IrpTransmogrifier: Parser for IRP notation protocols, with rendering, code generation, and recognition applications.

Table of contents

Note:

Possibly you should not read this document! If your are looking for a user friendly GUI program for generating, decoding, and analyzing IR signals etc, please try the GUI program IrScrutinizer, and get back here if (and only if) you want to know the details on IR signal generation and analysis.

Date	Description
2019-08-06	Initial version, for IrpTransmogrifier Version 1.0.0.
2019-08-11	Minor tweaks for version 1.0.1.
2019-12-27	Update for version 1.2.4.
2020-05-19	Update for version 1.2.6. More of an improvement and clarification than a description of new features.
2020-06-27	Update for version 1.2.7: IRP database maintenance.
2021-06-03	Update for version 1.2.10: Minor improvements.

Table 1: Revision history

1 Release notes

Release notes for the current version

2 Introduction

The *IRP notation* is a domain specific language for describing IR protocols, i.e. ways of mapping a number of parameters to infrared signals. It is a very powerful, slightly cryptic, way of describing IR protocols, that was first developed by John Fine in 2003. In early 2010, Graham Dixon (mathdon in the JP1-Forum) wrote a specification.

This project contains a parser, a number of basic classes, and utilities for IRP protocols. It does not contain a graphical user interface (GUI). See Main principles for a background. Instead, the accompanying program IrScrutinizer provides GUI access to most of the functionality described here.

This is a new program, written from scratch, that is intended to replace IrpMaster, DecodeIR, and ExchangeIR, and much more, like potentially replacing hand written decoders/renders. The project consists of an API library that is also callable from the command line as a command line program.

The name indicates that the program transforms ("transmogrifies") IRP form protocols to and from all sort of different things.

For understanding this document and program, a basic understanding of IR protocol is assumed. However, the program can be successfully used just by understanding that an "IRP protocol" is a "program" in a particular domain specific language for turning a

number of parameters into an IR signal, and the present program is a compiler/interpreter/decompiler of that language. Some parts of this document requires more IRP knowledge, however.

2.1 Background

This program can be considered as a successor of IrpMaster. The IRP parser therein is based upon a ANTLR, using the now obsolete version 3. The "new version" ANTLR4 is really not a new version, but a completely different tool, fixing most of the quirks that were irritating in IrpMaster (like the "ugliness" of embedding actions within the grammar and no left recursion). Unfortunately, this means that using version 4 instead of version 3 is not like updating a compiler or such, but necessitates a complete rewrite of the grammar and the actions.

It turned out that IrScrutinizer version 1 (i.e. the version based upon IrpMaster for generating and DecodeIR for decoding) had a fundamental problem: Although DecodeIR and IrpMaster (the latter through the data base `IrpProtocols.ini`) agree on most (but not all) protocols, they relied upon two different, dis-coupled sources of protocol information. If the data base for the sending protocols were changed or extended, this affected sending only, while decoding was not changed, and potentially conflicted with the sending protocols. Also, both DecodeIR and the IR Analyzer, used in IrScrutinizer versions before 1.3, have shown to be essentially impossible to maintain and extend.

2.2 Dependencies

The program is written entirely in Java, using no native libraries. Therefore, it runs on anything for which Java (presently version 8) is available. Except for building (using Maven and a number of its plugins), testing of the program (using TestNG), and of course the Java framework, it depends on ANTLR4, StringTemplate, and the command line decoder JCommander.

In order to avoid circular dependencies, it is prohibited to use any other Harctoolbox software.

2.3 Documentation principles

The role of program documentation has changed considerably the last few decades. Ideally, a program does not need any documentation at all, if its operation is entirely obvious. Unfortunately, as non-trivial problems almost never have trivial solutions, so some sort of documentation is almost always necessary for programs solving complex programs. The next best thing is that the program contains its own documentation, for GUI programs through tool-tips, info-popups etc, for command line programs through help texts. This can contain precise information over the syntax and semantic of, for example, commands. Since they are integrated in the program and maintained together with the program source, it is less likely that it lacks behind the actual program behavior,

than in the traditional program manual like this. They may also change faster than reference manual.

Documentation for API, and structures like XML Schemas should preferably be integrated into the source itself, using technologies like Javadoc, Doxygen, and, for XML documents, including Schemas etc, be documented using the available documentation elements.

Separate program documentation, on paper or electronically (like the present document), should then play another role than in the last century. It should be centered around explaining the concepts of the program, the "why", and not the "how". Some details that cannot in a natural way be explained within the program with a few sentences can also be covered.

The present document is written in that spirit. We do not explain all commands and parameters, at least not the ones that are (more-or-less) obvious. Also, since the program changes more often than the manual, it is written to be a "nice" document, not to exactly document a particular version of the program.

Since a text like this is unlikely to be read sequentially from start to finish, redundancy and repetitions are not considered evil and a sign of stylistic inaptitude, like in other type of documents.

All sort of suggestions for improving the documentation are always welcome. Also, reports of useless or misleading error messages, as well as suggestion on improving them, are solicited.

One or more tutorial Youtube videos are planned, explaining the program and its principles more informally.

2.4 Acknowledgement

I would like to acknowledge the influence of the JP1 forum, both the programs (in particular of course DecodeIR, and the discussions (in particular with Dave Reed ("3FG") and Graham Dixon ("mathdon")). This work surely would not exist without the JP1 forum.

2.5 Copyright and License

The program, as well as this document, is copyright by myself. Of course, it is based upon the IRP specification, but is to be considered original work. The "database file" `IrpProtocols.xml` is derived from many sources, in particular DecodeIR.html (which is public domain), so I do not claim copyright, but place it in the public domain.

The program uses some third-party project. It depends on ANTLR4 (license), and Stringtemplate (license) by Terence Parr. It also uses the command argumend decoder JCommander by Cédric Beust. This is free software released under the Apache 2.0 license.

The program and its documentation are licensed under the GNU General Public License version 3, making everyone free to use, study, improve, etc., under certain conditions.

Should the wish arise to use the program in a way not allowed by the GPL, please contact me for discussing a dual license agreement.

Data and code generated by the program can be used without any copyright restrictions.

3 Main principles

3.1 Design principles

It is easier and more logical to put a GUI on top of a sane API, then to try to extract API functionality from a program that was never designed for this but built around and into the GUI. The goal of *this* project is an API and a command line program.

3.1.1 Performance considerations

Performance consideration, both time and space, were given secondary priorities. Also, the decoding mechanism is intrinsically much slower that DecodeIR. A single decode can take several hundred milli-seconds. However, in normal interactive use from the command line or through IrScrutinizer, this does not introduced any noticeable delays.

3.1.2 IR signal formats

The current program reads raw format, and Pronto Hex format IR signals. These formats can also be output. Although many more formats exist, it is not planned to extend this. The "conversion expert" is IrScrutinizer (which is even symbolized by its icon, the Babel fish).

3.1.3 Data types

Everything that is a physical quantity (durations and frequency), are real numbers (in the Java code `double`),

Durations are always given in micro seconds. Unless in a context where the IRP says otherwise, all others quantities are given in (pure) SI units without prefix. So are duty cycle and relative tolerance both a real number between 0 and 1, not a number of percents. Modulation frequency is given in Hz, not it kHz (with the exception of the `GeneralSpec`, since this is defined in the specification).

Integer literals can be given in base 16 (using prefix "0x"), base 8 (using prefix "0"), base 2 (using prefix "0b"), as well as in base 10 (no prefix, omitting leading zeros).

Integer quantities are in principle arbitrarily large, but in most cases limited to 63 bits, since the implementation uses Java's `long`, 64 bits long, amounting to 63 bits plus sign. (Work is ongoing to remove this restriction, by, alternatively using Java's `BigInteger`.)

3.1.4 Internationalization

Being a command line program and API library, this project is not a a candidate for internationalization.

4 Theory and general concepts

4.1 Main concepts

4.1.1 IrSequence

Sequence of time durations, in general expressed in microseconds, together with a modulation frequency. The even numbered entries normally denote times when the IR light is on (disregarding modulation), called "flashes" or sometimes "marks", the other denote off-periods, "gaps" or "spaces". They always start with a flash, and end with a gap.

> **Note:**
> In some communities (Lirc and Linux, IRremote), the ending gap is not considered to be a part of the IrSequence/IrSignal. This must be taken into consideration if importing signals from these communities.

4.1.2 IrSignal

Consists of three IR sequences, called

1. *start sequence* (or "intro", or "beginning sequence"), sent exactly once at the beginning of the transmission of the IR signal,
2. *repeat sequence*, sent "while the button is held down", i.e. zero or more times during the transmission of the IR signal (although some protocols may require at least one copy to be transmitted),
3. *ending sequence*, sent exactly once at the end of the transmission of the IR signal, "when the button has been released". Only present in a few protocols.

Any of these can be empty, but not both the intro and the repeat. A non-empty ending sequence is only meaningful with a non-empty repeat.

By "sending an IrSignal *n* times" we shall mean sending an IrSequence consisting of one copy of the intro sequence, *n* - 1 copies of the repeat sequence, and one copy of the ending sequence, unless the intro sequence is empty, in which case the IrSequence has *n* repeats, followed by the ending sequence.

4.1.3 Numerical equality between durations

Two durations *d1* and *d2* are considered numerically equal, if *either* the difference is absolute less or equal than `absolutetolerance` *or* the difference divided by the largest is less than or equal than `relativetolerance`. (see `IrCoreUtils.approximatelyEquals`).

4.1.4 Parsing of IR signals as text

We need a new simple text format for raw IR signals:

4.1.4.1 Bracketed raw format

Simple text format for formatting of a raw IR signal, introduced with the current program. Intro-, repeat, and (optionally) ending sequences are enclosed by brackets ("[]", sometimes called "square brackets" or "square parenthesis"). Signs ("+" for flashes, "-" for gaps), as well as interleaved commands or semicolons, are accepted, but ignored (even if they are not interleaved). Optionally, the beginning sequence can be preceeded by `Freq=`*frequency_in_Hz*`Hz` giving the modulation frequency. If not explicitly given, a default value (dependent on the reading program, but in general 38000Hz) is assumed.

Example (an RC5-signal):

```
Freq=36000Hz[][+889,-889,+1778,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,
+889,-889,+889,-889,+889,-889,+889,-889,+889,-889,+889,-90886][]
```

Although we only consider the Pronto Hex and the raw form, parsing of text representation is a non-trivial task. IrpTransmogrifier presently contains three different parsers, tried in order until one succeeds. These are classes implementing the interface `IrSignalParser`, namely:

ProntoParser
Tries to interpret its argument as Pronto Hex, possibly after concatenating all lines of its argument.

BracketedIrSignalParser
Tries to parse an IrSignal in the bracketed text format, as defined above.

MultilineIrSignalParser
If the number of lines is 1, 2, or 3, these are considered intro-, repeat-, and ending sequence respectively. If there are more lines, these are concatenated, and considered an intro-sequence, without repeat-, and ending sequence.

Programs can add other formats and their corresponding parsers using the API. For an example, in IrScrutinizer, a parser for the Global Caché sendir format is added, see `org.harctoolbox.irscrutinizer.Interpretstring.interpretString()`.

4.1.5 Cleaner

Physically measured IR signals/sequences are in general "dirty", consisting of a number of close, but not quite equal, measured durations. To be useful for further processing, the signal/sequence needs to be cleaned. The naive way to implementing such a cleaning algorithm is to round off all durations to a multiple of an (more ore less arbitrarily selected) time unit. The cleaner in IrpTransmogrifier takes a more sophisticatedly approach: The collected durations found in the sequence(s) are bundled into "bins" (disjoint intervals), according to this closeness concept. Every duration belonging to a bin is "close" (determined by those parameters) to the bin middle. All the

durations within the bin are then replaced by the average of its members. It is thus not guaranteed that the distance between a duration and its replacement will be consistent with `absolutetolerance` and `relativetolerance`.

If invoked on a number of IR signals/sequences, the histogram is formed over all sequences, and the corresponding "cleaning" then applied to the individual signals/sequences.

The cleaner is not a separate function in the command line program, but can be invoked together with the `decode` and `analyze` commands.

4.1.6 Repeat finder

A repeat finder is a program that, when fed with an IrSequence, tries to identify a repeating subsequence in it, returning an IrSignal containing intro-, repeat-, and ending sequence, compatible with the given input data. IrpTransmogrifier can optionally run its data for the decode and analyze command though its built-in repeat finder. The repeat finder is configurable using the following parameters:

`absolutetolerance`
See Numerical equality between durations. Current default: 100.0.

`relativetolerance`
See Numerical equality between durations. Current default: 0.3.

`minrepeatgap`
To be recognized as a repeat sequence, the final gap must be at least this long. Default: 5000.0.

> Note:
> In some cases, the repeat finder is doing a "too good" job and may squash a signal so that the decoder does not produce the expected result. For example, a `JVC` signal is reduced beyond the `JVC` IRP. (To handle this case, the relaxed protocol `JVC_squashed` was entered into the IRP data base.)

4.1.7 The Configuration file/IRP protocol database

The configuration file `IrpProtocols.ini` of IrpMaster has been replaced by an XML file, called `IrpProtocols.xml`. The XML format is defined by a W3C schema, named irp-protocols, having the XML name space `http://www.harctoolbox.org/irp-protocols`. This format has many advantages over the previous, simpler, format, as it gives access to different XML technologies, for example for formatting and transforming. It can contain embedded (X)HTLM fragments, useful for for writing documentation fragments. The file is to be thought of as a data base of protocols and their properties and parameters, not as a configuration file for the present program. It can contain different parameters that can be used by different programs, for example, tolerance parameters for decoding. For this, arbitrary string-valued parameters are permitted. It is up to an interpreting program to determine the semantic. Within IrpTransmogrifier, this is used for providing protocol-specific values to the parameters.

If setting the attribute `type="xml"`, the element's content is considered an XML document fragment, i.e., arbitrary (well-formed) xml content can be present.

There is also an XSLT stylesheet `IrpProtocols2html.xsl`, which translates this file to readable HTML, for reading in a HTML browser. This makes it possible to browse the said file in an XSLT-enabled browser just by opening it. Unfortunately, this very practical mechanism is since recently considered a security problem. Firefox (and possibly other browsers) therefore unfortunately disables it for local files (URIs using `file:` scheme).)

More than one protocol data base file ("`IrpProtocols.xml`") can be deployed. This gives a user or a program the possibility to strictly divide his/her/its own entries from the official ones. A "small" file modifying and/or adding a few protocols is often called a "patch file", although there is actually no difference between the first and subsequent files, and the semantics is identical. All files are required to be valid XML, and should be valid with respect to the given XML schema.

From the command line, the option `--configfile` can be given several times, or several files can be given as argument, separated by commas (","). From the API, see the function `IrpDatabase.patch(File)` (and others).

The content of the patch file is basically merged into the data base, amending the information already there, with the exception that an empty entry deletes the original one. The exact rules are as follows:

- If a patch file contains an empty protocol element, the protocol with the same name will be (if present) removed from the data base. Otherwise, its content is amended into the present one.
- For the protocol properties (both the XML properties and the normal ones) similar rules apply: An empty property element removes that property from the protocol. A non-empty property in a protocol in the patch file is added to the end of the list that is the value of that property in the protocol (unless it is already present).
- However, the "properties" irp and documentation are different, since there can only be one of those in a protocol. For these, an entry in the patch file overwrites the original entry.

Syntax and semantics of the file is believed to be essentially self explaing. The exact syntax is given by the schema, and is therefore not repeated here.

Note that the program contains some functions for the maintenance of the data base.

4.1.7.1 Protocol parameters

Most protocol parameters are given as the `parameter` element. However, the protocol name, its IRP, and its documentation are handled differently:

`name`

The name of the protocol. It is specified in the mandatory attribute `name` in the `protocol` element. The name is folded to lowercase for searches and comparisons, which are therefore done case insensitively.

`irp`
The IRP form of the protocol, as text.

`documentation`
Also a separate, but optional, element. Any textual information can be put here. Arbitrary (valid) (X)HTML code can be included here, for example formatting instructions, tables, or links. A processing program may select to render the HTML content, pass it through, or just to ignore it. (For example the current version of the command line program will, using the command option `list --documentation` produce an "dumb" version, while `list --html` will give the original text, enclosed in a HTML `div` element.

The following parameters may be given for any protocol in the data base. They override the global values for the current protocol.

`absolutetolerance`
See Numerical equality between durations. Current default: 100.0.

`relativetolerance`
See Numerical equality between durations. Current default: 0.3.

`frequency-tolerance`
Tolerance in Hz for frequency comparisons. Set to -1 to disable frequency check. Current default: 2000.0.

`minimum-leadout`
Minimal duration in micro seconds that is accepted as final duration. Current default: 20000.0.

`prefer-over`
If a signal has multiple decodes, the present protocol is preferred over the one mentioned as `prefer-over`. May be given several times. (Normally, a special protol should be preferred over a more general one.)

`alt_name`
Alternative name, ("alias", "synonym") for the present protocol.

`reject-repeatless`
When decoding, normally the repeat sequence may match 0 times, i.e. not at all. If this parameter is `true` at least one repeat must be present for a match to be recognized. (Strictly speaking, this would have been possible to achieve by using "+" instead of "*" for the repeat indicator, but this would have other disadvantages.)

`decodable`
Setting this to `false` prohibits the program from trying to use this protocol for decoding. Normally, this is only used for portocols that are so involved that protocol decoding is impossible or not feasible.

`decode-only`

Setting this to `true` makes the program refuse to render the protocol. Should be used only for "protocols" that denote incomplete or otherwise flawed decodes ("relaxed" protocols), for example with missing parts or non-matching checksums.

Other parameters are allowed, but ignored by IrpTransmogrifier. They may be used by another program. Also, new parameters may be introduced in the future.

Some more commands and hints are given as comments in the file itself.

5 Use cases

In this section, we will discuss the different use cases from a high-level, theoretical standpoint, without delving into the usage of the program. The subsequent section Subcommands, which to a certain degree mirrors the present, will cover the usage of the program.

5.1 Rendering

Given a protocol name, present in the protocol data base (alternatively, an IRP protocol given explicitly with the `--irp` option), and a set of valid parameter values, an IrSignal is computed. This use case corresponds to IrpMaster (or MakeHex).

In earlier versions of IrScrutinizer, the word "generate" was used instead of "render". These words can be considered as synonyms.

5.2 Decoding

5.2.1 General

This use case corresponds to DecodeIR: given a numerical IR signal/sequence, find one (or more) parameter/protocol combination(s) that could have generated the given signal. This is implemented by trying to parse the given signal with respect to a number of candidate protocols, per default all. It is thus very robust, systematic, and customizable, but comparatively slow.

While every conforming IRP form with only one repeat also can be usable for rendering, the same is unfortunately not true for decoding. A few protocols cannot be used for decoding. Non-recognizable protocols are marked by setting the `decodable` parameter to `false`. To be useful for decoding, the IRP protocol should adhere to some additional rules:

- Non-deterministic grammars (like, for example, "A? A*") must be avoided.
- The "+" form of repetitions is discouraged in favor of the "*" form.
- The width and shift of a Bitfield must be constant (i.e. not dependent on protocol parameter).
- The decoder is capable of *simple* equation solving (e.g. `Arctech`), but not of complicated equation solving (e.g. `Fujitsu_Aircon_old`).

Presently all but the protocols `zenith`, `nec1-shirrif`, (non-constant bitfield width); `RTI_Relay_alt`, `fujitsu_aircon_old` (would require non-trivial equation solving)) are decodable. (Note how the non-decodable protocols `RTI_Relay_alt` and `fujitsu_aircond_old` was made decodable (`RTI_Relay` and `fujitsu_aircon`) by a changed parameterization.)

5.2.2 Multiple decodes

In many cases, a signal produces more than one decode. This is not necessarily an error nor a deficiency of the decoder. However, many protocols are effectively a special case of another protocol, for example, an signal that decodes to the Apple protocol by mathematical necessity is also a valid NEC1-f16 signal. It thus makes sense for the decoder to have an Apple decode to "override" a NEC1-f16 decode. This is specified by the `prefer-over` parameter. With this mechanism, the preferences are under control of the configuration file, not hard soldered into the program code as in DecodeIR.

5.2.3 Loose matches, "Guessing"

Many captured signals do not quite match the protocol they are supposed to match. However, in order to give the user of a device maximal comfort, the firmware in a receiving device is in general quite "forgiving", and accepts slightly flawed signals. The degree of "forgiveness" should be balanced against the possibility of "false positives": that the device erroneously considers "noise" of some form (often commands for a different device) as one of its commands. It is thus desirable for a program of this type to find a near match, "guess", when an proper match fails.

Generally speaking, it is my principle to "make everything as simple as possible, *but not simpler*". Sometimes a signal has several decodes, or none at all. Sometimes only a part of the data is used for the match. In these cases, the program always prefer to tell the truth (and the full truth) to the user, instead of, in the name of user friendlyness, to perform questionable simplifications. Unfortunately, a badly informed user tend to prefer a program delivering exactly one answer to every question over a program presenting the full truth.

There are a few different issues:

5.2.3.1 Matching of durations, non-leadouts

Non-ending durations are matched according to the numeric equality criterion, using the parameters `absolutetolerance` and `relativetolerance`.

5.2.3.2 Matching of ending durations

The ending duration has a slightly different role. When capturing, the ending duration is the time you have verified that nothing more is coming, not a real measurement. (Some "communities", like Lirc and IrRemote, do not consider an ending duration at all.) For this reason, it does not make sense to check the ending duration the same way as the

other, non-ending durations. There is instead a parameter `minimum-leadout` (possibly protocol dependent); the ending duration is considered as passed if it is longer or equal to `minimum-leadout`.

5.2.3.3 Matching of modulation frequency

There is a parameter `frequencytolerance`, which defaults to 2000. The frequency test is considered passed if the absolute difference between measured and expected frequency is less or equal to `frequencytolerance`. Setting `frequencytolerance` negative disables the frequency test, i.e., all values pass the test.

If an assymmetric interval is needed, instead the parameters `frequency-lower` and `frequency-upper` can instead be used, specifying the lowest and highest frequency that is to be accepted.

All of these parameters are to be given in Hz, not kHz.

It turns out that decoding of IrSignals and IrSequences are two fairly different use cases:

5.2.3.4 Matching of IR Signals (Intro-, Repeat and Ending sequence)

The task is to match an IR signal (with intro-, repeat- and ending sequence) to a protocol, turn out, "practically", to be more involved than just matching the different sequence to each other. The to-be-matched signal is often empirically measured, and its decomposition into sequences the result of entering a measured signal into a repeat finder. For a short measured sequence, the repeat part may not be identified as such. For a decoding program to be considered practically usable, this problem, and related problems, must be addressed and handled correctly.

For "strict" matching of a given IR signal, the intro-, repeat-, and ending sequences are required to match their theoretical counterparts (within numerical tolerances). If this fails, it may be sensible to convert the signal to an IrSequence (normally by concatenating the intro, repeat and ending sequences), and try to decode as IrSequence, as described in the next section.

A strict decode result of an IrSignal is a number (ideally exactly one) of decodes, each one containing a protocol and a set of corresponding parameters. (Technically, the function `Decoder.decodeIrSignal(IrSignal)` returns a `Decoder.SimpleDecodeSet`, being an `Iterable<Decoder.Decode>`).

5.2.3.5 Matching of IR Sequences

A completely faithful decoding an IR Sequence is theoretically a more complicated undertaking. Starting at position 0, in general several decodes can start there — although of course "normally" only one. Such a decode matches an intro sequence, zero or more repeat sequences, and the ending sequence. It thus matches a certain number of durations, less than or equal to the duration of the

input IrSequence. In the case that the decode is shorter than the input signal, the process repeats with the IrSequence that is remaining, leading to an exponential growth of decodes — at least in the general case. (Technically, the function `Decoder.decode(ModulatedIrSequence)` returns a `Decoder.DecodeTree`, which is an `Iterable<Decoder.TrunkDecodeTree>`, where each `Decoder.TrunkDecodeTree` consists of one decode (with a variable number of repeat-matches) ("trunk") followed by another `Decoder.DecodeTree`.)

5.3 Code generation for rendering and/or decoding

The task is: For a particular protocol and a particular target (C, C++, Java, Python,...), generate target code that can render or decode signals with the selected protocol. As opposed to the previous use cases, efficency (memory, execution time) (for the generated code) is potentially an issue.

Two mechanisms are available, XML and StringTemplate, described in the following two sections.

5.3.1 Code generation with XML

The program generates basically an XML version of `IrpProtocols.xml` (with the IRP protocol replaced by a much more "parsed" XML version). It is the task of the user to supply am XML transformation, for example using XSLT, that transfers the XML file to the expected target format. The program does not come with an XSLT engine, so this has to be invoked independently on the XML export. It is recommended to use XSLT version 2 for writing the transformation.

This is presently used for generating the Lirc export format (which is basically another XSLT transformation) of IrScrutinizer.

5.3.2 Code generation using StringTemplate

For code generation, the template engine StringTemplate can also be used. As opposed to the XML case, the program contains the transformation engine.

Target dependent code is not considered a part of this project, but is found in a separate project.

5.4 General code analysis

This use case is not really connected to parsing IRP, but fits in the general framework. This has been inspired by to the Analyzer and the RepeatFinder in Graham Dixon's ExchangeIR.

Theoretically speaking, this is an Inverse problem. Given one or many IR signals/ sequences, the problem is to find out what could have generated it, in the form of an IRP protocol. This problem in general does not have a unique solution; instead the "simplest" one is selected out of the possible solutions.

In a way, this is a more abstracted version of the decoding problem. There are around 10 different "template IRP" (for example, different forms of bi-phase and PWM modulation) that are tried. For every of those templates, a quantity, "weight" is computed, quantifying (in a somewhat arbitrary manner) how "complicated" the answer is. The template that produces the "simplest" answer, i.e. the one with the least weight, is selected.

The form of the final answer can be influenced by a number of different parameters, use the command `irptransmogrifier analyze --help` to list them.

5.5 IRP database maintenance

The command line program also contains some functions for maintenance of the IRP database, described next.

5.5.1 Validation

An IRP database is required to be valid with respect to the a W3C schema. For this, the common option `--validate` makes the program's XML parser read all files in validating mode, and stops if the input is not conforming.

5.5.2 Output of parsed database

The parsed database (possibly after merging of several files) can be output using the `list --dump` command. This option generate an XML file on the output. All other output is suppressed. This output can be, possibly after minimal hand editing (update version?) used as new `IrpProtocols.xml`.

The option `--xml` does the same as `--dump`, except that the XML comments in the original files are suppressed.

5.5.3 Other options

With the command `list --prefer-overs`, the protocol's `prefer-over`s are printed, transitively. The option `--check-sorted` checks the correct (alphabetic) sorting of the commands (with respect to their names). There are also a number of other options listing properties of the individual protocols, for example `--classify, --display, --warning, --weight`.

6 Extensions to, and deviation from, IRP semantic and syntax

This implementation of the IRP has a number of extensions, and a few deviations to the current specification version 2. These will be described in detail next.

For the complete syntax, see the ANTLR grammar.

6.1 Data types

While the original specification uses exclusively unsigned integers, here numbers that are intrinsically "physical" (modulation frequency, durations, duty cycle) are floating numbers, in the code `double`.

Integer numbers are in general implemented with Java's `long`, effectively limiting the number of bits to 63. In the future, it is planned to remove this restriction, using Java's BigInteger.

6.2 Repetitions

Possibly the major difficulty in turning the IRP Specification into programming code was how to make sense of its repetition concept. Most formalisms on IR signals (for example the Pronto format) considers an IR signal as an introduction sequence (corresponding to pressing a button on a remote control once), followed by a repeating signal, corresponding to holding down a repeating button. Any, but not both of these may be empty. In a few relatively rare cases, there is also an ending sequence, send after a repeating button has been released. Probably 99% of all IR signals fit into the intro/repetition scheme, allowing ending sequence in addition should leave very few practically used IR signals left. In "abstract" IRP notation, these are of the form A,(B)*,C with A, B, and C being (possibly empty) `bare irstream`s.

In contrast, the IRP notation reminds of they syntax and semantics of regular expressions: There may be any numbers of (infinite) repeats, and they can even be hierarchical (repetitions within repetitions). There does not appear to be a consensus on how this extremely general notation should be practically thought of as a generator of IR signals.

The predecessor program IrpMaster tried to be very smart here, by trying to implement all aspects, with the exception of hierarchical repetitions (repetitions within repetitions). This never turned out to be useful. The present program takes a simpler approach, by simply prohibiting multiple (infinite) repetitions.

6.3 Parameter Specifications

In the first, now obsolete, version 1 of the IRP notation the parameters of a protocol had to be declared with the allowed max- and min-value. This is not present in the current specification version. I have re-introduced this, using the name `parameter_spec`. For example, the well known NEC1 protocol, the Parameter Spec reads: `[D:0..255,S:0..255=255-D,F:0..255]`. (D, S, and F have the semantics of device, sub-device, and function or command number.) This defines the three variables D, S, and F, having the allowed domain the integers between 0 and 255 inclusive. D and F must be given, however, S has a default value that is used if the user does not supply a value. The software enforces that all values without default values are supplied, and within the stated limits. If, and only if, the parameter specs is incomplete, there may occur run-time errors concerning not assigned values. It is the duty of the IRP

author to ensure that all variables that are referenced within the main body of the IRP are defined either within the parameter specs, defined with "definitions" (Chapter 10 of the specification), or assigned in assignments before usage, otherwise a run-time error will occur (in the code, a `NameUnassignedException` will be thrown).

The preferred ordering of the parameters is: D, S (if present), F, T (if present), then the rest in alphabetical order,

The formal syntax is as follows, where the semantic of the '@' will be explained in a following section:

```
parameter_specs:
    '[' parameter_spec (',' parameter_spec )* ']'
    | '['  ']'

parameter_spec:
      name     ':' number '..' number ('=' expression)?
    | name '@' ':' number '..' number  '=' expression
```

6.4 GeneralSpec

For the implementation, I allow the four parts (three in the original specification) to be given in any order, if at all, but I do not disallow multiple occurrences — it is quite hard to implement cleanly and simply not worth it. (For example, ANTLR does not implement exclusions. The only language/grammar I know with that property is SGML, which is probably one of the reasons why it was considered so difficult (in comparison to XML) to write a complete parser.)

The default frequency is 38kHz, not 0kHz as in the specification. For the cases of modulation frequency not "really" known, but "sufficiently close" to 38kHz, it is recommended to rely on the default, not to state "38k" explicitly,

6.4.1 GeneralSpecs, duty cycle

Without any very good use case, I allow a duty cycle in percent to be given within the `GeneralSpec`, for example as `{37k,123,msb,33%}`. It is currently not used for anything, but preserved through the processing and can be retrieved using API-functions. If some, possibly future, hardware needs it, it is there.

6.5 Persistency of variables

In the specification and in forum contributions, all variables in a IRP description appear to be consider as intrinsically persistent: They do not need explicit initialization, if they are not, they are initialized to an undefined, random value. This may be a reasonable model for a particular physical remote control, however, from a scientific standpoint it is less attractive. In the current work, there is a way of denoting a variable, typically a toggle of some sort, as persistent by appending an "@" to its name in the parameter specs.

An initial value (with syntax as default value in the parameter spec) is here mandatory. For example, a toggle is typically declared as `[T@:0..1=0]` in the parameter specs. It is set to its initial value the first time the protocol is used. Rendering such a protocol typically updates the value, as given in an assignment, a 0-1 toggle goes like T=1-T). As opposed to variables that has not been declared as persistent, it retains its value between the invocations.

An instance of the `Protocol` class keeps the corresponding protocols's persistence variables values between invocations, unless explicitly changed by parameter assignments. In the command line program, this makes no sense, however.

6.6 Comments

Comments in the C syntax (starting with `/*` and ended by `*/`) are allowed, and ignored. Also, C++-style comments ("//", extending to the end of line) are accepted. (In the specifications, embedded comments in the IRP are not present.)

The function `list --irp` lists the IRP as given in the data base (preserving comments and whitespace), while `list --parsedirp` list the parsed version. The latter has comments and whitespace removed, and observes the `--radix` argument.

6.7 Whitespace

All white space, including line breaks, are ignored when parsing. (In the original spec, the IRP form had to be on one line.)

The function `list --irp` lists the IRP as given in the data base (preserving comments and whitespace), while `list --parsedirp` list the parsed version (with comments and whitespace removed).

6.8 Extents

The specification writes ``*An extent has a scope which consists of a consecutive range of items that immediately precede the extent in the order of transmission in the signal. ... The precise scope of an extent has to be defined in the context in which it is used.*", and, to my best knowledge, nothing more. I consider it as specification hole. I have effectively implemented this interpretation: “An extent has a scope which consists of a consecutive range of all non-extent items that immediately precede the extent in the order of transmission in the signal, starting with the first element after the last preceding extent, or from the start if there is no preceding extent.” Differently put: Every extent encountered resets the duration count.

6.9 Multiple definitions allowed

It turned out that the preprocessing/inheritance concept necessitated allowing several definition objects. These are simply evaluated in the order they are encountered, possibly overwriting previous content.

6.10 Names

The IRP documentation defines a name as starting with an uppercase letter, followed by an arbitrary number of uppercase letters and digits. I have, admittedly somewhat arbitrarily, extended it to the C-name syntax: Letters (both upper and lower cases) and digits allowed, starting with letter. Underscore "_" counts as letter. Case is significant.

6.11 Name spaces

There is a difference in between the IRP documentation and the implementation of the Makehex program, in that the former has one name space for both *assignments* and *definitions*, while the latter has two different name spaces. IrpTransmogrifier (just as the precessor IrpMaster) has one name space, as in the documentation. (This is implemented with the `NameEngine` class.)

6.12 Expressions

Several extensions to the expressions have been made. Note that, informally speaking, an expression is an integer, that, in different contexts is differently interpreted: as integer value, (potentially infinite) bit pattern (using 2-complement representation), and logical (0 is `false`, everything else is `true`).

For a summary of the complete syntax of `expression`, see the grammar.

6.12.1 Terminology

In this project, we use the terms *para_expression* (denoting an *expression* enclosed within parentheses) and *expression* instead of the specification's *expression* and *bare_expression*, since it was felt that the latter was wieldy and incompatible with normal-day usage.

6.12.2 Literals

Numerical literals can be given not only on base 10 (as in the specification), but also in bases 2 (with prefix `0b`), base 8 (prefix `0`), as well as base 16 (prefix `0x`).

A few pre-defined literals are introduced for convenience and readability. These are:

- `UINT8_MAX` = 2^8 - 1 = 0xFF = 255,
- `UINT16_MAX` = 2^16 - 1 = 0xFFFF = 65535,
- `UINT24_MAX` = 2^24 - 1 = 0xFFFFFF = 16777215,
- `UINT32_MAX` = 2^32 - 1 = 0xFFFFFFFF = 4294967295, and
- `UINT64_MAX` = 2^64 - 1 = 0xFFFFFFFFFFFFFFFF = 18446744073709551615.

6.12.3 Unary operators

In addition to the specification's unary minus ("-"), some additional unary operators have been implemented, described next.

6.12.3.1 Logical NOT, "!"

The exclamation point, logical not, acts like in C: it turns everything that evaluates to 0 (zero, `false`) into 1 (`true`), everything else to 0 (`false`).

6.12.3.2 Bit inversion, "~"

This operator turns all 0 to 1 and all 1 to 0 in the binary representation.

6.12.3.3 BitCount Function "#"

IrpMaster introduced the BitCount function as a unitary operator, denoted by "#". This is useful in many situations, for example, odd parity of `F` can be expressed as `#F%2`, even parity as `1-#F%2`. It is implemented through the Java Long.bitCount-function.

6.12.4 Binary operators

There are also some added binary operators, which will be described next.

The specification contains the *bitwise* operators "&", "|", and "^", having the syntax and semantics of C. In addition, the current implementation adds the following two *logical* operators.

6.12.4.1 Logical AND, "&&"

The logical operator && (with short-circuiting as in C and Perl) works as follows: To evaluate the expression `A && B`, first `A` is checked for being 0 or not. If 0, then A (which happens to be 0) (`false`) is returned, *without evaluating* `B`. If however, `A` is nonzero, `B` is evaluated, and the resulting value is returned.

6.12.4.2 Logical OR, "||"

The logical operator || (with short-circuiting as in C and Perl) works as follows: To evaluate expression `A || B`, first `A` is checked for being 0 or not. If non-zero, then A is returned, *without evaluating* `B`. If however, `A` is 0, `B` is evaluated, and the resulting value is returned.

6.12.4.3 Left shift "<<"

The left shift operators "<<" is implemented, with syntax and semantics as in the C and Java programming languages. (See this discussion.)

6.12.4.4 Right shift ">>"

The (arithmetic) right shift operator ">>" with syntax and semantics as ">>" (*not* ">>>", denoting logical shift) in the Java programming language. (Differently put, it preserves the leading bit, not shifting in 0.)

6.12.4.5 Numerical comparison operators, "<", "<=", ">", ">=", "==", "!="

The comparison operators have been added. They have the same syntax and semantics as in C, taking two numerical operators to 0 (false) or 1 (true).

6.12.5 Ternary operator

6.12.5.1 Conditional operator ?:

Similarly, the ternary operator `A ? B : C`, returning B if A is true (non-zero), otherwise C, has been implemented. As opposed to other operators (with the exception of exponentiation "**"), it is right associative.

6.13 Preprocessing and inheritance

Reading through the protocols, the reader is struck by the observation that there are a few general abstract "families", and many concrete protocol are "special cases". For example all the variants of the NEC* protocols, the Kaseikyo-protocols, or the rc6-families. Would it not be elegant, theoretically as well as practically, to be able to express this, for example as a kind of inheritance, or sub-classing?

For a problem like this, it is easily suggested to invoke a general purpose macro preprocessor, like the C preprocessor or m4. I have successfully resisted that temptation, and am instead offering the following solution: If the IRP notation does not start with "{" (as they all have to do to confirm with the specification), the string up until the first "{" is taken as an "ancestor protocol", that has hopefully been defined at some other place in the configuration file. Its name is replaced by its IRP string, with a possible parameter spec removed — parameter specs are not sensible to inherit. The process is then repeated up until, currently, 5 times.

The preprocessing takes place in the class `IrpDatabase`, in its role as data base manager for IRP protocols.

6.13.1 Example

This shows excepts from an example configuration file. Let us define the "abstract" protocol `metanec` by

```
                        [protocol]
                        name=metanec
                        irp={38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)
[A:0..UINT32_MAX]
```

having an unspecified 32 bit payload, to be subdivided by its "inherited protocols". Now we can define, for example, the NEC1 protocol as

```
                        [protocol]
```

```
                        name=NEC1
                        irp=metanec{A = D | 2**8*S | 2**16*F | 2**24*(~F:8)}
[D:0..255,S:0..255=255-D,F:0..255]
```

As can be seen, this definition does nothing else than to stuff the unstructured payload with D, S, and F, and to supply a corresponding parameter spec. The IrpMaster class replaces "`metanec`" by `{38.4k,564}<1,-1|1,-3>(16,-8,A:32,1,-78,(16,-4,1,-173)*)`" (note that the parameter spec was stripped), resulting in an IRP string corresponding to the familiar NEC1 protocol. Also, the "Apple protocol" can now be formulated as

```
                        [protocol]
                        name=Apple
                        irp=metanec{A=D | 2**8*S | 2**16*C:1 | 2**17*F | 2**24*PairID}
\
                        {C=1-(#F+#PairID)%2,S=135} \
                        [D:0..255=238,F:0..127,PairID:0..255]
```

The design is not cast in iron, and I am open to suggestions for improvements. For example, it seems reasonable that protocols that only differ in carrier frequency should be possible to express in a concise manner.

7 Installation

7.1 Installation of binaries

The most convenient way to install the program is to install IrScrutinizer, version 2.0.0 or later. For the Windows and the generic binary distribution, this will install a wrapper for IrpTransmogrifier too. The AppImage will start IrpTransmogrifier instead of IrScrutinizer, if called with the last component of the name equals to `irptransmogrifier` (for example, through a link ending with that name). (However, the MacOS installation presently does not support command line IrpTransmogrifier.)

Also RemoteMaster comes with IrpTransmogrifier and a wrapper (`irptransmogrifier.sh` or `irptransmogrifier.bat`) to start it as command line program.

IrpTransmogrifier can of course be installed separately. The latest released version can be found here. The program is basically just an executable jar-file. There is no "installer". Instead, unpack the binary distribution in a preferably empty directory. Start the program by invoking the wrapper (`irptransmobrifier.bat` on Windows, `irptransmogrifier.sh` on Unix-like systems like Linux and MacOS.) from the command line. Modify and/or relocate the wrapper(s) if desired or necessary. Do not double click the wrappers, since this program runs only from the command line. (Do not use the wrapper `irptransmogrifier` in the top directory of the source tree: it is intended only for development in the source tree; and is not intended for deployment.)

The program runs can be installed in a read-only location.

The Macintosh app for IrScrutinizer presently does not come with support for running IrpTransmogrifier as command line program.

The program presently requires Java 8 JRE or later to run. Some distributions of IrScrutinizer come with their own Java installations, that can run IrpTransmogrifier.

7.2 Building from sources

On Github, the latest official source- and binary distribution is found. Also, the built development version can be found.

The project uses Maven as its build system. Any modern IDE should be able to open/ import and build it as Maven project. Of course, Maven can also be run from the command line, like

```
mvn install
```

7.2.1 Third-party Java dependencies (jars)

The program depends on ANTLR4, Stringtemplate, as well as the command line decoder JCommander. When using Maven for building, these are automatically downloaded and installed to a local repository.

8 Usage of the program from the command line

Next it will be described how to invoke the program from the command line. The reader is assumed to possess an elementary command of the command line usage.

The usage message from `irptransmogrifier help --short` gives an extremely brief summary:

```
Usage: IrpTransmogrifier [options] <command> [command_options]
Commands:
   analyze         Analyze signal: tries to find an IRP form with parameters
   bitfield        Evaluate bitfield given as argument.
   code            Generate code for the given target(s)
   decode          Decode IR signal given as argument
   expression      Evaluate expression given as argument.
   help            Describe the syntax of program and commands.
   lirc            Convert Lirc configuration files to IRP form.
   list            List protocols and their properites
   render          Render signal from parameters
   version         Report version

Use
    "IrpTransmogrifier help" for the full syntax,
    "IrpTransmogrifier help <command>" for a particular command.
    "IrpTransmogrifier <command> --describe" for a description,
    "IrpTransmogrifier help --common" for the common options.
    "IrpTransmogrifier help --logging" for the logging related options.
```

Using from the command line, this is a program with sub commands. Before the sub command, common options can be given. After the command, command-specific options can be specified. Commands and option names can be abbreviated, as long as the abbreviation is unique. They are matched case sensitively, and can be abbreviated as long as the abbreviation is unambiguous.

All options have a long form, starting with two dashes, like `--sort`. (In rare cases, there might be more than one long name.) They can be abbreviated as long as the abbreviation remains unique. Most options also have a short, one letter form, starting with a single dash (like `-s`). For brevity, this document will not mention the short form. This information, if required, can instead be easily found using the `help` command.

Note that `help` and `version` are commands, not options, as in most other command line programs. (For compatibility reasons, also the options form works.)

The commands are briefly described next. Since the program contains its own documentation facility, the description is not aimed at being complete, but more to to comment upon the general idea behind.

8.1 analyze

The `analyze` command takes as input one or several sequences or signals, and computes an IRP form that corresponds to the given input (within the specified tolerances). The input can be given either as Pronto Hex or in raw form, optionally with signs (ignored). Several raw format input sequences can be given by enclosing the individual sequences in brackets ("[]"). However, if using the `--intro-repeat-ending` option, the sequences are instead interpreted as intro-, repeat-, and (optionally) ending sequences of an IR signal.

For raw sequences, an explicit modulation frequency can be given with the `--frequency` option. Otherwise the default frequency, 38000Hz, will be assumed.

Using the option `--input`, instead the content of a file can be taken as input, containing sequences to be analyzed, one per line, blank lines ignored. Using the option `--namedinput`, the sequences may have names, immediately preceeding the signal.

Input sequences can be pre-processed using the options `--chop`, `--clean`, and `--repeatfinder`.

The input sequence(s) are matched using different "decoders". Normally the "best" decoder match is output. With the `--all` option, all decoder matches are output. Using the `--decode` option, the used decoders can be further limited. The presently available decoders are: `TrivialDecoder, Pwm2Decoder, Pwm4Decoder, Pwm4AltDecoder, XmpDecoder, BiphaseDecoder, BiphaseInvertDecoder, BiphaseWithStartbitDecoder, BiphaseWithStartbitInvertDecoder, BiphaseWithDoubleToggleDecoder, SerialDecoder`.

The options `--statistics` and `--dump-repeatfinder` (the latter forces the repeatfinder to be invoked) can be used to print extra information. The common options `--absolutetolerance, --relativetolerance, --minrepeatgap` determine how the repeat finder breaks the input data. The options `--extent, --invert, --lsb, --maxmicroseconds, --maxparameterwidth, --maxroundingerror, --maxunits, --parameterwidths, --radix`, and `--timebase` determine how the computed IRP is displayed.

Using the `--girr` option, a Girr file can be produced. This embeds the generated IRP protocol in the file.

8.2 bitfield

The `bitfield` command computes the value and the binary form corresponding to the bitfield given as input. Using the `--nameengine` argument, the bitfield can also refer to names.

As an alternatively, the `expression` command sometimes may be used. However, a bitfield has a length, which an expression, evaluating to an integer value, does not.

8.3 code

Used for generating code for different targets.

8.4 decode

The `decode` command takes as input one or several sequences or signals, and output one or many protocol/parameter combinations that corresponds to the given input (within the specified tolerances). The input can be given either as Pronto Hex or in raw form, optionally with signs (ignored). Several raw format input sequences can be given by enclosing the individual sequences in brackets ("[]").

For raw sequences, an explicit modulation frequency can be given with the `--frequency` option. Otherwise the default frequency, 38000Hz, will be assumed.

Using the option `--input`, instead the content of a file can be taken as input, containing sequences to be analyzed, one per line, blank lines ignored. Using the option `--namedinput`, the sequences may have names, immediately preceeding the signal.

In the Harctoolbox world, IR sequences start with a flash (mark) and ends with a non-zero gap (space). In some other "worlds", the last gap is omitted. These signal are in general rejected. The option `--trailinggap duration` adds a dummy duration to the end of each IR sequence lacking a final gap.

Input sequences can be pre-processed using the options `--clean`, and `--repeatfinder`.

The common options `--absolutetolerance`, `--relativetolerance`, `--minrepeatgap` determine how the repeat finder breaks the input data.

8.4.1 Debugging

To debug why a certain signal/sequence does not decode the way expected, the logging facility and the `--protocol` argument (to reduce the logging output) can be useful to pinpoint the decoding process.

8.5 demodulate

This command demodulates its argument IrSequence, emulating the use of a demodulating IR receiver. This means that all gaps less than or equal to the threshold are squeezed into the preceeding flash. Typically the threshold is taken around the period of the expected modulation frequency.

8.6 help

This command list the syntax for the command(s) given as argument, default all. Also see the option `--describe` of the individual commands.

8.7 lirc

This command reads a Lirc configuration, from a file, directory, or an URL, and computes a corresponding IRP form. No attempt is made to clean up, for example by rounding times or finding a largest common divider.

8.8 list

This command list miscellaneous properties of the protocol(s) given as arguments. There are a large number of options for enabling or suppressing certain kind of output; use the command `irptransmogrifier list --help` for a list.

8.9 render

This command is used to compute an IR signal from one or more protocols, "render" it. The protocol can be given either by name(s) (or regular expression if using the `--regexp` option), or, using the `--irp` options, given explicitly as an IRP form. The parameters can be either given directly with the `--nameengine` option, or the `--random` option can be used to generate random, but valid parameters. (This is essentially a developer's and tester's option.) With the `--count` or `--number-repeats` option, instead an IR sequence is computed, containing the desired number of repeats.

The syntax of the name engine is as in the IRP specification, for example: `--nameengine {D=12,F=34}`. For convenience, the braces may be left out. Spaces around the equal sign "=" and around the comma "," are allowed, as long as the name engine is still only one argument in the sense of the shell — it may need to be enclosed within single or double quotes.

8.10 version

Reports version number and license.

9 Debugging/logging possibilities

The project contains quite powerful logging facilities, based on Java's `java.util.logging` framework. "Logging" is somewhat of a mis-normer for a program that runs only seconds, instead it is a form of debugging. Logging takes place according to the different levels: from highest to lowest `OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL`. Default level is `WARNING`.

The reader is assumed to know the basic principles for logging. Ideally, setting a lower and lower level should reveal more and more details on the inner workings of the program. Unfortunately, this is (at least presently) not always the case. Still, it may be useful for finding out why a particular signal did not decode as expected.

There are command line options, not only for setting the general log level, but also for changing the logging format, the logging file, for generating the log in XML format, and for setting the log level individually for different classes. See `help --logging` for the full list of these options.

10 The API

A Java programmer can access the functionality through a number of API functions.

The API is documented in standard Javadoc style, which can be installed from the source package, just like any other Java package. For the convenience of the user, the Javadoc API documentation is also available here current, released version only).

The released versions of project is available in the Maven central repository, and can easily be integrated into other Maven projects. For this, include the lines

```
        <dependency>
            <groupId>org.harctoolbox</groupId>
            <artifactId>IrpTransmogrifier</artifactId>
            <version>1.2.10</version>  <!-- or another supported version -->
        </dependency>
```

in the `pom.xml` of the importing project.

11 Appendix: ANTLR4 Grammar

This appendix shows the grammar file for IRP. It is used to generate the Java code for the IRP parser. It is also (in contrast to the ANTLR3 grammar used in IrpMaster) quite readable for humans.

```
/*
Copyright (C) 2017 Bengt Martensson.
```

```
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program. If not, see http://www.gnu.org/licenses/.
*/

grammar Irp;

// 1.7
// class Protocol
// Extension: * instead of ?, parameter_specs
protocol:
    generalspec bitspec_irstream definitions* parameter_specs? EOF
;

// 2.2, simplified
// Difference: This a simplified version; implementing exclusions is not really
// mainstream... Some silly input is not rejected.
// My semantics: read left-to-right, later entries overwrite.
// class GeneralSpec
generalspec:
    '{' generalspec_list '}'
;

generalspec_list:
    /* Empty */
    | generalspec_item (',' generalspec_item )*
;

// extension: dutycycle_item
generalspec_item:
    frequency_item
    | unit_item
    | order_item
    | dutycycle_item
;

frequency_item:
    number_with_decimals 'k'
;

dutycycle_item:
    number_with_decimals '%'
;

unit_item:
    number_with_decimals ('u' | 'p')?
;

// enum BitDirection
order_item:
    'lsb' | 'msb'
;

// 3.2
// abstract class Duration
```

```
// Note: spec did not consider extent as a duration
duration:
    flash
    | gap
    | extent
;

// class Flash extends Duration
// called flash_duration in spec
flash:
    name_or_number ('m' | 'u' | 'p')?
;

// class Gap extends Duration
// called gap_duration in spec
gap:
    '-' name_or_number ('m' | 'u' | 'p')?
;

// class NameOrNumber
// Extension: Spec allowed number (integers) only
name_or_number:
    name
    | number_with_decimals
;

// 4.2
// class extent (extends Duration)
// Semantics: An extent is a gap, with all preceding durations in the
// containing bare_irstream subtracted. More than one extent in one
// bare_irstream are thus allowed. The "counting" starts anew after each extent.
extent:
    '^' name_or_number ('m' | 'u' | 'p')?
;

//  5.2
// abstact class BitField extends IrpObject
// class FiniteBitField extends BitField
// class InfiniteBitField extends BitField
bitfield:
    '~'? primary_item ':' '-'? primary_item (':' primary_item)? # finite_bitfield
    | '~'? primary_item '::' primary_item                       # infinite_bitfield
;

// abstract class PrimaryItem
primary_item:
    name
    | number
    | para_expression
;

// 6.2
// class IrStream
irstream:
    '(' bare_irstream ')' repeat_marker?
;

// class BareIrStream
bare_irstream:
    /* Empty */
    | irstream_item (','  irstream_item)*
;

// interface IrStreamItem
// Note: extent is implicit within duration
```

```
irstream_item:
    variation
    | bitfield  // must come before duration!
    | assignment
    | duration
    | irstream
    | bitspec_irstream
;

// 7.4
// class BitSpec
bitspec:
    '<'  bare_irstream ('|' bare_irstream)* '>'
;

// 8.2
// class RepeatMarker
// NOTE: Semantically, at most one infinite repeat in a protocol makes sense.
repeat_marker:
    '*'
    | '+'
    | number '+'?
;

// class BitspecIrstream
bitspec_irstream:
    bitspec irstream
;

// 9.2
// class Expression
// called expression in spec
para_expression:
    '(' expression ')'
;

// called bare_expression in spec
expression:
                   primary_item
    |              bitfield
    |              '~'            expression
    |              '!'            expression
    |              '-'            expression
    |              '#'            expression
    | <assoc=right> expression '**'                expression
    |              expression ('*'  | '/' | '%')   expression
    |              expression ('+'  | '-')         expression
    |              expression ('<<' | '>>')        expression
    |              expression ('<=' | '>=' | '>' | '<') expression
    |              expression ('==' | '!=')        expression
    |              expression '&'                  expression
    |              expression '^'                  expression
    |              expression '|'                  expression
    |              expression '&&'                 expression
    |              expression '||'                 expression
    | <assoc=right> expression '?'                 expression ':' expression
;

expressionEOF:
    expression EOF
;

// 10.2
// (class NameEngine)
definitions:
```

```
    '{' definitions_list '}'
;

definitions_list:
    /* Empty */
    | definition (',' definition)*
;

definition:
    name '=' expression
;

// 11.2
assignment:
    name '=' expression
;

// 12.2
// Variations are only allowed within infinite repeats.
variation:
    alternative alternative alternative?
;

alternative:
    '[' bare_irstream ']'
;

// 13.2
// class Number
number:
      INT
    | HEXINT
    | BININT
    | 'UINT8_MAX'
    | 'UINT16_MAX'
    | 'UINT24_MAX'
    | 'UINT32_MAX'
    | 'UINT64_MAX'
;

// class numberWithDecimals extends Floatable
number_with_decimals:
    number
    | float_number
;

// Due to the lexer, have to take special precautions to allow name-s
// to be called k, u, m, p, lsb, or msb. See Parr p.209-211.
// class Name
name:
    ID
    | 'k'
    | 'u'
    | 'p'
    | 'm'
    | 'lsb'
    | 'msb'
;

// class ParameterSpecs
parameter_specs:
    '[' parameter_spec (',' parameter_spec )* ']'
    | '['  ']'
;
```

```
// class ParameterSpec
parameter_spec:
      name     ':' number '..' number ('=' expression)?
    | name '@' ':' number '..' number  '=' expression
;

// class FloatNumber
float_number:
    '.' INT
    | INT '.' INT
;

// Extension: Here allow C syntax identifiers;
// Graham allowed only one letter capitals.
ID:
    ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
;

INT:
    ( '0' .. '9')+
;

HEXINT:
    '0x' ( '0' .. '9' | 'a' .. 'f' | 'A' .. 'F' )+
;

BININT:
    '0b' ( '0' | '1' )+
;

// Extension: Not present by Graham.
COMMENT: // non-greedy
    '/*' .*? '*/'                       -> skip
;

LINECOMMENT:
    '//' ~('\n'|'\r')* '\r'? '\n'       -> skip
;

WS:
    [ \t\r\n\u000C]+                    -> skip
;
```