RecordFlux User's Guide

Release

AdaCore

Apr 27, 2026

Contents

1 Introduction **1**

- 1.1 Binary Data Formats and Communication Protocols . . . 1
- 1.2 Problems solved by RecordFlux . . . 2
- 1.3 Prerequisites . . . 3
- 1.4 Installation . . . 3

2 Overview **5**

- 2.1 The Protocol Verification Process . . . 5
- 2.2 The RecordFlux Toolset . . . 7
- 2.3 First Steps . . . 11

3 Appendix **22**

- 3.1 Command Line Options . . . 22
- 3.2 Specification Files . . . 25
- 3.3 External IO Buffers . . . 26
- 3.4 Reporting Errors . . . 27
- 3.5 Background . . . 27
- 3.6 GNU FDL . . . 27

1 Introduction

This document describes RecordFlux, a domain-specific language (DSL) and toolset for the creation of verifiable communication protocol implementations. After stating the problems RecordFlux helps solving, the manual outlines the required prerequisites and explains how to install and set up the tool. Subsequent sections give an introduction to the protocol verification process, describe the available tools and how they work together, and present introductory examples for message as well as protocol state machine formalization.

1.1 Binary Data Formats and Communication Protocols

BrakTooth, Infra:Halt, or Heartbleed are just a few of the high-profile security vulnerabilities related to communication protocols that have made it into the news in recent years. In fact, there are many more, but not all of them have a logo, a dedicated website and get the same amount of attention. Nonetheless, they have important aspects in common: they may be exploited to target critical infrastructure, they can create a serious supply chain security issue by affecting millions of devices in various configurations, and they are typically hard to prevent, identify or mitigate. This is especially

problematic as paradigms like Edge Computing, Predictive Maintenance or Autonomous Driving require ever more complex protocols and increased communication.

The reasons for those severe and recurring security vulnerabilities are manifold. Interaction between software components is governed by protocol and format specifications. Most of those specifications are incomplete, ambiguous, and even contradictory English language documents which need to be translated into software implementations manually. In addition to logic errors introduced during mandraulic translation phases, critical flaws are often poorly mitigated by widespread unsafe programming languages. It is not possible to express or check precise invariants characterizing the desired behavior, either at the source code or specification level. Consequently, many implementations contain severe security vulnerabilities waiting to be discovered and exploited by malicious actors.

Using standardized formats and protocols and relying on widespread implementations with good quality assurance can help to lower the risk of zero-day exploits. However, high profile vulnerabilities in commonly used protocol stacks, as found in Ripple20 or Amnesia:33, demonstrate that this by no means guarantees good security.

While standard implementations receive at least a certain level of scrutiny by the open source community and independent researchers, the situation is even worse for custom data formats and protocol implementations. Specifications (if present at all) and implementations are often only reviewed by a few people. Ad hoc design decisions resulting from urgent project needs complicate parsers and state machine implementations. Common pitfalls, which experienced protocol designers and implementers learn to avoid, like failing to check the length of a received message or making sure that privileged protocol states cannot be reached without prior authorization, can easily cause severe security problems in one-off in-house protocol implementations.

1.2 Problems solved by RecordFlux

With the SPARK language and toolset, programs can be proven to contain no runtime errors and to respect at all times the contracts with which the code is annotated. This process is highly automated and has been used successfully in mission critical software, for example in Aerospace & Defense, Avionics and Confidential Computing. However, the precision and trustworthiness that is gained comes at the price of additional effort for designing the software within the provable feature set of SPARK, specifying contracts, and guiding automatic theorem provers towards successful verification. The amount of code necessary for real-world data format and protocol implementations can often exceed the limit of what is formally provable with a manual implementation.

The RecordFlux toolset addresses this challenge by providing a high-level language tailored towards data formats and communication protocols which is precise and expressive enough to generate complex, formally-provable source code automatically. Its domain-specific language is used to precisely describe complex binary data formats and communication protocols. Through the toolset, users can formally verify specifications, generate provable SPARK code, and validate specifications using communication traces and existing implementations.

The formal specification in RecordFlux' language serves as a single source of truth in this process. Due to its abstract nature and its specialized support for binary formats and communication protocols, RecordFlux specifications can be written and understood by domain experts who are not necessarily programmers or verification engineers. The SPARK code that is generated from a valid RecordFlux specification can automatically be proven at the SPARK "gold" level. It contains no runtime errors like buffer overruns and integer overflows, and key properties – namely the behavior of the program with respect to the RecordFlux specification – are shown to be fulfilled at any time.

The RecordFlux language is expressive enough to define complex real-world binary messages and data formats. It can be used to precisely specify permitted value ranges, invariants that need to be maintained at all times, and complex dependencies between the elements of messages. The behavior of a protocol can be formally described by finite state machines which tightly integrate with the formal message specification. While abstract communication channels are used to define the integration with external components, external functions can be used to integrate with hand written code in a safe manner.

The automated correctness proofs performed at the specification level (i.e. before generating any source code) guarantee the following properties for message specifications:

- Determinism (no contradictory conditions)
- Liveness (no cycles)

- Reachability (no unused fields)
- Coherency (no overlaps)
- Completeness (no holes)

Consequently, the resulting message parsers and message serializers are – by construction – free from many issues that plague manual implementations of communication protocols. As functional correctness is proven statically, parsers guarantee that received messages comply with their specification, serializers ensure the creation of correct messages and state machines provide the specified protocol behavior.

The extensive integrity guarantees ensured for the generated code do not compromise performance and the compiled code has a small enough footprint to be usable in resource constrained, deeply embedded systems.

1.3 Prerequisites

The RecordFlux toolset generates code in the SPARK language, a formally analyzable subset of Ada 2012. AdaCore's *gnatprove* tool can be used to verify the source code produced by RecordFlux. While most of the generated code can be proven automatically, integrating it into a SPARK application requires good knowledge of the SPARK concepts, language and tools. The SPARK Reference Manual and the SPARK Toolset User's Guide give an overview of the SPARK language, process and toolset. This manual assumes familiarity with SPARK and a working setup of the SPARK tools. AdaCore's interactive training platform learn.adacore.com features extensive course material on SPARK, among other relevant topics such as mixed-language and embedded development.

While SPARK can be used to statically verify properties of the generated source code without compiling it, the GNAT compiler is required to produce a binary that can be deployed onto a target. RecordFlux makes very few assumptions on the target platform and the generated code can thus be used on various CPU architectures, native as well as cross configurations, and anything ranging from network servers to embedded systems. This manual assumes a working GNAT installation for the target platform as well as familiarity with the respective GNAT tools. See the GNAT documentation for details.

RecordFlux is integrated into GNAT Studio, AdaCore's lightweight multi-language IDE. For the integration, a working GNAT Studio installation and a correctly set up RecordFlux plug-in are required. See the *Installation section* for details. The use of GNAT Studio is not mandatory – all RecordFlux features can also be used on the command line.

The RecordFlux Simulator is a library which can be used to load RecordFlux specifications and dynamically interact with data or servers implementing the respective protocol from within a Python program. It is an optional component which does not require code generation or compilation. When used, it requires familiarity with Python 3 and of course a working Python environment.

1.4 Installation

System requirements

RecordFlux is supported on 64-bit Linux systems and should work on a variety of Linux distributions. The officially supported distributions are:

- Red Hat Enterprise Linux 7, 8, and 9
- SuSE Linux Enterprise Server 12 and 15
- Ubuntu 20.04 LTS and 22.04 LTS

The software has successfully been used on various other versions of Linux, including Arch Linux and Debian.

For installing RecordFlux itself, a *native* GNAT compiler for the host system must be installed. The following versions of GNAT are supported:

- GNAT Pro 22.2, 23.2, 24.2 and 25.0
- GNAT Community 2021

- FSF GNAT 11.2, 12.2, 13.2 or 14.1

A working installation of Rust 1.77 or newer must be installed. Rust 1.77 can be installed using rustup:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -q -y --profile␣
↪default --target x86_64-unknown-linux-gnu --default-toolchain 1.77
```

The latest release of GNAT Pro for Rust is also supported.

For compiling the generated code, one of the following versions of GNAT is required:

- GNAT Pro 21.2, 22.2, 23.2, 24.2 and 25.0
- GNAT Community 2021
- FSF GNAT 11.2, 12.2, 13.2 or 14.1

A successful installation of the native toolchain can be verified on the command line as follows:

```
$ gprbuild --version
GPRBUILD Pro AA.BB (YYYYMMDD) (x86_64-pc-linux-gnu)
Copyright (C) 2004-2022, AdaCore
...
```

The following external dependencies must be installed:

- GMP, *libgmp-dev* (Debian/Ubuntu), *gmp-devel* (Fedora) or *gmp* (Arch Linux)
- Graphviz (if graph visualization is used), *graphviz* in most distributions.

If FSF GNAT is used, the GNATcoll iconv binding must also be installed.

To run RecordFlux one of the following Python versions is needed:

- Python 3.10
- Python 3.11
- Python 3.12

In addition, the Python package installer *pip* is needed to install RecordFlux from the Python Package Index (PyPI). The tool can be installed using either the system package manager (*python3-pip* on Debian/Ubuntu/Fedora, *python-pip* on Arch Linux) or any other way described in the pip installation guide.

For the formal verification of the generated code, one of the following SPARK Pro versions is required:

- SPARK Pro 24.2 or 25.0

If you plan to use the RecordFlux Modeller, GNAT Studio needs to be installed and set up.

RecordFlux

Installing RecordFlux using pip requires an internet connection with access to the Python Package Index (PyPI) and a working GNAT installation. The following command will install RecordFlux and all required dependencies:

```
$ pip3 install RecordFlux
```

Alternatively, RecordFlux can be installed system-wide (run as root user) or into a virtual environment (run from within an activated *venv*). To check whether the installation was successful and the RecordFlux executable is in your path, request the version from the CLI:

```
$ rflx --version
RecordFlux 0.9.0
RecordFlux-parser 0.13.0
...
```

Once installed, the following command can be used to upgrade RecordFlux to the latest available version:

```
$ pip3 install RecordFlux --upgrade
```

VS Code Extension

The VS Code extension adds support for the RecordFlux language to VS Code. After the installation of RecordFlux, the extension can be installed using the CLI:

```
$ rflx install vscode
```

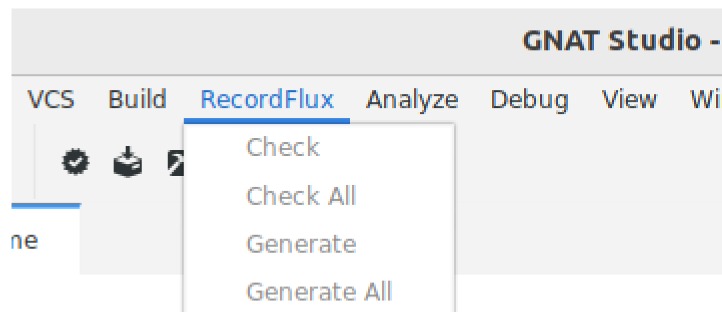
GNAT Studio Modeller Plugin

The RecordFlux Modeller is integrated into GNAT Studio as a plugin which needs to be installed before use. After installation of RecordFlux, run the installation procedure on the command line:

```
$ rflx install gnatstudio
```

Should your GNAT Studio settings directory be different from *$HOME/.gnatstudio*, the installation path can be changed using the parameter *--gnat-studio-dir*.

For the installation to become effective, GNAT Studio must be restarted. If the installation was successful, a RecordFlux menu will be available:

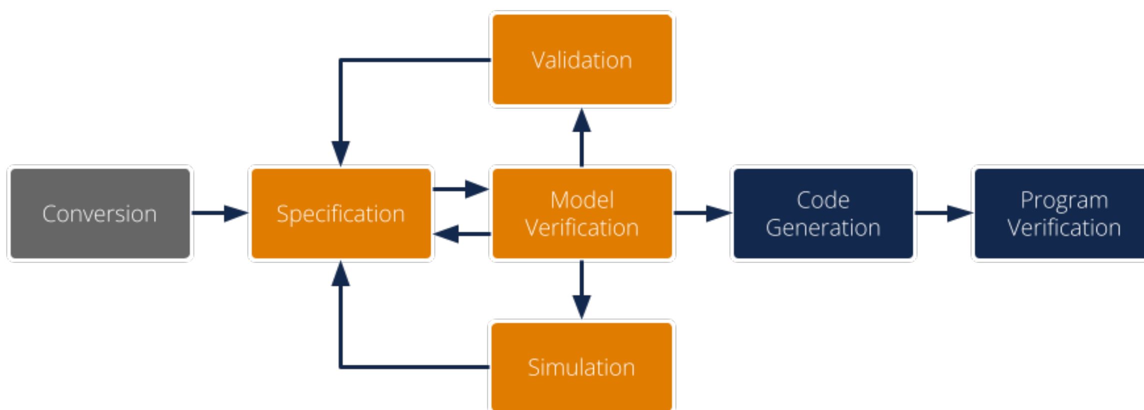


2 Overview

This section provides an overview of the protocol verification process, the parts of RecordFlux involved in working with binary communication protocols as well as some initial examples. For a complete description of the RecordFlux language refer to the RecordFlux Language Reference.

2.1 The Protocol Verification Process

Communication protocols don't usually have a formal specification today, but are typically specified in imprecise English language texts. There are efforts to establish a formal process to specify, e.g., internet protocols, but until such a process has been defined and widely adopted, additional work is needed to create formal specifications from informal ones. This section outlines a typical process starting from an informal specification leading to a formally verified protocol implementation.



The process starts with the optional *Conversion* step, where the protocol in question or part of it is translated from a machine-readable input specification into a formal RecordFlux specification. As mentioned above, only few protocols offer such a specification today. However, the Internet Assigned Numbers Authority (IANA) maintains a repository of machine-readable Number Resources for a variety of protocols. Those definitions can be automatically converted into RecordFlux specifications to improve one of the most tedious tasks of protocol formalization, namely the correct translation of long lists of name / constant pairs.

A main activity when creating a formally verified protocol implementation is clearly the actual *Specification* of the machine readable formal definition of the protocol. Working from a paper specification, the RecordFlux user can use the tool's language to express basic data types, the format of messages and the behavior of communicating entities. To improve maintainability and to structure the specification in a coherent way, multiple specification documents can reference each other to form the overall specification. The language is designed to be modular: protocols do not need to "know" each other, and two protocols can be associated with each other independent of their respective specifications. In accordance with many protocol layering approaches, this allows for the creation of protocol libraries with reusable building blocks.

During specification development one question is of central importance: Does the formal specification correspond to the protocol? The process of ensuring that the correct thing has been specified is called *Validation*. As even the source specification may be vague, incorrect, contradictory or simply different from how the protocol is being implemented in practice, checking the formal specification against the output of existing implementations or protocol samples is of great value. The RecordFlux toolset provides ways to do exactly this: check whether a valid protocol sample is accepted by the specification, or whether an invalid sample is correctly rejected. The validation will typically be done while developing the formal specification, such that its validity is established incrementally.

Similar in its goals, but different in terms of process and implementation is the optional *Simulation* step. While for validation the focus is mostly on the correctness of message formats, simulation allows for direct interaction with existing implementations. To validate the formal specification, it can be loaded in a Python program and used to receive, check and send messages adhering to the protocol. While the main objective is to detect unexpected behavior, it can also be used to build Python tools accompanying the formally verified protocol stack with little effort. Of course the generated code could be used to interact with existing software, too. However, rapid prototyping and the vast source of third-party libraries make a Python environment a much more efficient choice.

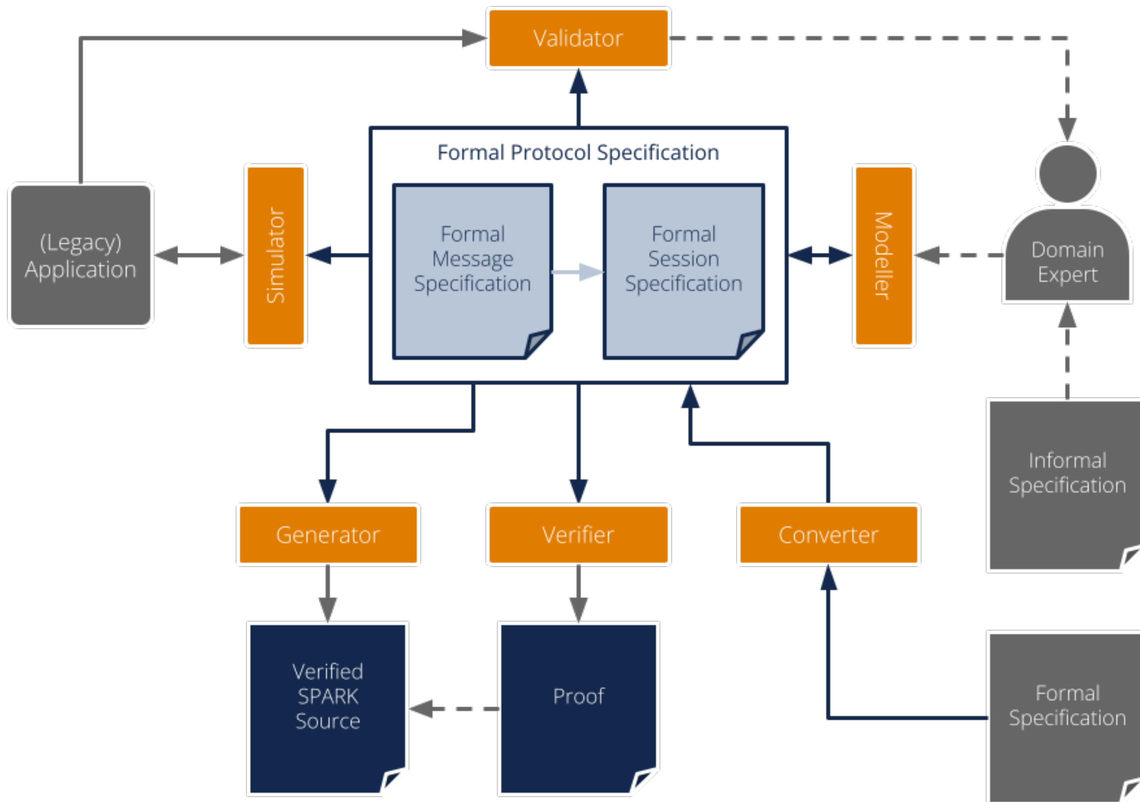
The *Model Verification* step answers another question: Does the formally specified protocol have the desired properties? It's worth noting that this is different from the goals of validation. Validation is to ensure that we do the right thing, verification is to ensure that we do the thing right. The model verification built into RecordFlux will typically be done iteratively during specification development. It ensures a number of static properties, such as determinism of the message parser or that generated code can always be proven to contain no runtime errors. There are also user-defined properties that are verified at the model level, e.g. that user-defined transitions are not statically false.

When the RecordFlux model has successfully been verified, the Code Generation phase transforms it into SPARK

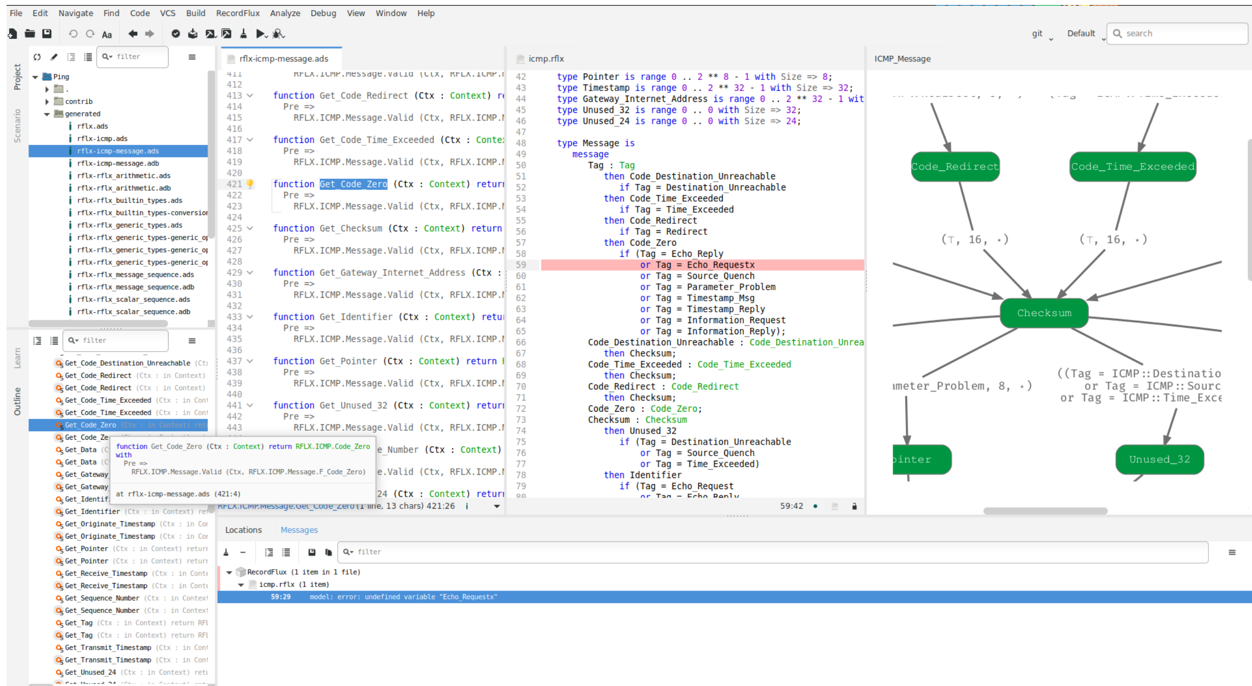
source code. It can be combined with hand-written SPARK, Ada, or C code and compiled into binaries using the GNAT compiler. The code generation process can be tailored towards resource-constrained, embedded systems with precise control over memory allocation and buffer sizes. As only few basic runtime functions are required, the generated code is suitable for environments with minimal runtimes.

An explicit design goal of RecordFlux is that all code generated from a successfully verified model can automatically be proven correct, without a need for user modifications or interactive proofs. This is done in the Program Verification step where the generated code, together with the hand written code it may be integrated in, is verified using the SPARK toolset.

2.2 The RecordFlux Toolset



As outlined above, a domain expert creates the formal representation of a protocol in the RecordFlux language. Having a text representation, RecordFlux specifications can be produced with any IDE or text editor, managed in version control systems, or compared using text comparison tools. However, the RecordFlux Modeller provides additional capabilities tailored to protocol specification development. A graphical representation of messages and state machines helps users to understand the structure of more complex specifications. The specification can be edited side-by-side with generated code, handwritten code, and other relevant artifacts. All specifications, or only a single specification file, can be checked, and SPARK code can be generated directly from within the Modeller.



All functionality in the Modeller is also available on the command line through the *rflx* command. This is RecordFlux's main CLI, which has a number of subcommands of the form *rflx <subcommand>*. Some functionality is available only on the command line. See the output of *rflx --help* for a list of all subcommands and global options and use *rflx <subcommand> --help* to show options specific to a particular subcommand.

The Converter, available through the *rflx convert* subcommand of the RecordFlux CLI, provides the possibility to convert IANA Number Resources into RecordFlux specifications. Given a Number Resources file in XML format and an output directory, the converter produces a RecordFlux specification containing representations of the respective number definitions.

To check whether a given specification is correct, the Verifier performs a number of formal verification steps. The tool is available from within the Modeller (*RecordFlux > Check* or *RecordFlux > Check All*) but can also be run from the command line using the *rflx check* interface. The CLI exits with a non-zero status code in case of errors and thus can be easily integrated into a CI/CD pipeline. With the command line interface, arbitrarily many (unrelated) specifications can be checked at once.

Similar to the Verifier, the Generator is available from within the Modeller as well as on the command line through the *rflx generate* interface. As only successfully verified specifications are guaranteed to lead to provable SPARK code, the Generator automatically performs the verification, unless this is suppressed using the *--no-verification* switch (e.g. for known-good specifications that have been checked in a CI/CD pipeline). SPARK source files are generated into the directory specified by the *-d* switch on the command line. The result can be included in the list of source directories and analyzed by *gnatprove* as usual (see the SPARK User's Guide for details).

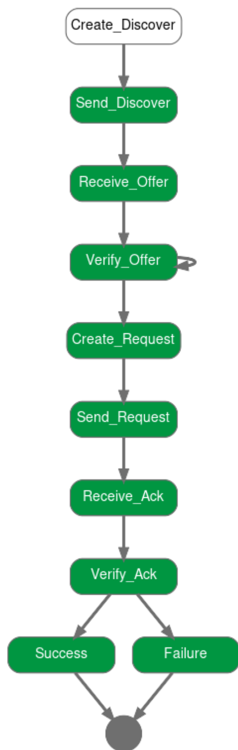
The Optimizer reduces the size of the generated state machine code. In order to use the Optimizer, a project file for the SPARK code must be provided by the user and passed as an argument to *rflx optimize*.

The Validator is available through the *rflx validate* subcommand on the command line. It can be used to check whether a message specification correctly formalizes real-world data, or vice versa, whether a given data sample corresponds to the specification. Two types of samples can be used: valid samples which must be accepted by a specification (passed with the *-v* option) and invalid samples which must be rejected (passed with the *-i* option). All samples must be raw binary files without any metadata. There are two ways how samples can be provided to the tool and the same rules apply for both *-v* and *-i*. If there are several samples of the same kind in one directory and the samples have a *.raw* file extension, then it is sufficient to just provide this directory. Otherwise, the paths of the individual sample files must be provided one by one. In the latter case the sample file can have any extension or even have no extension. However, if

it has an extension, then it must be included in the path as well. There can be as many *-v* and *-i* options given to the tool as needed. However, each of those options must have exactly one argument. Raw packets can, e.g., be exported from packet analyzers like Wireshark or extracted from a PCAP file using this script. To facilitate execution within a CI/CD pipeline, the *--abort-on-error* switch causes the tool to exit with an error code if any samples are rejected. Upon completion, the Validator will produce a report, with an option to display how much of a message has been covered:

```
$ rflx validate \
    --coverage \
    -i tests/examples/data/http_2/frame/invalid \
    -v tests/examples/data/http_2/frame/valid \
    examples/specs/http_2.rflx HTTP_2::Frame

Parsing examples/specs/http_2.rflx
Processing HTTP_2
[...]
Verifying HTTP_2::Frame
tests/examples/data/http_2/frame/valid/GOAWAY_58.raw                             PASSED
tests/examples/data/http_2/frame/valid/GOAWAY_66.raw                             PASSED
tests/examples/data/http_2/frame/valid/HEADERS_26.raw                            PASSED
[...]
tests/examples/data/http_2/frame/valid/PING_44.raw                               PASSED
tests/examples/data/http_2/frame/valid/PING_45.raw                               PASSED
tests/examples/data/http_2/frame/valid/PING_48.raw                               PASSED
tests/examples/data/http_2/frame/valid/PING_49.raw                               PASSED
tests/examples/data/http_2/frame/valid/PUSH_PROMISE_63.raw                       PASSED
tests/examples/data/http_2/frame/valid/RST_STREAM_64.raw                         PASSED
tests/examples/data/http_2/frame/valid/SETTINGS_0.raw                            PASSED
[...]
tests/examples/data/http_2/frame/valid/WINDOW_UPDATE_10.raw                      PASSED
[...]

--------------------------------------------------------------------------------
                     RecordFlux Validation Coverage Report
Directory: .
--------------------------------------------------------------------------------
File                                          Links       Used        Coverage
http_2.rflx                                      56         37          66.07%
--------------------------------------------------------------------------------
TOTAL                                            56         37          66.07%
--------------------------------------------------------------------------------


================================================================================
                                Uncovered Links
================================================================================


                                  http_2.rflx
--------------------------------------------------------------------------------
http_2.rflx:141:13: missing link     Stream_Identifier     ->      Pad_Length
http_2.rflx:143:13: missing link     Stream_Identifier     ->   Application_Data
http_2.rflx:146:13: missing link     Stream_Identifier     ->    Exclusive_Flag
http_2.rflx:151:13: missing link     Stream_Identifier     ->    Exclusive_Flag
http_2.rflx:170:13: missing link        Pad_Length         ->   Application_Data
[...]
http_2.rflx:69:13: missing link Settings_Value_Enable_Push ->        Final
```

If more complex validation beyond checking messages is required, the Simulator can be used. It allows loading RecordFlux message specifications into a Python program and using the resulting model to parse and generate messages at runtime. While the code does not benefit from the formal guarantees of generated SPARK code, errors can be caught at runtime. This makes the Simulator a useful tool during specification development to validate a specification against an existing real-world implementation. An example of using the Simulator can be found in examples/apps/ping/ping.py in the RecordFlux source repository.



The Visualizer can be used to create graphical representations of a formal RecordFlux specification. It is available on the command line through the *rflx graph* command. It creates images for all messages and state machines found in the specifications passed on the command line and stores them in the output directory specified by the *-d* switch. By default, SVG images are created, but the *-f* switch may be used to select alternative formats like JPG, PNG or PDF. The *-i* switch may be used to filter out state machine states which must not be included in the output, which can be helpful, for example, to eliminate error states which may complicate the non-error case unnecessarily.

```
$ rflx graph -d out dhcp_client/specs/dhcp_client.rflx

Parsing dhcp_client/specs/dhcp_client.rflx
Parsing dhcp_client/specs/ipv4.rflx
Parsing dhcp_client/specs/dhcp.rflx
Processing IPv4
Processing DHCP
Processing DHCP_Client
Creating out/IPv4_Option.svg
Creating out/IPv4_Packet.svg
Creating out/DHCP_Static_Route.svg
Creating out/DHCP_Option.svg
Creating out/DHCP_Message.svg
Creating out/DHCP_Client_Session.svg
```

2.3 First Steps

The example used here is a minimal binary publish-subscribe ("Pub-Sub") protocol (i.e., a message broker). In the following sections we will first formally describe the message format using RecordFlux, generate SPARK code, and build a simple server which we then prove to contain no runtime errors. In a subsequent section, we will also define the protocol behavior using the RecordFlux language which will give us an abstract formal definition of the protocol and further reduce the amount of hand-written code.

In our example protocol, there is only one single implicit channel that clients can subscribe to. Once subscribed, a client may publish messages which the broker distributes to all other currently subscribed clients. A client does not receive messages published by itself. When finished, clients can unsubscribe from the broker and will no longer be able to publish or receive messages.

A number of status messages are used to communicate the result of an operation from the broker to the clients. The *SUCCESS* message indicates that a *SUBSCRIBE*, *PUBLISH* or *UNSUBSCRIBE* operation completed successfully. An *ERROR_NOT_SUBSCRIBED* message is emitted when a client tries to publish or unsubscribe while not currently subscribed, implying that a client has to be subscribed to publish. An *ERROR_NO_SUBSCRIBERS* is sent by the broker if no other clients are subscribed when publishing information, which is considered an error in this example. An *ERROR_MESSAGE_TOO_LONG* message is emitted when the published message exceeds an implementation-defined length, which may be shorter than the maximum possible message length. An *ERROR* message is sent back in all other cases.

Example: A Formally-Verified Message Parser

The messages of our example publish-subscribe protocol are a binary format which is encoded as follows:

Field	Length [bits]	Domain	Description
Identifier	12	1 <= *Identifier* <= 4000	Unique client identifier
Command	4	*SUBSCRIBE* = 1, *PUBLISH* = 3, *UNSUBSCRIBE* = 4, *ERROR* = 11, *ERROR_NOT_SUBSCRIBED* = 12, *ERROR_NO_SUBSCRIBERS* = 13, *ERROR_MESSAGE_TOO_LONG* = 14, *SUCCESS* = 15	Message type
Length	8	0 <= *Length* < 2^8	Length of published payload in bytes, present only when *Command* = *PUBLISH*
Payload	8 * Length	Arbitrary bytes	Payload of published data, present only when *Command* = *PUBLISH* and *Length* > 0

As we can see, the message has several interesting properties which our formal specification needs to cover: Some of the fields are not multiple of 8 bits (*Identifier*, *Command*) or not byte aligned (*Command*). While representable in the available bit width, not all values are valid for all fields. For *Identifier*, the values 0 and 4001 .. 4095 are invalid and must be rejected. Likewise, the *Command* field has invalid values (e.g., 2) which do not represent a valid message type. There are also optional fields like *Length* and *Payload*, which are present only for messages where the *Command* field has the value 3 (i.e. *PUBLISH*). Lastly, the *Payload* field has a variable length determined by *Length* field.

Files, Packages and Names

Let's formalize our Pub-Sub message format in the RecordFlux language. RecordFlux types (scalar types as well as messages) are defined in modules known as packages, whose syntax is inspired by SPARK. There is exactly one package per file; the file name has to be the same as the package name, folded to lowercase, and the file extension is "rflx". We will use the name "Pub Sub" for our example protocol and create a file named *pub_sub.rflx* for the specification:

```
package Pub_Sub is
   -- Type specifications (basic types, messages) go here
end Pub_Sub;
```

Single line comments, as in SPARK or SQL, start with a double hyphen (--) and comprise all text until the end of the line. There are no block comments in the RecordFlux language.

Names follow mostly the same rules as for the SPARK language: Letters A-Z, a-z, digits and underscores can be used. A name must not begin with an underscore or a digit, and must not end with an underscore; consecutive underscores are also prohibited. Names in RecordFlux are case-insensitive, i.e. *Pub_Sub* and *pub_sub* refer to the same thing. Future versions of RecordFlux will also be case-preserving, hence it is already considered good practice to use identical casing for all appearances of a name.

Scalar Types

To limit the size and allowed values of a numeric field in a message, we need to define a type that has the desired properties. For the *Identifier* field we define a type that can represent values from 1 to 4000, inclusive, and whose instances occupy 12 bits:

```
type Identifier is range 1 .. 16#F_A0# with Size => 12;
```

Numbers, like in SPARK, are base 10 by default, but can be represented in arbitrary bases by using the *<base>#<value>#* notation. In the above example we represent the upper limit of 4000 by its hexadecimal representation *16#F_A0#*. Note, that single underscores (_) can be introduced into numeric literals in arbitrary positions to improve readability.

As for the package name, "*Identifier*" adheres to the naming rules stated above. The range declares the lower, and upper bounds of a field of this type and the Size attribute defines the precise storage space of the type in bits. For consistency, sizes in RecordFlux specifications are always defined in bits – without exception. In the message definition below we will see how to deal with protocols that define lengths in terms of bytes.

Of course the upper bound must be consistent with the available space defined in the Size attribute. If we were to define the above type with an upper bound of 5000 (which obviously does not fit into 12 bits), the RecordFlux toolset would flag our specification as illegal:

```
Processing Pub_Sub
pub_sub.rflx:3:9: model: error: size of "Identifier" too small
```

We could proceed and define the Type field of our protocol as a 4 bit numeric value, similar to the Identifier field. While we could even express the fact that the type field must not be zero by choosing 1 as the lower limit, there would still be values that will be accepted (e.g. 2) but which are invalid according to the protocol specification. An enumeration type is much better suited to represent discrete choices as in the Command field:

```
type Command is
(
   SUBSCRIBE => 1,
   PUBLISH => 3,
   UNSUBSCRIBE => 4,
   ERROR => 11,
   ERROR_NOT_SUBSCRIBED => 12,
   ERROR_NO_SUBSCRIBERS => 13,
   ERROR_MESSAGE_TOO_LONG => 14,
   SUCCESS => 15
) with Size => 4;
```

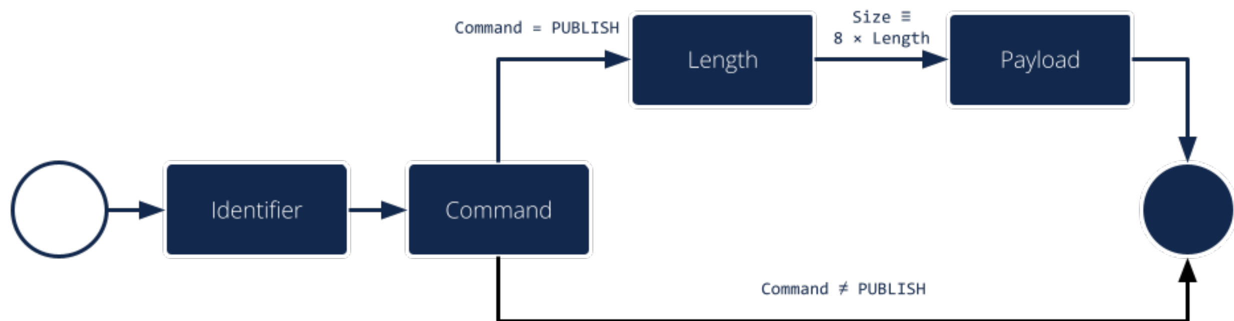
As before, the *type* keyword introduces a type named *Command* with a storage size of 4 bits. The possible choices for the enumeration type are listed in parenthesis as pairs of name and value, separated by an arrow (=>). Fields with an enumeration type are only considered valid if their value matches one of the enumeration choices and are rejected otherwise. Refer to the section on *Always_Valid* enumeration types in the Language Reference on how to handle fields where all values are valid, but only some have names.

As with integer ranges, the RecordFlux toolset will protect the user from including values in the specification that cannot be represented in the available space. If we changed the value of *SUCCESS* to *8#42#* (octal 42 / decimal 34), which does not fit into 4 bits, we get an error:

```
Processing Pub_Sub
pub_sub.rflx:5:9: model: error: size of "Command" too small
```

Message Types

With the package and the interesting scalar types in place, we can formalize the actual message structure. As can be seen in the format description, our pub-sub protocol message has the interesting property that some fields are only present under certain conditions. Formalizing such a message is easy in the RecordFlux language and hints at an important property of its messages: While simple messages may appear similar to SPARK records, or structs of linear fields in other languages, they are in fact arbitrarily complex directed acyclic graphs of fields. The edges of those messages carry information about conditions, the starting position and bit size of subsequent fields, which may in fact depend on previous fields in the graph:



Just like scalars, messages are also types. The specification for our pub-sub message is as follows:

```
type Length is unsigned 8;
```

```
type Message is
   message
      Identifier : Identifier;
      Command : Command
         then Length
            if Command = PUBLISH
         then null
            if Command /= PUBLISH;
      Length : Length;
      Payload : Opaque
         with Size => 8 * Length;
   end message;
```

This contains quite a few new constructs – let's unpack them one by one. The *Length* type is an integer type as we have used it for the *Identifier* previously. It's worth noting that the ranges of integers may contain arbitrary expression, such

as exponentiation (**) or subtraction (-) in the example above.

A message is encompassed by a *message ... end message* block which contains a list of fields, edges, and conditions making up the message graph. Readers familiar with other languages may notice that we call our message "*Message*" which appears to conflict with the *message* keyword. The same is true for the message field "*Identifier*" which has the same name as the previously defined type. Contrary to other languages (and SPARK in particular), RecordFlux' language allows the use of keywords as type names, and keywords and type names as identifiers to give the user the greatest possible flexibility when formalizing an existing protocol.

Similar to records in SPARK, message fields are delimited by a colon (*:*) and terminated by a semicolon (*;*). Thus *Field_Name : Field_Type;* declares a field with the name "*Field_Name*" of type "*Field_Type*". It gets interesting for optional fields: the *then* keyword creates an edge to another field explicitly, usually in conjunction with a condition marked by the *if* keyword. In our example, the *Length* field follows the *Command* field if (and only if) the *Command* field contains the value *PUBLISH*. As in SPARK, and unlike other languages, RecordFlux uses a single equal sign for comparison. For enumeration types like *Command*, the enumeration element name has to be used rather than its numeric field value.

When no *then* keyword is present for a field, there is an implicit *then* keyword with the next field as a target, e.g. an implicit "*then Payload*" for the Length field in the above example. Multiple *then*-clauses are allowed for a single field to define multiple edges to other fields under different conditions. There may even be multiple edges (i.e. *then* keywords) leading to the same field; e.g., to model separate conditions that may cause a field to be present. To indicate that a message ends when a specific condition is true, the *null* keyword can be used in a *then* clause. In the above example, the message is considered complete when Command does not have the value *PUBLISH* (*then null if Command /= PUBLISH*). If none of the *then*-clauses match for a given input, the message is considered invalid and rejected.

Among other things, the RecordFlux Verifier ensures that *then*-clauses are mutually exclusive, i.e. the resulting parser is deterministic. If we changed the above example to contain two *then*-clauses for Command which are not mutually exclusive, our specification would get rejected:

```
Command : Command
   then Length
      if Command = PUBLISH
   then Length
      if Command = PUBLISH or Command = UNSUBSCRIBE;
```

```
Processing Pub_Sub
pub_sub.rflx:21:10: model: error: conflicting conditions for field "Command"
pub_sub.rflx:23:19: model: info: condition 0 (Command -> Length): Command = Pub_
↪Sub::PUBLISH
pub_sub.rflx:25:19: model: info: condition 1 (Command -> Length): Command = Pub_
↪Sub::PUBLISH or Command = Pub_Sub::UNSUBSCRIBE
```

Another noteworthy aspect of our example is the *Payload* field. It is of type *Opaque*, one of RecordFlux's few built-in types defining a sequence of arbitrarily many bytes. While the content of opaque fields is not defined, their length can be specified using a *Size* attribute. In our case, it defines the total length of the *Payload* field to be eight times the value of the *Length* field, since the *Length* field specifies the number of bytes and all RecordFlux field sizes are in bits.

Verifying the Specification and Generating Code

With the formal specification in place, we can use the RecordFlux verifier to prove the formal model of our specification. On the command line, use the *rflx check* subcommand with the specification file as its only parameter:

```
$ rflx check specs/pub_sub.rflx
Parsing specs/pub_sub.rflx
Processing Pub_Sub
```

The RecordFlux Modeller can also be used to create and check specifications. A simple project file named *pub_sub.gpr* located in the root directory of our example project configures directories for hand-written code (*src*), generated code (*generated*) and specifications (*specs*). It also enables support for the RecordFlux specification language and sets the output directory for generated code:

```
project Pub_Sub is

   for Languages use ("Ada", "RecordFlux");
   for Source_Dirs use ("src", "generated", "specs");

   package Recordflux is
      for Output_Dir use "generated";
   end Recordflux;

end Pub_Sub;
```

In GNAT Studio, the *RecordFlux > Check* menu entry can then be used to verify a specification, the *Check All* entry verifies all RecordFlux specifications in a project at once. To generate SPARK from the formal specification, either use *RecordFlux > Generate* from within GNAT Studio, or *rflx generate* on the command line (the output directory must be specified using the *-d* switch, as the *rflx* command line tool does not yet read this information from the project file):

```
$ rflx generate -d generated specs/pub_sub.rflx
Parsing specs/pub_sub.rflx
Processing Pub_Sub
Generating Pub_Sub::Identifier
Generating Pub_Sub::Command
Generating Pub_Sub::Length
Generating Pub_Sub::Message
Creating generated/rflx-pub_sub.ads
Creating generated/rflx-pub_sub-message.ads
Creating generated/rflx-pub_sub-message.adb
Creating generated/rflx-rflx_arithmetic.ads
Creating generated/rflx-rflx_builtin_types-conversions.ads
Creating generated/rflx-rflx_builtin_types.ads
Creating generated/rflx-rflx_generic_types.ads
Creating generated/rflx-rflx_generic_types-generic_operators.ads
Creating generated/rflx-rflx_generic_types-generic_operations.ads
Creating generated/rflx-rflx_message_sequence.ads
Creating generated/rflx-rflx_scalar_sequence.ads
Creating generated/rflx-rflx_types.ads
Creating generated/rflx-rflx_types-operators.ads
Creating generated/rflx-rflx_types-operations.ads
Creating generated/rflx-rflx_arithmetic.adb
Creating generated/rflx-rflx_generic_types-generic_operations.adb
Creating generated/rflx-rflx_message_sequence.adb
Creating generated/rflx-rflx_scalar_sequence.adb
Creating generated/rflx.ads
```

The generated code can be integrated into an existing SPARK package hierarchy by passing the *--prefix=Root.Package* parameter to the generate command. Note, that the root packages are assumed to exist and the code generator will not generate them:

```
$ rflx generate --prefix=My_Package -d generated specs/pub_sub.rflx
Parsing specs/pub_sub.rflx
```

(continues on next page)

(continued from previous page)

```
Processing Pub_Sub
Generating Pub_Sub::Identifier
Generating Pub_Sub::Command
Generating Pub_Sub::Length
Generating Pub_Sub::Message
Creating generated/my_package-pub_sub.ads
Creating generated/my_package-pub_sub-message.ads
[...]
```

Using the Generated Code

To allow us to focus on the protocol part of our implementation, we will assume two SPARK packages: a packet *Socket* for communication with clients and a packet *DB* for handling client subscriptions. The communication packet has the following specification:

```
with RFLX.RFLX_Types;

package Socket
   with SPARK_Mode, Abstract_State => Network
is
   Initialized : Boolean := False with Ghost;

   procedure Initialize (Port : Natural)
      with Global => (In_Out => Initialized, Output => Network),
           Pre    => not Initialized,
           Post   => Initialized;

   procedure Send (Data : RFLX.RFLX_Types.Bytes)
      with Global => (Input => Initialized, In_Out => Network),
           Pre    => Initialized;

   procedure Receive (Data    : out RFLX.RFLX_Types.Bytes;
                      Success : out Boolean)
      with Global => (Input => Initialized, In_Out => Network),
           Pre    => Initialized;
end Socket;
```

The library implements a shared communication channel based on the UDP protocol. Here it is simply for demonstration purposes: a realistic implementation would allow multiple connections and more detailed error handling. The Initialize procedure must be called once with the UDP port number to use before sending or receiving any data. This is enforced by the SPARK contracts. The *Send* procedure sends out the complete buffer passed in through the *Data* parameter. The *Receive* procedure fills its *Data* parameter with the received packet and signals success through the *Success* parameter.

The communication package already uses code generated from the protocol specification. The *RFLX.RFLX_Types* package is the default instance of the generic types package *RFLX.RFLX_Generic_Types* which contains basic definitions for scalar types, indexes, offsets and arrays of bytes which are used throughout the generated code. Instantiations with custom types are possible, but in this example our communication library uses the default type for byte buffers *RFLX.RFLX_Types.Bytes* directly.

Our second helper package for handling subscribers to our pub-sub protocol also uses generated code. For every RecordFlux package, a corresponding SPARK package is generated within the *RFLX* hierarchy. Consequently, the package *RFLX.Pub_Sub* contains code generated for the (scalar) type definition from the *Pub_Sub* package. To avoid

unnecessary type conversions, we use the generated type *RFLX.Pub_Sub.Identifier* in the database package:

```
with RFLX.Pub_Sub;

package DB with
   SPARK_Mode, Abstract_State => Subscribers
is

   function Is_Subscribed (ID : RFLX.Pub_Sub.Identifier) return Boolean
      with Global => (Input => Subscribers);

   procedure Unsubscribe (ID : RFLX.Pub_Sub.Identifier)
      with Global => (In_Out => Subscribers),
           Pre    => Is_Subscribed (ID);

   procedure Subscribe (ID : RFLX.Pub_Sub.Identifier)
      with Global => (In_Out => Subscribers),
           Post   => Is_Subscribed (ID);

   type Identifiers is array (Natural range <>)
      of RFLX.Pub_Sub.Identifier;

   function Current_Subscribers return Identifiers
      with Global => (Input => Subscribers);
end DB;
```

The subprograms allow for subscribing, unsubscribing and checking whether a specific identifier is currently subscribed. Contracts ensure that unsubscribing is possible only for subscribed entities. The *Current_Subscribers* function returns a list of currently subscribed identifiers.

With the helper packages described, we can implement the actual logic of the message broker in SPARK. The broker itself is very simple, consisting only of a single Run procedure which is expected to run in a loop in the main program:

```
with Socket;

package Broker
   with SPARK_Mode
is
   procedure Run with
      Pre => Socket.Initialized;
end Broker;
```

Before we go into the details of the *Run* subprogram, we implement a helper procedure *Send_Status* to construct and send status and error messages. The procedure will be private within the body of the *Broker* package and create a simple message (i.e. one that does not have *PUBLISH* as a command). This helps simplifying our state machine logic and also demonstrates the principles of message generation:

```
package body Broker with SPARK_Mode
is
   use type RFLX.RFLX_Types.Bytes_Ptr;
   use type RFLX.RFLX_Types.Index;

   subtype Status is RFLX.Pub_Sub.Command
   range RFLX.Pub_Sub.ERROR .. RFLX.Pub_Sub.SUCCESS;
```

(continues on next page)

(continued from previous page)

```
   procedure Send_Status (Id : RFLX.Pub_Sub.Identifier; St : Status)
      with Pre => Socket.Initialized;

   procedure Send_Status (Id : RFLX.Pub_Sub.Identifier;
                          St : Status)
   is
      Context : RFLX.Pub_Sub.Message.Context;
      Buffer : RFLX.RFLX_Types.Bytes_Ptr :=
         new Types.Bytes'(1 .. 4_096 => 0);
   begin
      RFLX.Pub_Sub.Message.Initialize (Context, Buffer);
      RFLX.Pub_Sub.Message.Set_Identifier (Context, Id);
      RFLX.Pub_Sub.Message.Set_Command (Context, St);
      RFLX.Pub_Sub.Message.Take_Buffer (Context, Buffer);
      Socket.Send (Buffer.all);
      RFLX.RFLX_Types.Free (Buffer);
   end Send_Status;
   -- ...
end Broker;
```

The *Send_Status* procedure receives the identifier of the client to address the message to. Status is a subtype of the generated *RFLX.Pub_Sub.Command* enumeration type which corresponds directly to the *Command* enumeration in our RecordFlux specification. We have chosen the enumeration type in a way that the subtype comprises only commands for simple status messages. SPARK will then verify that we do not accidentally pass, e.g., *PUBLISH* to *Send_Status*.

Working with messages always requires a context. Among other things, the context holds a pointer to the actual message, the current state of the message serialization or parsing, and the actual field values for scalar types. The raw message data is held in a buffer of the previously mentioned *Bytes* type. After initializing the context with a pointer to that buffer using the *Initialize* procedure, the generated *Set_<FieldName>* procedures can be used to set the value of scalar message fields. For example, to set the *Identifier* field of Message in the *Pub_Sub* specification, the procedure *RFLX.Pub_Sub.Message.Set_Identifier (Context, Value)* would be used.

The SPARK contracts on the generated subprograms ensure that required initialization is performed, and that fields are set in the correct order. If we forgot to call Initialize, *gnatprove* would emit an error:

```
broker.adb:28:27: medium: precondition might fail, cannot prove RFLX.Pub_Sub.Message.Has_
↪Buffer (Ctx)
```

SPARK also proves that message fields are set in the right order. While it may not result in a problem for the scalar values in our message specification, field position may depend on previous fields and setting them out of order could result in incorrect messages. Hence, the order is always enforced. If we set *Command* before *Identifier*, we'd get another error:

```
broker.adb:28:27: medium: precondition might fail, cannot prove RFLX.Pub_Sub.Message.
↪Valid_Next (Ctx, RFLX.Pub_Sub.Message.f_command)
```

To actually use the generated message, we first need to retrieve the pointer to the message buffer from the context. When calling *Initialize*, the pointer is stored inside the context and SPARKs borrow checker ensures that it cannot be accessed outside the context anymore. To retrieve it, the *Take_Buffer* procedure can be used. After that, the message buffer may be passed to our *Socket.Send* procedure. Note, that after taking the buffer, the context will essentially be only usable to retrieve scalar values. The SPARK contracts ensure that no fields are set, and no value of a non-scalar field is retrieved, before the context is reinitialized.

Finally, the memory allocated in the *Send_Status* procedure needs to be freed. RecordFlux generates a *Free* procedure

for the built-in type *Bytes_Ptr*. Again, SPARK ensures that no memory leak can occur. If we forgot to call *Free*, an error would be generated:

```
broker.adb:24:07: medium: resource or memory leak might occur at end of scope
```

With a way to send status messages easily, we can look into the implementation of our main subprogram *Run*:

```
procedure Run is
   Context : RFLX.Pub_Sub.Message.Context;
   Buffer : RFLX.RFLX_Types.Bytes_Ptr :=
      new RFLX.RFLX_Types.Bytes'(1 .. 4_096 => 0);
   Success : Boolean;

   use type RFLX.Pub_Sub.Length;
begin
   Socket.Receive (Buffer.all, Success);
   if not Success then
      RFLX.RFLX_Types.Free (Buffer);
      return;
   end if;

   RFLX.Pub_Sub.Message.Initialize
      (Context,
       Buffer,
       RFLX.RFLX_Types.To_Last_Bit_Index (Buffer'Last));
   -- ...
end Run;
```

Like before, we have a context variable and a pointer to the buffer which is used to receive a message from the network. We also initialize the context using the *Initialize* procedure. The difference here, is the third parameter (*Written_Last*), which we pass to *Initialize*. It determines the last bit of the message. When we generated a message, the default value *0* was used to indicate that we started with an empty message which got populated by the *Set_<FieldName>* procedures. When parsing the received data, we have to use the end of the message instead. The *To_Last_Bit_Index* function can be used to convert the byte size of the buffer into a bit index as required by *Initialize*.

Once we have initialized the context with the received data, we can call *Verify_Message* to perform the verification of the received message. The *Well_Formed_Message* function is then used to retrieve the result of this operation:

```
procedure Run is
   -- ...
begin
   -- ...
   RFLX.Pub_Sub.Message.Verify_Message (Context);
   if RFLX.Pub_Sub.Message.Well_Formed_Message (Context) then
      declare
         Id : RFLX.Pub_Sub.Identifier :=
            RFLX.Pub_Sub.Message.Get_Identifier (Context);
         Cmd : RFLX.Pub_Sub.Command :=
            RFLX.Pub_Sub.Message.Get_Command (Context);
      begin
         case Cmd is
            when RFLX.Pub_Sub.SUBSCRIBE =>
               DB.Subscribe (Id);
               Send_Status (Id, RFLX.Pub_Sub.SUCCESS);
```

(continues on next page)

(continued from previous page)

```
            when RFLX.Pub_Sub.UNSUBSCRIBE =>
               if DB.Is_Subscribed (Id) then
                  DB.Unsubscribe (Id);
                  Send_Status
                     (Id,
                      RFLX.Pub_Sub.SUCCESS);
               else
                  Send_Status
                     (Id,
                      RFLX.Pub_Sub.ERROR_NOT_SUBSCRIBED);
               end if;
            when RFLX.Pub_Sub.PUBLISH =>
               if DB.Is_Subscribed (Id) then
                  declare
                     Length : RFLX.Pub_Sub.Length
                        := RFLX.Pub_Sub.Message.Get_Length (Context);
                     Subscribers : DB.Identifiers
                        := DB.Current_Subscribers;
                  begin
                     if Subscribers'Length <= 1 then
                        Send_Status
                           (Id,
                            RFLX.Pub_Sub.ERROR_NO_SUBSCRIBERS);
                     elsif Length > 4_000 then
                        Send_Status
                           (Id,
                            RFLX.Pub_Sub.ERROR_MESSAGE_TOO_LONG);
                     else
                        RFLX.Pub_Sub.Message.Take_Buffer (Context, Buffer);
                        for Subscriber in Subscribers'Range loop
                           Socket.Send (Buffer.all);
                        end loop;
                     end if;
                  end;
               else
                  Send_Status
                     (Id,
                      RFLX.Pub_Sub.ERROR_NOT_SUBSCRIBED);
               end if;
            when others =>
               Send_Status
                  (Id,
                   RFLX.Pub_Sub.ERROR);
         end case;
      end;
   end if;
   -- ...
end Run;
```

If the message is well-formed, fields can be accessed using the generated *Get_<FieldName>* functions. Even for a well-formed message the SPARK contracts make sure that no missing fields are accessed. If our code tried to access the *Length* field in a received message where the *Command* field is not set to *PUBLISH*, this would be detected statically:

```
broker.adb:81:51: medium: precondition might fail
```

As before, we need to take the buffer out of the context and free it to avoid resource leaks. As we may have forwarded a publish message, we need to check whether the buffer is still in the context. This can be done by *Has_Buffer* function:

```
procedure Run is
   -- ...
begin
   -- ...
   if RFLX.Pub_Sub.Message.Has_Buffer (Context) then
      RFLX.Pub_Sub.Message.Take_Buffer (Context, Buffer);
   end if;
   RFLX.RFLX_Types.Free (Buffer);
end Run;
```

A very simple main procedure which initializes the networking library and calls *Run* in a loop concludes our example:

```
with Socket;
with Broker;

procedure Main is
begin
   Socket.Initialize (Port => 8_888);
   loop
      Broker.Run;
   end loop;
end Main;
```

Proving and Running the Result

To prove our program using *gnatprove*, we need to extend our project file to configure the proof process:

```
project Pub_Sub is
   - ...
   package Prove is
      for Proof_Switches ("Ada") use (
         "-j0",
         "--prover=z3,cvc5,altergo,colibri",
         "--steps=0",
         "--timeout=180",
         "--memlimit=2000",
         "--checks-as-errors=on",
         "--warnings=error",
         "--counterexamples=off"
      );
   end Prove;
   for Main use ("main.adb");
end Pub_Sub;
```

Depending on the complexity of the message specifications, code generated by RecordFlux will require significant time and resources to prove successfully. The *-j0* option instructs *gnatprove* to use all available CPU cores for parallel proof. The more cores are available, the better. The number of cores used can be limited by choosing a number greater than 0. A non-standard order of provers is selected using the *--provers* switch. Different provers or a different order may work as well, but the above example shows the combination we have found to be most effective. We also found it

helpful to disable the step limit (*--steps=0*) as the default limit is too low and the steps required by different solvers vary greatly. Disabling counter examples reduces the proof overhead and is advisable for the generated code. If proofs fail for generated code, adapting *--timeout* or *--memlimit* may be necessary. For details on the options, consult the SPARK User's Guide.

With the proof options in place, we can run *gnatprove* on the code:

```
$ gnatprove -P pub_sub.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in obj/gnatprove/gnatprove.out
```

Finally, we can build and run the example:

```
$ gprbuild -P pub_sub.gpr
Compile
[Ada] main.adb
[Ada] broker.adb
[Ada] socket.adb
[Ada] db.adb
Bind
[gprbind] main.bexch
[Ada] main.ali
Link
[link] main.adb

$ ./obj/main
```

3 Appendix

3.1 Command Line Options

The following command line options are available for the *rflx* command:

```
usage: rflx [-h] [-q] [--version] [--no-caching] [--no-verification]
            [--max-errors NUM] [--workers NUM] [--unsafe] [--legacy-errors]
            {check,generate,optimize,graph,validate,install,convert,run_ls,doc}
            ...

positional arguments:
  {check,generate,optimize,graph,validate,install,convert,run_ls,doc}
    check               check specification
    generate            generate code
    optimize            optimize generated state machine code
    graph               generate graphs
    validate            validate specification against a set of known valid or
                        invalid messages
    install             set up RecordFlux IDE integration
    convert             convert foreign specifications into RecordFlux
                        specifications
    run_ls              run language server
    doc                 open documentation

```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  -q, --quiet           disable logging to standard output
  --version
  --no-caching          ignore verification cache
  --no-verification     skip time-consuming verification of model
  --max-errors NUM      exit after at most NUM errors
  --workers NUM         parallelize proofs among NUM workers (default: NPROC)
  --unsafe              allow unsafe options (WARNING: may lead to erronous
                        behavior)
  --legacy-errors       use old error message format
```

The *rflx check* subcommand has the following options:

```
usage: rflx check [-h] SPECIFICATION_FILE [SPECIFICATION_FILE ...]

positional arguments:
  SPECIFICATION_FILE  specification file

options:
  -h, --help          show this help message and exit
```

The *rflx generate* subcommand has the following options:

```
usage: rflx generate [-h] [-p PREFIX] [-n] [-d OUTPUT_DIRECTORY]
                     [--debug {built-in,external}]
                     [--ignore-unsupported-checksum]
                     [--integration-files-dir INTEGRATION_FILES_DIR]
                     [--reproducible]
                     [SPECIFICATION_FILE ...]

positional arguments:
  SPECIFICATION_FILE    specification file

options:
  -h, --help            show this help message and exit
  -p PREFIX, --prefix PREFIX
                        add prefix to generated packages (default: RFLX)
  -n, --no-library      omit generating library files
  -d OUTPUT_DIRECTORY   output directory
  --debug {built-in,external}
                        enable adding of debug output to generated code
  --ignore-unsupported-checksum
                        ignore checksum aspects during code generation
  --integration-files-dir INTEGRATION_FILES_DIR
                        directory for the .rfi files
  --reproducible        ensure reproducible output
```

The *rflx graph* subcommand has the following options:

```
usage: rflx graph [-h] [-f {dot,jpg,pdf,png,raw,svg}] [-i REGEX]
                  [-d OUTPUT_DIRECTORY]
                  SPECIFICATION_FILE [SPECIFICATION_FILE ...]
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  SPECIFICATION_FILE    specification file

options:
  -h, --help            show this help message and exit
  -f {dot,jpg,pdf,png,raw,svg}, --format {dot,jpg,pdf,png,raw,svg}
                        output format (default: svg)
  -i REGEX, --ignore REGEX
                        ignore states with names matching regular expression
  -d OUTPUT_DIRECTORY   output directory
```

The *rflx validate* subcommand has the following options:

```
usage: rflx validate [-h] [--split-disjunctions] [-v VALID_SAMPLE_PATH]
                     [-i INVALID_SAMPLE_PATH] [-c CHECKSUM_MODULE]
                     [-o OUTPUT_FILE] [--abort-on-error] [--coverage]
                     [--target-coverage PERCENTAGE]
                     SPECIFICATION_FILE MESSAGE_IDENTIFIER

positional arguments:
  SPECIFICATION_FILE    specification file
  MESSAGE_IDENTIFIER    identifier of the top-level message (e.g.,
                        Package::Message)

options:
  -h, --help            show this help message and exit
  --split-disjunctions  split disjunctions before model validation (may have
                        severe performance impact)
  -v VALID_SAMPLE_PATH  known valid sample file or directory
  -i INVALID_SAMPLE_PATH
                        known invalid sample file or directory
  -c CHECKSUM_MODULE    name of the module containing the checksum functions
  -o OUTPUT_FILE        path to output file for validation report in JSON
                        format (file must not exist)
  --abort-on-error      abort with exitcode 1 if a message is classified as a
                        false positive or false negative
  --coverage            enable coverage calculation and print the combined
                        link coverage of all provided messages
  --target-coverage PERCENTAGE
                        abort with exitcode 1 if the coverage threshold is not
                        reached
```

The *rflx install* subcommand has the following options:

```
usage: rflx install [-h] [--gnat-studio-dir GNAT_STUDIO_DIR]
                    {gnatstudio,vscode,nvim,vim}

positional arguments:
  {gnatstudio,vscode,nvim,vim}

options:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  --gnat-studio-dir GNAT_STUDIO_DIR
                        path to the GNAT Studio settings directory (default:
                        $HOME/.gnatstudio)
```

The *rflx convert* subcommand has the following options:

```
usage: rflx convert [-h] [--reproducible] {iana} ...

positional arguments:
  {iana}
    iana          convert IANA registry into RecordFlux specifications

options:
  -h, --help      show this help message and exit
  --reproducible  ensure reproducible output
```

The *rflx run_ls* subcommand has the following options:

```
usage: rflx run_ls [-h]

options:
  -h, --help  show this help message and exit
```

The *rflx doc* subcommand has the following options:

```
usage: rflx doc [-h] {lrm,ug}

positional arguments:
  {lrm,ug}    documentation to open: Language Reference Manual (lrm), User's
              Guide (ug)

options:
  -h, --help  show this help message and exit
```

3.2 Specification Files

Style Checks

By default, the style of specification files is checked. Error messages about style violations have a style check identifier appended to the message in the following form "[style:<identifier>]". For example: "[style:line-length]". Style checks can be disabled for individual files by adding a pragma to the first line of the file.

Example

```
-- style: disable = line-length, blank-lines

package P is

end P;
```

It is also possible to disable all style checks by using the identifier `all`.

Example

```
-- style: disable = all

package P is

end P;
```

Integration Files

For each RecordFlux specification file with the `.rflx` file extension, users may provide a file with the same name but the `.rfi` file extension. This is useful to specify buffer sizes for state machines. This file is in the YAML data format. Buffer sizes are provided in bytes. If no such file is provided, RecordFlux uses a default buffer size of 4096 bytes.

Integration file structure

The following example of an integration file defines, for the state machine `My_State_Machine`, a default buffer size of 4096 bytes, a buffer size of 2048 bytes for the global variable `My_Global_Var`, and a buffer size of 1024 bytes for the variable `My_State_Variable` defined in the state `My_State`.

```
Machine:
  My_State_Machine:
    Buffer_Size:
      Default: 4096
      Global:
        My_Global_Var: 2048
      Local:
        My_State:
          My_State_Variable: 1024
```

3.3 External IO Buffers

By default, all message buffers of a state machine are stored inside the state machine's `Context` type. The `Read` and `Write` primitives give access to a message buffer when the state machine reaches an IO state. An alternative is to use externally defined buffers. This approach can save memory and prevent copy operations, but it removes some safety guarantees (see the notes at the end of the section).

External IO buffers can be enabled for a state machine by setting the `External_IO_Buffers` option in the integration file.

```
Machine:
  My_State_Machine:
    External_IO_Buffers: True
```

If external IO buffers are enabled, the buffer for all messages that are accessed in any IO state must be allocated externally and provided during the initialization of the state machine context as arguments to the `Initialize` procedure. When an IO state is reached, the corresponding buffer of the accessed message can be removed from the state machine context. The buffer must be added again before the state machine can be continued.

```
if Buffer_Accessible (Next_State (Ctx), B_Request) then
   Remove_Buffer (Ctx, B_Request, Request_Buffer);

   ...

   Add_Buffer (Ctx, B_Request, Request_Buffer, Written_Last (Ctx, B_Request));
end if;
```

(continues on next page)

(continued from previous page)

```
Run (Ctx);
```

Before a buffer can be removed or added, it must be checked that the buffer is accessible. The buffer is accessible if the next state of the state machine will perform either a read or write operation on the given buffer. The function `Buffer_Accessible` can be used to check this condition. If all read or write operations on a given channel are used with a single message buffer, then the function `Needs_Data` or `Has_Data`, respectively, is sufficient to determine this condition. Each external buffer is uniquely identified by an enumeration literal of the `External_Buffer` type. In the example shown above, `B_Request` identifies the buffer of a message called `Request`. When adding a buffer, the index of the last written byte has to be provided. If the buffer has not been changed, `Written_Last` can be used to set it to the same value as before.

CAUTION: The message buffer must only be modified if the state machine expects data, i.e., `Needs_Data` is true. Changing the contents of the buffer at other times may cause the state machine to behave unexpectedly. This property cannot be guaranteed by the generated SPARK contracts.

3.4 Reporting Errors

Please report issues on GitHub:

https://github.com/AdaCore/RecordFlux/issues/new?labels=bug

If a tool invocation produces a bug box, please include its complete content and all input files in the report.

As an AdaCore customer, please open a ticket in GNAT Tracker.

3.5 Background

More information about the theoretical background of RecordFlux can be found in our paper:

> Reiher T., Senier A., Castrillon J., Strufe T. (2020) RecordFlux: Formal Message Specification and Generation of Verifiable Binary Parsers. In: Arbab F., Jongmans SS. (eds) Formal Aspects of Component Software. FACS 2019. Lecture Notes in Computer Science, vol 12018. Springer, Cham (paper, preprint)

3.6 GNU FDL

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document 'free' in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of 'copyleft', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this

License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The 'Document', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as 'you'.

A 'Modified Version' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A 'Secondary Section' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The 'Invariant Sections' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The 'Cover Texts' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A 'Transparent' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not 'Transparent' is called 'Opaque'.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The 'Title Page' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, 'Title Page' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled 'History', and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled 'History' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on.

These may be placed in the 'History' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled 'Acknowledgements' or 'Dedications', preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled 'Endorsements'. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as 'Endorsements' or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled 'Endorsements', provided it contains nothing but endorsements of your Modified Version by various parties -- for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled 'History' in the various original documents, forming one section entitled 'History'; likewise combine any sections entitled 'Acknowledgements', and any sections entitled 'Dedications'. You must delete all sections entitled 'Endorsements.'

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an 'aggregate', and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License 'or any later version' applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.

Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled 'GNU
Free Documentation License'.
```

If you have no Invariant Sections, write 'with no Invariant Sections' instead of saying which ones are invariant. If you have no Front-Cover Texts, write 'no Front-Cover Texts' instead of 'Front-Cover Texts being LIST'; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.