
with Radar.ObjectMessage.F11Message;
with Radar.ObjectMessage.F25Message;
with Radar.ContainerMessage;
with Generated_Data;
with Memory_Stream;
with Ada.Text_IO;
with Ada.Calendar;

procedure Main is

  Container : access Radar.ContainerMessage.Instance :=
    new Radar.ContainerMessage.Instance;

  Object : array (1 .. $NUMBER_OF_TIMES) of ObjectMessage_Access;
  F4      : array (1 .. $NUMBER_OF_TIMES) of F4Message.F4Message_Access;
  F2      : array (1 .. $NUMBER_OF_TIMES) of F2Message.F2Message_Access;
  F3      : array (1 .. $NUMBER_OF_TIMES) of access F3Message.Instance;
  F7      : array (1 .. $NUMBER_OF_TIMES) of F7Message.F7Message_Access;
  F8      : array (1 .. $NUMBER_OF_TIMES) of F8Message.F8Message_Access;
  F11     : array (1 .. $NUMBER_OF_TIMES) of F11Message.F11Message_Access;
  F25     : array (1 .. $NUMBER_OF_TIMES) of F25Message.F25Message_Access;

  Read_Container : Radar.ContainerMessage.Instance;

  My_Memory_Stream : Memory_Stream.Stream_Access := null;

  Gen_Data : Generated_Data.Array_Data_Type :=
    Generated_Data.Read_Data ($NUMBER_OF_TIMES);

  use type Ada.Calendar.Time;
begin
  -- Create
  declare
    Start_Create : Ada.Calendar.Time;
    Finish_Create : Duration;
  begin
    Start_Create := Ada.Calendar.Clock;

    for K in 1 .. $NUMBER_OF_TIMES loop
      Object (K) := Container.Add_Object;

      F4(K) := new F4Message.Instance;
      F7(K) := new F7Message.Instance;
      F11(K) := new F11Message.Instance;
      F8(K) := new F8Message.Instance;
      F25(K) := new F25Message.Instance;
      F2(K) := new F2Message.Instance;
      F3(K) := F2(K).Add_F3_Array;

      Object(K).Set_F4 (F4 (K));
      F4(K).Set_F7 (F7(K));
      F7(K).Set_F8 (F8(K));
      F7(K).Set_F11 (F11(K));
      F11(K).Set_F25 (F25(K));
      Object(K).Set_F2 (F2(K));

      Object(K).Set_F0 (Gen_Data(K).F0);
      Object(K).Set_F1 (Gen_Data(K).F1.all);

      F3(K).Set_F3 (Gen_Data(K).F3.all);

      F4(K).Set_F5 (Gen_Data(K).F5.all);
      F4(K).Set_F6 (Gen_Data(K).F6.all);
    end loop;
  end;
end;

```

```

F8(K).Set_F9 (Gen_Data(K).F9.all);
F8(K).Set_F10 (Gen_Data(K).F10.all);

F25(K).Set_F26 (Gen_Data(K).F26.all);
F25(K).Set_F27 (Gen_Data(K).F27.all);

F11(K).Set_F12 (Gen_Data(K).F12.all);
F11(K).Set_F13 (Gen_Data(K).F13.all);
F11(K).Set_F14 (Gen_Data(K).F14);
F11(K).Set_F15 (Gen_Data(K).F15);
F11(K).Set_F16 (Gen_Data(K).F16);
F11(K).Set_F17 (Gen_Data(K).F17);
F11(K).Set_F18 (Gen_Data(K).F18);
F11(K).Set_F19 (Gen_Data(K).F19.all);
F11(K).Set_F20 (Gen_Data(K).F20.all);
F11(K).Set_F21 (Gen_Data(K).F21.all);
F11(K).Set_F22 (Gen_Data(K).F22);
F11(K).Set_F29 (Gen_Data(K).F29);
F11(K).Set_F30 (Gen_Data(K).F30);
F11(K).Set_F23 (Gen_Data(K).F23);
F11(K).Set_F24 (Gen_Data(K).F24);
F11(K).Set_F28 (Gen_Data(K).F28.all);
end loop;

Finish_Create := Ada.Calendar.Clock - Start_Create;
Ada.Text_IO.Put (Finish_Create'Img);
Ada.Text_IO.Set_Col (15);
end;

-- Serialization
declare
  BUFFER_SIZE      : constant := $NUMBER_OF_TIMES * 700;

  Start_Serialize  : Ada.Calendar.Time;
  Finish_Serialize : Duration;
begin

  My_Memory_Stream := new Memory_Stream.Memory_Buffer_Stream (
    BUFFER_SIZE);

  Start_Serialize := Ada.Calendar.Clock;

  Container.Serialize_To_Output_Stream (My_Memory_Stream);

  Finish_Serialize := Ada.Calendar.Clock - Start_Serialize;
  Ada.Text_IO.Put (Finish_Serialize'Img);
  Ada.Text_IO.Set_Col (29);
end;

-- Size
Ada.Text_IO.Put (Container.Get_Cached_Size'Img);

-- Deserialization
declare
  Start_Deserialize : Ada.Calendar.Time;
  Finish_Deserialize : Duration;
begin
  Start_Deserialize := Ada.Calendar.Clock;

  Read_Container.Parse_From_Input_Stream (My_Memory_Stream);

  Finish_Deserialize := Ada.Calendar.Clock - Start_Deserialize;
  Ada.Text_IO.Put (Finish_Deserialize'Img);
end;

if not Generated_Data.Verify_Data (Gen_Data, Read_Container) then
  Ada.Text_IO.Put_Line ("Error");
end if;

```

```
    Memory_Stream.Free (My_Memory_Stream);  
end Main;
```



# Glossary

## **Abstract Syntax Notation (ASN)**

Platform independent language that is able to describe data structures in any programming language. See also Abstract Syntax Notation One (ASN.1).

## **Abstract Syntax Notation One (ASN.1)**

Standardized data serialization format.

## **Ada**

A programming language developed in the 1980s at the behest of the United States Department of Defense. The language was originally designed for embedded and real-time systems. As of this writing the latest stable release is Ada 2012. Previous stable releases include Ada 83, Ada 95 and Ada 2005.

## **Ada Util**

A collection of utility packages for Ada 2005.

## **Application Programming Interface (API)**

An established protocol for interfacing with software components.

## **AUnit**

AUnit is a unit testing framework developed and maintained by Ada-Core.

## **Comma-Separated Values (CSV)**

Data serialization format that separates text encoded values with commas.

## **Extensible Markup Language (XML)**

XML is markup language designed to be easily readable by both human and machine. It is used to store information, communicate information over a network etc.

## **GCC Runtime Library Exception**

Software licensed under GPL with the GCC Runtime Library Exception

can be redistributed with a different licenses as along as a set of conditions are fulfilled.

**GNAT**

GNAT is an open-source Ada compiler with support for Ada 83, Ada 95, Ada 2005 and Ada 2012.

**GNAT Component Collection JSON (GNATColl.JSON)**

GNAT Component Collection is a software library that provides general-purpose packages intended for use with the GNAT compiler. GNATColl.JSON is one such package, which provides Ada constructs for creating and parsing JSON encoded data.

**GNAT Programming Studio (GPS)**

Integrated development environment with Ada support.

**GNU General Public License (GPL)**

Open-source software license that exists in multiple versions.

**Interface Description Language (IDL)**

Describes an interface in a programming language independent way.

**JavaScript Object Notation (JSON)**

JavaScript Objection Notation is a text-based format for data-interchange.

**protoc**

The official Protocol Buffers compiler with built-in support for C++, Java and Python. Can be extended to support other languages through the use of a plug-in system.

**Protocol Buffers**

Google's open-source implementation of a format for serializing structured data for use in communication protocols and as a means to store information. The official implementation currently has support for generating code that support serialization and deserialization in C++, Java and Python.

**Remote Procedure Call (RPC)**

Technique that allows a program to call procedures which execute in a different process.

**Saab Security and Defence Solutions (Saab SDS)**

Saab's operations are divided into six business areas, one such business area is Security and Defence Solutions with a focus on providing solutions for protecting society and individuals.

**Schema**

See Interface Description Language (IDL).

**Standard output (stdout)**

A stream that is connected to a terminals output channel. A program typically displays information by writing to standard output.

**Tagged type**

A tagged type stores information about the type of an object so that it can identified at run-time.

**Target language**

The language a compiler translates source code into.



# Bibliography


- Ada Resource Association. (n.d.). Accessing the Ada Language Reference Manuals - Ada Resource Association. Retrieved May 23, 2013, from <http://www.adaic.org/ada-resources/standards/ada-95-documents/lrm>
- AdaCore. (n.d.). GNAT reference manual. Retrieved August 20, 2013, from [http://gcc.gnu.org/onlinedocs/gnat\\_rm/](http://gcc.gnu.org/onlinedocs/gnat_rm/)
- AdaCore. (n.d.). JSON: handling JSON data — GNATColl 1.6w documentation. Retrieved May 16, 2013, from <http://docs.adacore.com/gnatcoll-docs/json.html>
- Barnes, J. (2006). *Programming in Ada 2005*. Harlow: Addison-Wesley/Pearson Education.
- Barnes, J. (2008). Structure and visibility. In *Ada 2005 rationale* (Vol. 5020, pp. 93–117). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-79701-2\_4
- Bloch, J. (2005, October). How to design a good API and why it matters. Library-Centric Software Design LCSD'05. Retrieved from <http://lcsd05.cs.tamu.edu/slides/keynote.pdf>
- Bray, T., Paoli, J. & Sperberg-McQueen, C. M. (1998, February). Extensible Markup Language (XML) 1.0. Retrieved May 14, 2013, from <http://www.w3.org/TR/1998/REC-xml-19980210>
- Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Yergeau, E. M. F. (2013, February). Extensible Markup Language (XML) 1.0 (fifth edition). Retrieved May 14, 2013, from <http://www.w3.org/TR/REC-xml/>
- Brosgol, B. M. (2008). A comparison of the object-oriented features of ada 2005 and java™. In F. Kordon & T. Vardanega (Eds.), *Reliable software technologies — ada-europe 2008* (Vol. 5026, pp. 115–129). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-68624-8\_9
- Cohen, N. H. (1995). *Ada as a second language* (Second edition). McGraw-Hill Series in Computer Science. New York: McGraw-Hill.
- Crockford, D. (n.d.). JSON. Retrieved May 14, 2013, from <http://www.json.org>

- Crockford, D. (2006, July). The application/json media type for JavaScript Object Notation (JSON). IETF RFC 4627.
- Deutsch, L. P. (1996, May). DEFLATE compressed data format specification version 1.3. IETF RFC 1951.
- Dubuisson, O. (2000, June). ASN.1 communication between heterogeneous systems. Retrieved May 14, 2013, from <http://www.oss.com/asn1/resources/books-whitepapers-pubs/dubuisson-asn1-book.PDF>
- Free Software Foundation. (2013, April). GCC Runtime Library Exception rationale and FAQ. Retrieved May 16, 2013, from <http://www.gnu.org/licenses/gcc-exception-faq.html>
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994, November). *Design patterns: elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.
- Google. (2012a, April). code\_generator.h - Protocol Buffers — Google Developers. Retrieved May 21, 2013, from [https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.code\\_generator](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.code_generator)
- Google. (2012b, February). Developer Guide - Protocol Buffers — Google Developers. Retrieved April 5, 2013, from <https://developers.google.com/protocol-buffers/docs/overview>
- Google. (2012c, April). Encoding - Protocol Buffers — Google Developers. Retrieved May 18, 2013, from <https://developers.google.com/protocol-buffers/docs/encoding>
- Google. (2012d, April). Other languages - Protocol Buffers — Google Developers. Retrieved April 4, 2013, from <https://developers.google.com/protocol-buffers/docs/reference/other>
- Google. (2012e, April). Techniques - Protocol Buffers — Google Developers. Retrieved May 18, 2013, from <https://developers.google.com/protocol-buffers/docs/techniques>
- Google. (2012f, April). Third-Party Add-ons - Protocol Buffers — Google Developers. Retrieved May 18, 2013, from <https://developers.google.com/protocol-buffers/docs/proto>
- Google. (2013a, April). C++ Generated Code - Protocol Buffers — Google Developers. Retrieved August 5, 2013, from <https://developers.google.com/protocol-buffers/docs/reference/cpp-generated>
- Google. (2013b, March). Language Guide - Protocol Buffers — Google Developers. Retrieved May 18, 2013, from <https://developers.google.com/protocol-buffers/docs/proto>
- Google. (2013c, March). plugin.pb.h - Protocol Buffers — Google Developers. Retrieved April 5, 2013, from <http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/plugin.pb.h>

- IBM. (1972, July). IBM Fortran program products for OS and the CMS component of VM/370 general information. Retrieved May 14, 2013, from [http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-0\\_IBM\\_FORTTRAN\\_Program\\_Products\\_for\\_OS\\_and\\_CMS\\_General\\_Information\\_Jul72.pdf](http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-0_IBM_FORTTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf)
- Lippman, S. B., Lajoie, J. & Moo, B. E. (2005). *C++ primer* (4th ed.). Addison-Wesley.
- Pope, A. R. (1984, October). Encoding CCITT X.409 presentation transfer syntax. *SIGCOMM Comput. Commun. Rev.* 14(4), 4–10. doi:10.1145/1024908.1024909
- Robinson, W. (n.d.). python\_generator.cc - protobuf - Protocol Buffers - Google's data interchange format - Google Project Hosting. Retrieved May 20, 2013, from [http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/python/python\\_generator.cc](http://code.google.com/p/protobuf/source/browse/trunk/src/google/protobuf/compiler/python/python_generator.cc)
- Saab Group. (n.d.). Saab - defence and security. Retrieved April 8, 2013, from <http://www.saabgroup.com>
- Shafranovich, Y. (2005, October). Common format and MIME type for Comma-Separated Values (CSV) files. IETF RFC 4180.
- Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E. & Leroy, P. (2006a). Section 3: declarations and types. In *Ada 2005 reference manual. language and standard libraries* (Vol. 4348, pp. 19–85). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69336-9\_3
- Taft, S., Duff, R., Brukardt, R., Ploedereder, E. & Leroy, P. (2006b). Annex m (informative): summary of documentation requirements. In *Ada 2005 reference manual. language and standard libraries* (Vol. 4348, pp. 653–668). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69336-9\_25
- Taft, S., Duff, R., Brukardt, R., Ploedereder, E. & Leroy, P. (2006c). Section 13: representation issues. In *Ada 2005 reference manual. language and standard libraries* (Vol. 4348, pp. 267–304). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69336-9\_13





<b>Avdelning, Institution</b> Division, Department  ADIT, Dept. of Computer and Information Science 581 83 Linköping		<b>Datum</b> Date  DATEOFPUBLICATION
 Linköpings universitet		
<b>Språk</b> Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English  <input type="checkbox"/> _____	<b>Rapporttyp</b> Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	<b>ISBN</b> -  <b>ISRN</b> LiU-Tek-Lic-PUBYEAR:2013  <b>Serietitel och serienummer ISSN</b> Title of series, numbering -
<b>URL för elektronisk version</b>  http://XXX		Linköping Studies in Science and Technology  Thesis No. THESISNUMBER
<b>Titel</b> Title  Ada code generation support for Google Protocol Buffers  <b>Författare</b> Author  Niklas Ekendahl		
<b>Sammanfattning</b> Abstract  <p>We now live in an information society where increasingly large volumes of data are exchanged between networked nodes in distributed systems. Recent years have seen a multitude of different serialization frameworks released to efficiently handle all this information while minimizing developer effort. One such format is Google Protocol Buffers, which has gained additional code generation support for a wide variety of programming languages from third-party developers.</p> <p>Ada is a widely used programming language in safety-critical systems today. However, it lacks support for Protocol Buffers. This limits the use of Protocol Buffers at companies like Saab, where Ada is the language of choice for many systems. To amend this situation Ada code generation support for Protocol Buffers has been developed. The developed solution supports a majority of Protocol Buffers' language constructs, extensions being a notable exception.</p> <p>To evaluate the developed solution, an artificial benchmark was constructed and a comparison was made with GNATColl.JSON. Although the benchmark was artificial, data used by the benchmark followed the same format as an existing radar system. The benchmark showed that if serialization performance is a limiting factor for the radar system, it could potentially receive a significant speed boost from a substitution of serialization framework. Results from the benchmark reveal that Protocol Buffers is about 6 to 8 times faster in a combined serialization/deserialization performance comparison. In addition, the change of serialization format has the added benefit of reducing size of serialized objects by approximately 45%.</p>		
<b>Nyckelord</b> Keywords    Serialization, Google Protocol Buffers, Ada, GNATColl.JSON		