JOURNAL OF APPLIED CRYSTALLOGRAPHY
ISSN 1600-5767

A portable general-purpose application programming interface for CIF 2.0

John C. Bollinger*

Department of Structural Biology, St Jude Children's Research Hospital, Memphis, Tennessee 38105, USA.
*Correspondence e-mail: john.bollinger@stjude.org

Received 21 August 2015
Accepted 16 November 2015

Edited by A. J. Allen, National Institute of Standards and Technology, Gaithersburg, USA

Keywords: CIF; CIF 2.0; computer programs.

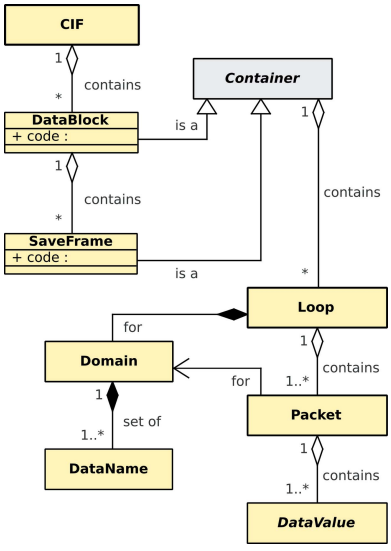
Supporting information: this article has supporting information at journals.iucr.org/j

The CIF API is an application programming interface and accompanying reference implementation for reading and writing CIFs and manipulating CIF data, with support for all versions of CIF through CIF 2.0. It features full support for Unicode in data block and save frame codes, data names, and data values; flexible character encoding; CIF 2.0 List and Table data types; CIF version auto-detection; event-based parsing; and arbitrary-precision numeric values. The interface and implementation are written in portable C, and they have been successfully built and tested on Linux, OS X and Windows. The CIF API is open-source software, available for use under the GNU Lesser General Public License.

1. Introduction

Since its introduction in 1991, the Crystallographic Information File (CIF; Hall *et al.*, 1991, 2005) has become a well established format for data exchange and archiving in crystallography. CIF and applications built upon it now support a global ontology-based Crystallographic Information Framework for data exchange and processing (also CIF; Hall & McMahon, 2005). Over that time, however, conventions, capabilities and expectations for electronic data have evolved. The constraints on CIF and its parent format, STAR (Hall, 1991), were unremarkable when they were first introduced, but a quarter-century later, science requires richer, more flexible data formats than those constraints readily permit. These requirements are addressed in a new version of STAR (Spadaccini & Hall, 2012*a*) that takes Unicode as its character repertoire, modifies and extends the CIF's quoting rules, and provides new list and associative array data types. These capabilities underpin a new STAR dictionary definition language, DDLm (Spaddaccini & Hall, 2012*b*), that supports associating programmatic behaviours ('methods') with data definitions, among other advances. These features are being introduced into CIF (both senses) as well, initially in the form of DDLm-based data dictionaries and a new version of the CIF format (CIF 2.0; Bernstein, *et al.*, 2016).

Its name notwithstanding, CIF technology is not inherently specific to crystallography. Nevertheless, although CIF has flourished in crystallography, neither it nor STAR has achieved much penetration into other disciplines. There are undoubtedly many reasons for the lack of CIF uptake outside the discipline of its genesis, but one of them is software developer disinterest and even resistance to providing CIF support in their software. Anecdotal evidence collected by the IUCr's Committee on the Maintenance of the CIF Standard (COMCIFS) revealed that a commonly cited barrier to programmers both in and out of crystallography integrating



CIF support into their software was the lack of a standard application programming interface (API) to assist them in reading and writing CIF data. This barrier is only heightened by divergence of CIF 2.0 from CIF 1.1, which is not a welcome characteristic even to developers who already support CIF. Although there are excellent libraries in various computer languages for handling CIF 1.1 data (Hall & Bernstein, 1996; Westbrook *et al.*, 1997; Hester, 2006; Lin, 2010; Gildea *et al.*, 2011), there are few for CIF 2.0, and their language coverage is much less.

Wanting to reduce the barrier to incorporating CIF support into software generally, and especially wanting to foster support for CIF 2.0, in 2011 COMCIFS decided to choose or create an API for CIF. This author was invited to lead the ensuing discussion, which was carried out on the IUCr's public web forum (IUCr, 2011). The requirements that emerged from that effort (see below) were not all satisfied by any one existing CIF support library known to any of the participants. Therefore the author undertook development of a detailed interface and reference implementation of just such a library: the CIF API.

2. Requirements and design considerations

As an initial matter, there can be some disagreement about exactly what an 'API' comprises, as the term is used in at least two senses: (1) in a strict sense of the word 'interface', limiting an API to data types, function signatures and constants exposed for use by client applications; and (2) in an inclusive sense that additionally incorporates implementations of the interface functions. An API in the former sense is of little practical value without at least one implementation, but that sense of API is distinguished by affording the possibility of multiple independent implementations. The distinction between API and implementation was maintained throughout the collaborative specification process, and the CIF API provides a detailed programmatic interface in the former sense. As discussed in more detail below, however, this is paired with a reference implementation to validate the API design and make it useful in practice.

Having defined what 'API' meant for the purposes of this work, an essential first task in selecting or creating an API for CIF 2.0 was to establish acceptance requirements. Both requirements for functionality and constraints on implementation were identified, as described below.

2.1. Functional requirements and non-requirements

A central requirement and impetus for the CIF API is full support for the CIF 2.0 specifications. The API must provide for inputting, managing and outputting character data, potentially drawing from the full range of Unicode. It must handle the new compound data types. Its parser must recognize and handle the new and revised quoting mechanisms for character data, and its built-in output facilities must format character data according to CIF 2.0 requirements.

Furthermore, the API's usefulness would be unduly circumscribed if it supported only CIF 2.0. Inasmuch as CIF is in part an archival format, CIF 1.1 (and older) documents can be expected to persist indefinitely, and it is desirable to have a single system that can handle those as well as CIF 2.0 documents. In the most general terms, then, the API must support inputting and parsing CIF text from external sources; it must support outputting logical CIF structure and content to external sinks as well formed CIF text; and between source, if any, and sink, if any, and in memory where applicable, the API must support all valid inquiries and modifications of logical CIF structure and data.

On the other hand, the objective was a CIF API, not an API specific to any given CIF dictionary or application. In particular, even though part of the motivation for CIF 2.0 itself was to support DDLm, the CIF API is not specific to DDLm or to any other DDL or dictionary. This is in no way intended to discount the value of CIF dictionaries or of validating instance documents against such dictionaries; it simply reflects the chosen purpose and scope of the project. It is anticipated that the CIF API will readily support higher-level libraries and applications that provide for validation, dictionary-specific functions and data structures, and the like, but these are not inherently necessary to use the CIF API with any particular CIF data.

2.2. Implementation constraints

Inasmuch as the project was intended to answer a call for a consistent, broadly usable API, it was a project objective that the API should support programs written in multiple languages without being implemented independently in each target language. That is, the interface should take a single form that readily can be bound to other programming languages. Similarly, it was an objective that the API should support applications written for the widest possible array of computing platforms. The combination of these requirements was distilled to a simpler one: the API definition and implementation must be written in portable C, compatible with both the C90 and the C99 standard (International Standards Organization, 1990, 1999).

Additionally, it was recognized that there are at least two distinct modes in which applications may want to consume CIF: (1) a mode based on constructing and manipulating an internal object model of a whole CIF, analogous to XML DOM (W3C, 1998), and (2) a lightweight, event-driven mode, analogous to SAX (XML-DEV, 1998). The former is the more conventional form for CIF support libraries, but the latter is important for handling large CIFs and for working under strict performance or memory usage constraints. Instead of separate efforts to support these alternatives, it was decided that the API must provide for both.

Regardless of usage mode, it is an unfortunate fact that a non-trivial fraction of real-world CIFs contain syntax errors. Moreover, even the most careful application programmers occasionally commit errors. For robustness and ease of use, therefore, it was an objective that to the greatest extent

possible the API must provide for error recovery and informative error reporting, especially in the context of parsing CIFs. Furthermore, to assist application developers in avoiding programming errors, it was a requirement that the API be thoroughly documented, especially with respect to preconditions and postconditions of all interface functions, and to the significance of all public data structures.

3. High-level design

The centrepiece of the API is an object model for CIF data implementing a variation on a characterization of a CIF data model by Hester (2011) (see Fig. 1). Such object models are not inherently novel, but a distinguishing feature of this one, reflected in the CIF API, is that it does not make any special provision for 'scalar' data – that is, data that are presented in CIF outside any loop construct. Instead, it requires scalars to be modelled *via* one or more single-packet loops, which simplifies both the model and the API. Within that framework, however, the CIF API preserves the presentational distinction between looped and scalar data by using up to one special loop object per data block or save frame to hold items intended for presentation in scalar form. The parser records scalar data in those special loops, the data manipulation functions ensure that they never contain more than one packet, and the formatter outputs them in scalar format. Client code can access these as loops, if desired, but the provided per-item data manipulation functions make that optional. Indeed, to a large extent they allow the distinction between scalar data and data presented in single-packet loops to be ignored altogether where it is not of interest.

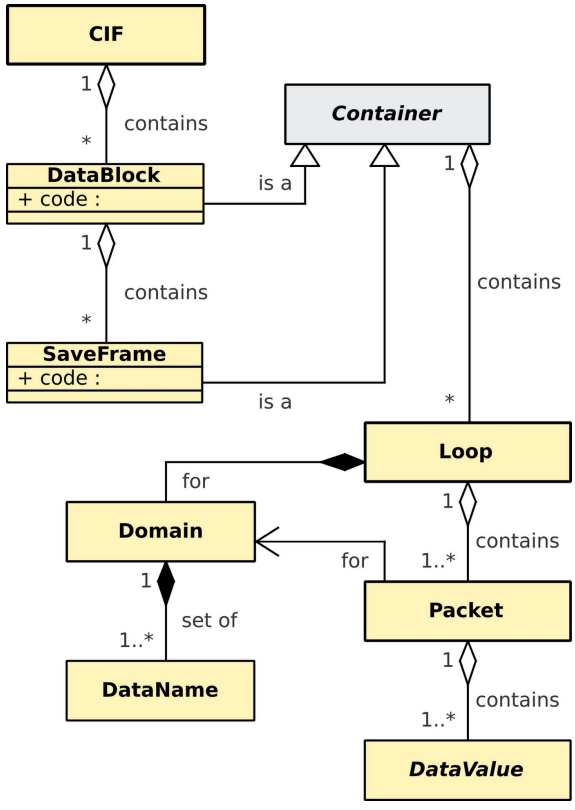


Figure 1
A UML representation of the high-level data model on which the CIF API design is based. The term 'domain' refers to the set of data names for which a loop contains data.

The API provides a representation for each object in the model: whole CIFs, data blocks and save frames, loops (encompassing the data model's domains of data names), loop packets, and values. These data types are opaque to application programmers, so that instances must be created, manipulated and destroyed *via* API functions. Furthermore, within this framework, data values of all types are presented to and accepted from client applications in a consistent form, so that for many purposes neither the API nor client applications need to be aware of the CIF data type of the data represented by any given value object.

Providing opaque data types to applications clarifies how those applications are permitted to obtain and manipulate instances, and furthermore fosters application binary compatibility with respect to updates to the API library. As long as applications rely only on documented API features, they will not be at risk of presenting data to API functions that are incorrectly formed for the API version in use, and they will sustain little risk of inadvertently corrupting CIF data. More generally, opaque data structures are an aspect of a generally object-oriented API design, in plain C. Almost every API function accepts a pointer to an opaque data structure as its first argument and can be construed as a method of the referenced data type of that argument. Of the few exceptions to that form, almost all can be characterized as object creation methods (*i.e.* constructors) accepting a double pointer by which to return a pointer to the new object.

Nearly all CIF API functions return a result code that communicates the success of the function or else the nature of its failure. All functions draw on a central set of function return codes, though there are few functions whose set of possible return values comprises more than a handful of these codes. Side effects visible to the caller are performed *via* function arguments, for example by writing a new value to the referent of a pointer argument.

The CIF API directly incorporates the basic Unicode character data type, UChar, of the International Components for Unicode (ICU) project (http://site.icu-project.org/). Although this could be criticized for exposure of implementation details, neither C90 nor C99 provides a standard data type that adequately fills this role, so the only viable alternative to UChar would have been a type defined by the CIF API itself. Direct use of UChar has the advantage, however, of affording applications a convenient entry to working with Unicode data obtained from or intended for the CIF API, in that the full ICU library, on which the reference implementation depends anyway, can be brought to bear without any additional data conversion or type casting.

4. Notable features

Certain features of the CIF API and its reference implementation are worthy of special attention. Some of them are

driven directly by the requirements discussed above; others emerged from the process of designing and implementing the API details.

4.1. Flexible parsing

The CIF API's built-in parser automatically identifies and handles both standard CIF 1.1 and standard CIF 2.0, distinguishing between them on the basis of their initial CIF-version comment, if any. Such a comment is required by CIF 2.0, so its absence is normally taken as an indication that the input is in CIF 1.1 format. The absence of a version comment in a file otherwise complying with CIF 2.0 can be overcome *via* a parser option. Options are also available to enable parsing pre-v1.1 CIFs that contain control characters not permitted in CIF 1.1 and above, albeit without flagging violations of the much smaller line-length limit that originally applied to CIFs.

The character encoding of the input is automatically detected in many cases, and an appropriate default is chosen in most other cases. For cases where automatic choice of encoding is unreliable or known to fail, however, the encoding to be used can be specified to the parser. In practice, this is an issue primarily for CIF 1.1, for which the character encoding specifications are intentionally vague and machine dependent. Well formed CIF 2.0 input, on the other hand, is always encoded according to UTF-8. Nevertheless, the parser can handle CIF 2.0 text encoded according to other schemes, too, automatically in some cases and by explicit instruction in the rest.

Many real-world CIFs contain syntax errors, often as a result of human mistakes committed while editing CIFs by hand. The CIF API's parser can often recover from such errors and extract some or even all of the valid data from affected CIFs. This behaviour is disabled by default, but can easily be enabled for all detectable errors. With somewhat more effort, a client program can arrange to be informed of all parse errors *via* a callback function, so as to choose for itself which ones justify aborting the parse or to perform any wanted bookkeeping.

4.2. Event-based parsing

The CIF API's default parsing mode constructs an object model of the input CIF data and, upon completion of the parse, provides it to the client application for inquiry and modification. Building and storing such a model has significant computational and resource overhead, however, which may be unwarranted or even unsupportable for some applications and environments. As an alternative, the CIF API's parser provides for the client application to instead be notified *via* callback functions about significant events occurring during a parse, such as the start or end of a data block or of a loop, or the parsing of a data item. An application relying on such callbacks can avoid unwanted overhead associated with building and storing an object model, instead extracting the data of interest directly during the parse and disabling construction of an object model.

Object model building and event-based parsing are not mutually exclusive, however. The callback system is still active when an object model is being built, and it can be used to influence which CIF structures are included in the model – selecting only a specific data block, for example, or skipping save frames that may be present in the input. It can also be used to implement validation procedures that can or must be performed at parse time, without the context of a whole CIF. For instance, it can be used to raise a validation error in the event that a single loop construct contains data from multiple categories (though the CIF API itself has only a very weak concept of categories), and it can be used to validate data types against those permitted for their associated data names.

Additionally, although the CIF API focuses on a semantic representation of CIF data, as opposed to a syntactic one, event callbacks can be registered with the parser that provide a syntax-level view of the input CIF. Such callbacks provide access to semantically void syntactic details such as comments, insignificant spaces and line breaks between syntactic units, and placement of keywords relative to those. The program *cif_linguist* (§5.5) uses these to preserve comments and formatting as much as possible when it transforms a CIF.

4.3. Robust comprehensive data handling

The CIF API provides comprehensive support for all data expressible in CIF format. In particular, it fully handles CIF 2.0's new features for expressing data, such as Unicode character data and CIF 2.0's List and Table data types, including nesting to any depth. Even within the area also covered by applications and libraries oriented toward CIF 1.1, however, it exhibits a few unusual capabilities.

Because every version of CIF so far released permits data values of character type to be expressed unquoted in CIF data files (subject to some restrictions), it is ambiguous whether a value that has the form of a number in fact represents a number or whether it instead represents character data. The difference is important because numeric data are subject to transformations that are inappropriate for character data. Some CIF libraries and applications have approached this problem heuristically, as early CIF specifications indicated was appropriate (Hall *et al.*, 1991) and according to published CIF 1.1 convention (Hall *et al.*, 2005): if a value is presented in a CIF delimited only by whitespace, and it can be parsed according to a given numeric syntax, then it is interpreted as a number. In practice, however, for almost every purpose, the interpretation of data items must be guided by external data definitions. Although data definitions are outside its scope, the CIF API takes an approach that can accommodate both styles: it provides a form for data values that are known to represent numbers, but it relies on the client application to control which values take that form. In particular, numeric appearing values parsed from a CIF are initially handled as strings of characters, but any value in that form is automatically coerced to numeric form (if possible) when it is operated upon by an API function that requires it to represent a numeric data value. In any event, the numeric form always retains the exact text origin-

ally specified for the value, so that can be recovered even after a coercion.

Also, CIF, being for the most part a text format, places no limits on the range or precision of numeric values expressed in instance documents. To the extent that applications want to extract numeric data from CIFs, this presents a potential problem: it is unclear which numeric format, if any, among those supported by the host environment is appropriate for handling a given datum. Moreover, even having chosen a numeric format, it is not always the case that the standard library functions can be relied upon to convert between text and numeric representation without loss of precision, even above and beyond any that may be a necessary consequence of the conversion. Thankfully, such issues are rarely of great import, as usually CIF data are consumed by systems having representational capabilities similar to those of the system by which the data were produced. Nevertheless, the CIF API makes as few assumptions as possible about the capabilities of the host environment. Although it cannot completely solve the numeric representation and text/number conversion problems, the reference implementation avoids them internally by storing numeric values in an arbitrary-precision floating point format. It provides its own conversions between that format and text, and between that format and one of the host's numeric formats. The former ensures that data read *via* the CIF API can later be rewritten exactly, with no loss of precision. The latter is not necessarily lossless for all data, but it makes use of the full precision available from the target format.

The existence of CIF conventions that treat some values differently from others based on whether they are whitespace delimited establishes *de facto* that CIF in general distinguishes whitespace-delimited values from other values. In principle, therefore, a future new convention or a data dictionary could specify different interpretations for other values depending on whether they are presented with delimiters. CIF 2.0 neither clarifies nor narrows that distinction. For complete generality, then, a CIF library must treat whether values are quoted or not as an inherent property, not merely a property of a particular external representation. The CIF API does this, and moreover provides additional validation and data type coercions around changes to that property, where appropriate. The built-in CIF formatter also respects that property to the extent possible: values flagged as unquoted will be formatted without any form of quotation if CIF syntax allows it, and values flagged as quoted will always be formatted in one of the permitted quoted forms.

On the other hand, the CIF API does not track what style of quotes is used for any given quoted value. In view of the fact that changes to values or to CIF syntax version may necessitate differing quotation style, the CIF API must always analyse each quoted value on output to determine what form of quotation can be used for it. The function employed for this purpose is directly accessible to client programs. Even when values and CIF syntax version do not change, the built-in CIF output routine may not exactly reproduce the quoting style of input values.

4.4. SQLite storage engine

The reference implementation of CIF API version 0.4 relies on a CIF storage engine implemented on top of an SQLite database (https://www.sqlite.org/index.html). The implementation relies on declarative referential integrity and on database transactionality to help it maintain the integrity of CIF data under its care, even in the event of application errors. Using a relational database for storage allows significant portions of the API implementation to be written at a higher level than otherwise would be possible, and having SQLite behind the scenes also sets the stage for future extensions revolving around direct access to the backing database.

4.5. Test suite and example programs

The reference implementation comes with an extensive test suite and several example programs. The test suite has proven invaluable during development, both for catching regressions arising from changes and updates, and for detecting environment-specific problems during testing outside the primary development environment. Although most of the example programs are too specialized to serve other than as examples or functional tests, one is a CIF 2.0 syntax checker that some may find independently useful.

5. Development directions

It is intended that the API itself (strict sense) be stable for an extended time, in that future releases will maintain backwards compatibility. There is nevertheless opportunity for extensions and for companion software, some of which are now discussed.

5.1. SQLite serialization format

The API's use of SQLite for temporary storage management opens an intriguing opportunity for future development: providing for the SQLite database file corresponding to a CIF to be retained persistently and later reopened. Such a database file would then constitute an SQLite serialization of CIF data, with the property that it could be opened and ready for use very quickly, without parsing, regardless of the volume of data within. SQLite data files are highly portable, being independent of machine details such as native word size and byte order, so such files would be suitable for exchange. This alternative might be of particular interest for speeding the handling of large libraries of CIF-based reference data or of large individual CIF data sets such as CIF dictionaries.

5.2. Alternative storage engine

The potential new uses of the SQLite storage engine notwithstanding, profiling and comparative tests show that recording data in SQLite accounts for substantially all of the built-in parser's run time. Although CIF API performance compares favourably with that of other parsers (see supporting information), an optional alternative storage engine is under consideration for a future CIF API implementation, which would exchange some of the non-essential

advantages of the SQLite engine for a speedier, purely memory-based approach. Inasmuch as the API design intentionally hides storage details from client applications, it should be possible for applications to select between storage engines without widespread code changes.

5.3. Comment handling

The current version of the CIF API focuses on a semantic representation of CIF, and therefore makes only minimal provision for CIF comments, which are semantically void. Other than passing them to a registered callback function, the parser ignores them. There is no mechanism for associating comments with data in the object model, and therefore the built-in formatter does not print any. Mechanisms for recording and outputting comments are under consideration for a future version of the API, but this presents a design challenge because CIF format does not inherently make any association between comments and nearby data.

5.4. Validation modules

Although the CIF API itself does not provide specific support for any DDL, it is capable of supporting any DDL. Inasmuch as CIF validation is an important capability for some applications, the author is considering CIF API companion modules providing for validation against DDL2, DDLm and/or DDL1 dictionaries.

5.5. Program *cif_linguist*

An experimental program, *cif_linguist*, accompanies the CIF API and is built upon it. This program performs translation between several dialects of CIF, including from CIF 1.1 to CIF 2.0 and *vice versa*, and a future version will also translate CIF data to STAR 2.0 format and to and from CIFXML (Day *et. al.*, 2011). *Cif_linguist* avoids CIF syntax that is unique to CIF 1.1, so, but for the initial 'magic code', UTF-8-encoded CIF 1.1 output of this program is well formed CIF 2.0 as well. In particular, *cif_linguist* can translate from general CIF 1.1 (or earlier) format to this CIF 2.0-friendly dialect of CIF 1.1, which can be useful for environments where both formats must be supported, or that engage in a transition from CIF 1.1 to CIF 2.0 data and infrastructure. Further development of this program to support these additional data formats, to better handle comments and formatting, and to be better supported by automated tests is a project priority.

6. Availability

The CIF API reference implementation has been built and successfully tested on Linux (CentOS 6, Debian 7), OS X 10.8 and Windows 7. Substantial effort was devoted during development to avoiding unspecified and implementation-defined behaviours in both interface and implementation, and the ease with which the code – originally developed on Linux – was ported both to Windows and to Mac tends to support the success of those efforts. There is every reason to expect that little, if any, adaptation will be required to make the API and implementation operational in any other environment that can support its dependencies on SQLite and ICU.

The CIF API and implementation are open-source software, offered for use under the terms of the GNU Lesser General Public License version 3 (Free Software Foundation, 2007). This permits the API to be used and modified by anyone, including in conjunction with programs whose own source is closed. Although the author hopes that software in general and crystallographic software in particular will be open source, the CIF API is intended to be a viable tool for everyone, regardless of their posture on free or open-source software.

The full source distribution of the CIF API and implementation is available from the author. The distribution includes source code, build system, documentation, test suite, example programs, and program *cif_linguist*. The current distribution sources can also be downloaded from COMCIFS's GitHub site (http://github.com/COMCIFS/cif_api/), and the documentation for the current version can be accessed online at http://comcifs.github.io/cif_api/.

7. Summary and conclusions

The CIF API provides a highly functional, flexible and portable software infrastructure for reading and writing CIFs and managing CIF-based data. It constitutes a ready-built mechanism for adapting to and using CIF 2.0, while not requiring abandonment of CIF 1.1 or separate support for the two CIF versions. The CIF API supports all major desktop operating systems, and it is licensed in a manner that is compatible with use in any software project.

Acknowledgements

The author wishes to thank Herbert J. Bernstein and James R. Hester for discussion and suggestions leading to the requirements for the CIF API. The author also thanks the attendees of the IUCr's 2013 Crystallographic Information and Data Management workshop for commentary and discussion that influenced the direction of the CIF API design.

References

Bernstein, H. J., Bollinger, J. C., Brown, I. D., Gražulis, S., Hester, J. R., McMahon, B., Spadaccini, N., Westbrook, J. D. & Westrip, S. P. (2016). *J. Appl. Cryst.* **49**, 277–284.
Day, N. E., Murray-Rust, P. & Tyrrell, S. M. (2011). *J. Appl. Cryst.* **44**, 628–634.
Free Software Foundation (2007). *GNU Lesser General Public License*, http://www.gnu.org/licenses/lgpl.html.
Gildea, R. J., Bourhis, L. J., Dolomanov, O. V., Grosse-Kunstleve, R. W., Puschmann, H., Adams, P. D. & Howard, J. A. K. (2011). *J. Appl. Cryst.* **44**, 1259–1263.
Hall, S. R. (1991). *J. Chem. Inf. Model.* **31**, 326–333.
Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *Acta Cryst.* A**47**, 655–685.
Hall, S. R. & Bernstein, H. J. (1996). *J. Appl. Cryst.* **29**, 598–603.
Hall, S. R. & McMahon, B. (2005). Editors. *International Tables for Crystallography*, Vol. G, *Definition and Exchange of Crystallographic Data*. Dordrecht: Springer.

Hall, S. R., Westbrook, J. D., Spadaccini, N., Brown, I. D., Bernstein, H. J. & McMahon, B. (2005). *International Tables for Crystallography*, Vol. G, *Definition and Exchange of Crystallographic Data*, edited by S. R. Hall & B. McMahon, pp. 25–36. Dordrecht: Springer.
Hester, J. R. (2006). *J. Appl. Cryst.* **39**, 621–625.
Hester, J. R. (2011). Personal communication.
International Standards Organization (1990). *ISO/IEC 9899:1990 – Information Technology – Programming Language C*. International Standards Organization, Geneva, Switzerland.
International Standards Organization (1999). *ISO/IEC 9899:1999 – Programming Languages – C*. International Standards Organization, Geneva, Switzerland.
IUCr (2011). *CIF Application Programming Interface Forum*, http://forums.iucr.org/viewforum.php?f=27.
Lin, Y. (2010). *J. Appl. Cryst.* **43**, 916–919.
Spadaccini, N. & Hall, S. R. (2012*a*). *J. Chem. Inf. Model.* **52**, 1901–1906.
Spadaccini, N. & Hall, S. R. (2012*b*). *J. Chem. Inf. Model.* **52**, 1907–1916.
W3C (1998). *Document Object Model (DOM) Level 1 Specification*, http://www.w3.org/TR/1998/Rec-DOM-Level-1-19981001/.
Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *J. Appl. Cryst.* **30**, 79–83.
XML-DEV (1998). *Simple API for XML*, http://lists.xml.org/archives/xml-dev/199805/msg00226.html.