Writing a Formal IT Specification

Rex Jaeschke (rex@RexJaeschke.com)

Last modified: 2023-08-05

Rex is an independent consultant who has worked in the IT industry since 1976. Since 1984, he's been involved in producing and reviewing formal specifications for C, C#, CLI (a subset of Microsoft's .NET™ framework), C++/CLI, Hack, Java™, JavaScript™, Office Open XML (the file format used by Microsoft's Office™ suite as well as by other products), PHP, Powershell™, and XPS (XML Paper Specification). This work was done via ANSI, ISO, Ecma International, open-source forums, and commercial organizations.

[As this author's experience with IT standardization has mostly involved programming languages, comments and examples used throughout this document sometimes refer specifically to that target audience.]

Acknowledgements: This document contains small amounts of text from formal specifications for the C and C# programming languages. Thanks to David Keaton for his contributions, primarily regarding Unicode.

Contents

1. Introduction

A *formal specification* for an IT project allows implementers to understand what is required to build an implementation (or create a process) that conforms (see §4) to that specification, and it allows a conformance test suite (or checklist) to be developed that can be used to check an implementation's conformance. Users of tools (or processes) that conform to that specification can use the specification to learn the potential impact of moving source code, data, or processes between different implementations.

[For a programming language, this involves a description of that language's lexical and syntactic grammar and related constraints, its type system, type conversion rules, expressions, and statements, among other things. It might also describe the minimal library and runtime environment needed to support that language, and it might describe how to interoperate with other languages and to access facilities in underlying platforms. Programmers working with that language can use the specification to learn the potential impact of moving source code between conforming implementations with regard to syntactic and semantic issues.]

This document outlines a number of considerations involved when creating a formal IT specification, in general, and for a programming language or software component, in particular.

Important Action: Write a clear and concise charter for the formalization effort. What are the goals, and just as importantly, the non-goals. Separate reality from the "wouldn't it be nice to" thoughts.

2. Starting Point: Existing Base Document or No

Most specifications that get formalized start out as some sort of less-formal documentation, often written by the person who dreamed up the idea, possibly with contributions from a few others. Over time, that specification has evolved, possibly in large and inconsistent ways. A few specifications start as an idea and are formalized from the beginning before much (if any) implementation work is done beyond a proof-of-concept. I'll refer to these approaches as

- Consolidating prior art
- Design by committee

Both approaches have their pros and cons.

When consolidating prior art, the base document may be quite large, and have terminology and extensive formatting on which a non-trivial user base relies. Trying to change that at all can be a challenge, and if non-trivial changes are made, will support/tooling be made available to convert existing implementations and their data? In short, what is the impact on the (potentially large) implementer base?

The good news when starting from scratch is that there is no existing investment in a particular approach, and you can do anything! The bad news is that you have to do everything!

Building on a proven and successful model is quite a different task than trying to define a model that is intended to become successful!

3. Do You Need Official Accreditation?

A *Standards Development Organization* (SDO) is one that publishes a formal specification. (An SDO might be accredited somehow, or it might just be a recognized [typically non-profit] industry organization.) While some SDOs make their specifications freely available from a website, support web-based publishing, and have minimal rules regarding document format, others (such as ISO), charge for their specifications, require that documents be paginated, make them available in hard-copy form as well, and have very detailed document format requirements. They might also restrict availability of interim drafts of specifications developed under their umbrella.

The procurement process for some organizations (such as the EU and some federal and state governments) requires that certain products conform to specifications produced by accredited SDOs.

A *Technical Committee* (TC) is one that produces a formal specification. It might publish that specification itself (in which case, the TC is also an SDO) or it might submit the specification to an accredited SDO (for example, to ISO via it's Publicly Available Specification [PAS] process) in order to get some sort of widely recognized branding.

Note that the title of this section says "Need" not "Want!" For example, while ISO branding might be seen as an attractive attribute, are you really willing to submit the specification in Microsoft Word format using a specific template; be restricted to a very small set of fonts; be required to number each note, example, figure, and table, and then create at least one cross-reference to each of those numbers; to have the text right-justified; and to use a specific set of bullet symbols for nested unnumbered lists?

[For an example of a very large and complex set of publishing requirements, see "ISO/IEC Directives, Part 2: Principles and rules for the structure and drafting of ISO and IEC documents" and ISO's in-house Style Guide.]

Important Decision: Will you be bound by an SDO's specific set of rules? The answer is either "Yes" or "No," never "Maybe!" Once you make that decision, you should expect to be locked-in to it, forever! Note that even if the answer is "No," it's OK to borrow various aspects from such rules, as some of them can make sense outside their context.

See §7 below for more discussion on formatting, which may well influence your decision on this section's question.

4. Conformance

Conformance (sometimes called *compliance*) is the degree to which an implementation agrees with the syntactic and semantic requirements of some formal specification. In the context of a programming language or library, this gives rise to the following terms:

- A *strictly conforming program* shall use only those features of the language specified in this specification as being required. (This means that a strictly conforming program cannot use any conditionally normative feature.) It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior.

- A *conforming program* is one that is acceptable to a conforming implementation. (Such a program is permitted to contain extensions or conditionally normative features.)
- A *conforming implementation* shall accept any strictly conforming program.

The text in a specification that specifies requirements is considered *normative*. All other text is *informative* (sometimes called *non-normative*); that is, is for information purposes only. Unless stated otherwise, all text is normative. Normative text can be further broken into *required* and *conditional* categories. *Conditionally normative* text specifies a feature and its requirements where the feature is optional. However, if that feature is provided, its syntax and semantics shall be exactly as specified.

There are two common ways to identify text as being normative:

- Use English words such as “shall,” “shall not,” “may,” and “may not” [as ISO requires].
- Use an all-uppercase set of words (such as “MUST,” “MUST NOT,” “REQUIRED,” and “SHALL,” as defined by RFC 2119, *Key words for use in RFCs to Indicate Requirement Levels*, March 1997, https://datatracker.ietf.org/doc/html/rfc2119.

The second (RFC) approach allows tools to programmatically search a specification for such requirements, while the first does not.

Important Decision: What mechanism will you use to identify text as being normative?

Ideally, all requirements of a specification are absolute with regards to reproducible behavior. However, it can be useful, and it’s a widespread practice to have a number of flavors of behavior. For example:

- *behavior, implementation-defined* — unspecified behavior where each implementation documents how the choice is made.
- *behavior, locale-specific* — behavior that depends on local conventions of nationality, culture, and language that each implementation documents.
- *behavior, undefined* — behavior, upon use of a non-portable or erroneous construct or of erroneous data, for which this specification imposes no requirements.
- *behavior, unspecified* — behavior where this specification provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

These provide wiggle room for certain kinds of interesting situations. For example, when merging different kinds of prior art, compromises might have to be made by allowing multiple ways of doing something, as each implementation up to that point has its own set of (entrenched) followers. That is, maybe it’s not possible or a good idea to break existing implementations.

Not all programming language specifications have these distinctions; C is one that does.

Important Decision: What flavors of behavior will you allow?

Over time, it is not uncommon to introduce a new approach to something, and, in the process, to advise against using the previous approach. The old approach might even be removed in a subsequent revision. Advising about

such a situation is known as *deprecation*. If it is intended to remove support for some approach, it is useful to deprecate that in one edition before removing it in the next. That is, to give one version's worth of notice.

As most marketplace-relevant specifications get revised over time, it is useful for a consumer of a conforming implementation to be able to inquire at runtime, as to which edition the implementation claims to conform.

5. Legal Considerations

When consolidating prior art, existing base documents may well be copyrighted.

Who will own the copyright of the final specification?

Any trademarks used need to be acknowledged.

Are patents involved? If so, can they be used royalty-free?

Who owns contributions made once the standardization process begins?

Important Action: Have at least one person charged with learning the legal issues and communicating them with the group.

6. Specification Packaging

Here are the main ways of packaging a specification:

1. The specification is a *single part*; everything is contained in a single document (which could be represented in source as one or more files). Let's call it SpecX:2021.
2. The specification is *multi-part*: each part is a separate document, but the parts are related. Let's call the parts of such a 3-part specification SpecY-1:2019, SpecY-2:2020, and SpecY-3:2021. Some parts could be formal specs, others might be informative Technical Reports (TRs), rationale about the decisions made and the specification history and evolution, user guides, a test suite, whatever! Development and evolution of the parts might be asynchronous, even managed by different TCs or subgroups. Compliance might be on a per-part basis. In one arrangement, the first few parts might be mandatory core components, while other, optional, parts might be added later. In such cases, an alternative to having multiple, optional parts is to add new appendices to an existing (core) spec. [One non-trivial consideration is that of having one specification/part refer to another through synchronized linking. This can be difficult to achieve/manage when they are in separate documents.]
3. Multiple, separate specifications, which are unrelated or not closely related.

When choosing a packaging approach, consider how it might be mapped onto the editing platform (§8).

Important Decisions: What packaging approach will you choose? Is your specification likely to grow with respect to the number of components? How will revision of one component/part impact the others? Will separate components/parts be formalized by different groups?

7. Specification Formatting

It is this author's opinion that the shape and format of a specification should be determined as much as possible by the TC producing it. After all, one presumes the TC's members know their own subject matter and audience, and they have more than a little idea of what their industry needs, wants, and will tolerate.

This author's goals in producing a specification include the following:

- Actively support the use of collaborative platforms like GitHub.
- Allow the publication in non-paginated form with little or no transformation needed from the source form.
- Allow the specification organization and form to be as lightweight as possible.
- Allow the TC maximum flexibility in deciding things like heading styles, fonts, list numbering, and list bullet formats.

Bottom line, the SDO shouldn't get in the way of the experts writing the specification as they see fit for their industry!

8. The Editing and Publishing Platforms

8.1 A Collaborative Approach to Editing

The advent of collaborative tools like GitHub (using markdown) has made the specification-editing process much easier. In the classic "everything in one file" model, the specification was presided over by a single editor. TC members made proposals by writing a paper showing the changes and additions they wanted to the spec, and once some version of those were approved by the membership, the editor integrated them into the base specification, which might not be made available for some time. With the GitHub-like model, every member (and even people outside the TC, but with access to an electronic copy of the specification) can propose edits, and each person's contribution is kept separate and can be applied or undone separate from all others.

In this day and age, it is important to be able to publish a specification without requiring a lot of extra work on top of maintaining the base document, and to present it to the public in a manner with which they are used to working, which for many—maybe most—implementers is not in a fixed-width, paginated form.

8.2 Publishing in Something Other than Markdown on GitHub

Often, a GitHub-based specification is made up of a set of source files, one per chapter, with references between chapters implemented as hyperlinks. This is fine if the end-user will be happy to read the specification directly from a GitHub repo. However, when that is *not* seen as the most appropriate rendering for end users, the specification files will need to be transformed into something else, such as a single HTML file. The challenge then is to make sure all the links are adjusted accordingly.

By design, markdown is simple and relatively limited when it comes to certain typesetting requirements. For example, it has *no* support for underlining text or the use of subscripts and superscripts. As it happens, one can "extend" markdown's capabilities by embedding corresponding HTML tags. For example, "_{xxx}" and

"^{xxx}" allow xxx to be set as a subscript or superscript, respectively. And given markdown's limited support for tables, it can be really tempting to embed HTML tags to enhance that. However, if the markdown eventually has to be transformed to something *other* than HTML, each kind of embedded HTML augmentation will need to be handled specially.

If you chose to transform to a paginated rendering, you may want to figure out how to handle bad line and page breaks.

9. Grammar Notation and Validation

A programming language specification needs to clearly state the syntax of each lexical and syntactic construct. This can be done in one of two ways: using formal grammar notation, and in narrative.

[Other specifications than can benefit from a formal grammar include library interfaces, query facilities, configuration file contents, and any form of script.]

The following example shows a subset of the grammar for C#'s integer literals, expressed using a notation involving subscripts:

integer-literal::
    decimal-integer-literal
    hexadecimal-integer-literal

decimal-integer-literal::
    decimal-digits *integer-type-suffix*$_{opt}$

decimal-digits::
    decimal-digit
    decimal-digits *decimal-digit*

decimal-digit:: one of
    `0 1 2 3 4 5 6 7 8 9`

integer-type-suffix:: one of
    `U u L l UL Ul uL ul LU Lu lU lu`

Here is the same subset expressed using the (popular) ANTLR notation (which does not involve the use of subscripts):

```
Integer_Literal
    : Decimal_Integer_Literal
    | Hexadecimal_Integer_Literal
    ;

fragment Decimal_Integer_Literal
    : Decimal_Digit+ Integer_Type_Suffix?
    ;
```

```
fragment Decimal_Digit
    : '0'..'9'
    ;

fragment Integer_Type_Suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' |
'lu'
    ;
```

The advantage of using a notation that does not involve characters other than the basic English/Latin set, is they are easy to typeset and easier to manipulate by tools processing the raw source.

A formal grammar is preferrable to an English-language description, as the latter can more easily be ambiguous. No value is added if the narrative simply repeats what the grammar already indicates. In fact, saying the same thing twice almost always leads to differences when one is changed and the other isn't, in which case, which one is the normative requirement?

It is not always possible or desirable for a grammar to be complete, in which case, narrative can be used to add *constraints*. For example, the ANTLR grammar for declaring a field in C# is, as follows:

```
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    …
    | 'volatile'
    | unsafe_modifier
    ;
```

However, this allows multiple occurrences of the same modifier, which is not permitted. As such, the following constraint is provided: "It is an error for the same modifier to appear multiple times in a *field_declaration*."

It can be very useful to be able to programmatically extract the grammar from a specification and run it through some corresponding validation tool to make sure that grammar is correct. (This helps detect typos introduced when a grammar is edited.)

Note that ANTLR has an important restriction with regards to left-recursive rules, and while those are common and desirable, it can be a challenge to build a tool that transforms to something that will validate.

In general, writing a lexer and parser, either by hand or using a tool such as ANTLR, is outside the scope of a language specification. However, it is useful for a specification to minimize the gap between the specified grammar and that required to build a lexer and parser.

[The Ecma-based TC49/TG2 committee that maintains the C# specification has built an ANTLR grammar extraction tool for validation. Click here for more information.]

Important Decisions: What notation will you use for lexical and syntactic grammars? How will you express constraints? Will you build a grammar extraction tool, so the grammar can be validated?

10. Example Extraction and Testing

Perhaps the most common occurrences of informative text (§4) are examples containing code (or data) that demonstrates some point. Inevitably, readers of a specification will extract such examples to learn and to write tests. As such, it is important that examples are well-formed, work, and produce the claimed behavior. It can be challenging to build a tool to extract and test embedded examples, especially when they contain only fragments of a complete program.

[The Ecma-based TC49/TG2 committee that maintains the C# specification has built an example extraction and testing system. Click here for that system's user guide to see the kinds of things such a tool needs to handle.]

11. Support for Unicode

The lingua franca for IT specifications is English, and the keywords in mainstream programming languages and their associated libraries are likely to be written using USA/English letters (possibly with underscores and decimal digits). That said, with the advent of Unicode, mainstream development platforms and tools provide support for dealing with the world's writing systems, and **a specification that fails to recognize and handle other than USA/English characters in source files and data might run into opposition.**

From a programming language perspective, this gives rise to the following questions:

1. Can an input source file contain any/all Unicode characters?
2. Which character encodings is a conforming implementation (§4) required to accept? (For Unicode, presumably, at least UTF-8.)
3. Can a character literal or a character in a string literal be expressed using a Unicode escape sequence or hex representation?
4. Can an identifier (a user-defined name for a variable, procedure, user-defined type, and such) contain non-USA/English letters and digits?
5. Do runtime library procedures correctly handle Unicode characters?

Two Unicode technical reports are particularly helpful in constructing a programming language's rules for using Unicode characters:

1. The Unicode Consortium. *Unicode Standard Annex, UAX #44, Unicode Character Database [online]*. Edited by Ken Whistler. Available at https://www.unicode.org/reports/tr44. [UAX #44 lists properties of Unicode characters, such as punctuation, white space, and upper or lower case, which can make it easier to categorize the enormous number of characters involved, in order to specify rules for their use in a programming language. When Unicode changes, this report is kept up to date, making it easier for a

programming language specification that refers to UAX #44 to stay up to date with the latest character set developments.]

2. The Unicode Consortium. *Unicode Standard Annex, UAX #31, Unicode Identifier and Pattern Syntax [online]*. Edited by Mark David and Robin Leroy. Available at https://www.unicode.org/reports/tr31. [UAX #31 builds on the properties in UAX #44 to describe a syntax for identifiers. A programming language can use this syntax as-is, or use it as a base with modifications specific to that programming language. This report is also kept up to date.]

Important Decision: Do you need to provide support for dealing with other than USA/English characters?

12. Support for Internationalization

It is reasonable that end-users of a conforming implementation interact in their own language using their own cultural conventions. This ability is greatly helped by the use of locales.

Important Decision: Do you need to provide support for dealing with locales?

13. Chapter Layout

Consider having the following chapters in a specification:

13.1 Front Matter: Foreword/Introduction

Identify the SDO and TC that produced the spec.

Perhaps have a bit of history as to the lead-up to making this spec.

For a revision, it might be useful to state the main differences between this and the previous version. However, if over time there are an increasing number of versions, and the industry supports older ones as well as the current one, a better approach is to have an appendix containing the differences between each successive version. Such information could also go in a separate part.

[ISO only makes the current edition available, which can be problematic.]

13.2 Scope

Have a few paragraphs that state the high-level purpose/goals of this spec. It can also be useful to say what will *not* be covered.

13.3 Terms and Definitions

Define the core terms that will be used throughout. They might be organized alphabetically, or grouped by category or some characteristic. If necessary, add some small examples or tutorial material to enhance their description, as these are the basic building blocks.

It might be useful to say something like “Definitions for terms not defined here can be found in Wikipedia,” assuming that is true. Or point to other locations of industry-specific terms.

[ISO requires each entry be a phrase, possibly with a note, and nothing more.]

It is common in publishing to introduce non-core terms when they are first used in the body of the spec, by setting them in italics.

[ISO does not permit such first-use highlighting.]

Most importantly, do not define a term in more than one place, as they likely will eventually get out of synch!

If new terms and their definitions are typeset in a specific way, they can be extracted from the specification programmatically.

Important Decision: How will terms be defined?

13.4 Normative References and Bibliography

A *normative reference* list has the titles of all documents necessary for the understanding and implementation of the specification, and, hopefully, where electronic copies can be obtained. (One part in a multi-part specification may well reference a different part in that series.) A *dated reference* is one that refers to a specific version, whereas an *undated reference* refers to the most current one. Both have their pros and cons: while having an undated reference automatically refers to any new version that may become available, that version might have added or changed certain features that are incompatible with the referencing spec.

It is a good idea to make sure that each normative reference is referenced at least once in normative text.

A *bibliography* list has the titles of documents that might be helpful—but are not necessary—for the understanding and implementation of this specification.

While it can be useful to reference each bibliography entry at least once, that is not necessary.

[ISO has certain requirements on normative reference entries, and it requires the bibliography to be an unnumbered appendix at the very end. It requires there be references to every entry. In both lists.]

13.5 Conformance Requirements

See §4.

13.6 Overview/Mini-Tutorial

It might be useful to describe one or more scenarios of using various combinations of the spec's features.

Alternatively, this information could reside in a separate part of a multi-part spec, and be updated asynchronously without requiring the core version to be reissued just to add such (informative-only) text.

13.7 Specification Chapters

Here's where the meat of the specification is defined.

A chapter may contain normative and/or informative information.

[ISO calls a chapter a *clause*, and a subchapter/section (and subsubchapter/subsection) a *subclause*. It has rules about numbering and headings.]

13.8 Specification Appendices

Typically, this is where supporting information goes. However, for large amounts of such information, consider placing it in a separate part of a multi-part specification (which allows it to be maintained without requiring the core specification to be reissued).

An appendix may contain normative and/or informative information.

[ISO calls an appendix an *annex*. It has rules about numbering, headings, and whether an annex is normative.]

13.9 Fitting It All Together

An SDO may provide descriptive examples that can help authors to visualize how a finished product will look in a suggested format. For example, ISO provides a *Model document of an International Standard* and a *Model Document of an Amendment* which contain helpful hints within the suggested chapter layouts.

14. Section Layout

For consistency, it is useful to make sections that describe variations on a theme have a common layout. For example:

6. Statements

6.3. The `for` statement

<Introductory statement about this feature's purpose>

Syntax

<The lexical or syntactic grammar>

Constraints

<Any constraints on the grammar that cannot (easily) be expressed in the grammar directly>

Semantics

<The meaning of all forms of the feature.>

Examples

<Code fragments to reinforce the syntax and semantics. An alternate to having all the examples at the end of the section is to place them in-line in the other blocks above, directly where they apply.>

This approach is also applicable for preprocessor directives and library facilities.

Important Decision: How will you layout feature descriptions in a consistent manner?

15. Future Extensions

As a specification evolves, it might need or benefit from a number of extensions. In the context of a programming language:

1. What is the policy regarding adding new keywords? Can they be contextual only? That is, can they be recognized as such in certain contexts only?
2. What is the policy regarding adding new operators? They could be added using symbols or as identifier-like names.
3. Are namespaces supported? For programming in the large, it can be very useful to be able to reuse names within different namespaces, for different purposes.
4. What is the policy regarding adding new library functions? Are certain name prefixes or patterns reserved?

16. Chapter and Section Numbering

In non-trivial specifications it is common to number chapters and sections, and to have cross-reference links to such numbered headings. For example, in §14, we see that the chapter covering statements is numbered 6, while the **`for`** statement within that chapter is numbered 6.3.

Such numbering is supported in mainstream word-processing programs, but is **not** supported in markdown.

[The Ecma-based TC49/TG2 committee that maintains the C# specification has built a tool to allow assignment of such numbering, and the insertion and deletion of chapters and sections. Click here for more information.]