JOURNAL OF APPLIED CRYSTALLOGRAPHY

ISSN: 1600-5767
journals.iucr.org/j

Proving the correctness of the algorithm for building a crystallographic space group

Karolis Petrauskas, Andrius Merkys, Antanas Vaitkus, Linas Laibinis and Saulius Gražulis

J. Appl. Cryst. (2022). **55**, 515–525

JOURNAL OF APPLIED CRYSTALLOGRAPHY

ISSN 1600-5767

Received 14 December 2021
Accepted 22 March 2022

Edited by J. Ilavsky, Argonne National Laboratory, USA

Keywords: space-group derivation; crystallographic algorithms; formal verification; theorem proving; Isabelle/HOL.

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

Proving the correctness of the algorithm for building a crystallographic space group

Karolis Petrauskas,[a] Andrius Merkys,[b] Antanas Vaitkus,[b] Linas Laibinis[a] and Saulius Gražulis[a,b]*

[a]Institute of Computer Science, Faculty of Mathematics and Informatics, Vilnius University, Didlaukio 47, LT-08303 Vilnius, Lithuania, and [b]Institute of Biotechnology, Life Sciences Center, Vilnius University, Sauletekio 7, LT-10257 Vilnius, Lithuania. *Correspondence e-mail: grazulis@ibt.lt

An application of formal verification (using the proof assistant Isabelle/HOL) for ensuring the correctness of scientific data processing software in the crystallographic domain is presented. The proposed process consists of writing a pseudocode that describes an algorithm in a succinct but mathematically unambiguous way, then formulating or reusing necessary Isabelle theories and proving algorithm properties within these theories, and finally implementing the algorithm in a practical programming language. Both the formal proof and the semi-formal algorithm analysis are demonstrated on an example of a simple but important algorithm (widely used in crystallographic computing) that reconstructs a space-group operator list from a subset of symmetry operators. The *cod-tools* software package that implements the verified algorithm is also presented. On the basis of the reported results, it is argued that broader application of formal methods (*e.g.* formal verification of algorithm correctness) allows developers to improve the reliability of scientific software. Moreover, the formalized (within the proof assistant) domain-specific theory can be reused and gradually extended, thus continuously increasing the automation level of formal algorithm verification.

1. Introduction

Computers are essential for scientific investigations, and software plays a crucial role in elucidation of scientific results. In crystallography, computers were used as early as 1946 (Shaffer *et al.*, 1946) to determine molecular structures. Today, research software is recognized as an important result of scientific investigation and regarded as a form of publication.

Despite this progress, algorithms in crystallography papers, as well as the majority of other crystallographic algorithm descriptions, are typically given as an informal or semi-formal human-oriented text. Moreover, in those cases where the algorithm correctness argument or proof is presented to show that the algorithm is actually producing the desired result, it is also usually expressed in a semi-formal manner, mixing formal concepts, properties and informal proof steps. This lack of rigour, leading to occasional ambiguities and potential errors, makes understanding, verification and implementation of employed algorithms more difficult. This problem becomes even more acute when we want to modify or optimize an existing algorithm.

In this paper we present an approach for demonstrating correctness of the algorithm for building a space group based on formal verification by automated theorem proving. The

Require: H – a subgroup of a finite group G
Require: g – an element of the finite group G, $g \in G$
Ensure: The list L of the operators of a subgroup $L \leq G$ without duplicates
Ensure: L contains both g and the elements of H
1: **procedure** SimpleBuilder(H, g)
▷ Build a space group generated by H and g
2: $L \leftarrow [e, h_1, h_2, \ldots, h_n]$, where $\forall i.\ h_i \in H$
3: $L_{\text{new}} \leftarrow [g]$
4: **while** L_{new} is not empty **do**
5: $g' \leftarrow \text{head}(L_{\text{new}})$
6: $L_{\text{new}} \leftarrow \text{tail}(L_{\text{new}})$
7: $L \leftarrow \text{append}(L,\ g')$
8: **for all** $h' \in L$ **do**
9: $g'' \leftarrow h' \otimes g'$
10: **if** $g'' \notin L \cup L_{\text{new}}$ **then**
11: $L_{\text{new}} \leftarrow \text{append}(L_{\text{new}},\ g'')$
12: **end if**
13: **end for**
14: **end while**
15: **return** L
16: **end procedure**

approach consists of two steps. First, the algorithm is expressed in a (language-independent) pseudocode form. Such a format employing an abstract mathematical notation and skipping certain auxiliary details (such as memory management or decomposition into files) is simple enough to be understood by a human reader but at the same time strict enough for further analysis by formal means. Next, the algorithm pseudocode is converted to the formal notation of the Isabelle/HOL (Paulson, 1986; Nipkow *et al.*, 2002) proof assistant in a separate mathematical theory describing the problem domain. The property of algorithm correctness then becomes a number of logical statements (theorems) that are proven correct by relying on various (both automatic and interactive) proof strategies and other mathematical theories available in Isabelle/HOL.

The advantages of such an approach are the following. First, by completing the proof in a proof assistant, we have formal assurance of the algorithm correctness as well as, in our experience, gaining additional insights into how and when (*i.e.* under what assumptions) the algorithm works. Second, despite the fact that construction of the Isabelle theory and completion of the correctness proof from scratch takes a considerable effort, the resulting theory and proven results can be reused for a class of similar algorithms from this problem area. Finally, expressing (the core part of) the algorithm in the pseudocode form facilitates its reusability and implementation in different specific programming languages and serves as an intermediary for its formal verification.

To apply the method outlined above, we have chosen to verify the crystallographic space-group-reconstruction algorithm (Hall, 1981; Grosse-Kunstleve, 1999) formally. On the one hand, this algorithm is well known and simple enough to make the application of formal methods to its verification tractable; on the other hand, it contains interesting group-algebraic details for which application of formal methods gives valuable insights. Moreover, a practical need to implement a version of this algorithm in the *cod-tools* package (Gražulis *et al.*, 2015; Merkys *et al.*, 2016, 2021; Vaitkus *et al.*, 2021) raised the question of whether our understanding of the algorithm as described in the published sources is sound and whether our implementation of that algorithm is correct.

The need to prove program correctness is not just of academic interest. Errors in programs have already resulted in wrong interpretation of crystallographic data and the publishing of wrong results in the scientific press (Chang, 2003). The reason for this unfortunate mistake was confusion of anomalous pairs I+ and I−, which were incorrectly converted to F− and F+, and this confusion slipped through all the tests that were employed to check the program correctness. Although introduction of formal verification cannot guarantee that programs perform as desired in absolutely all cases (for instance, the formal specification itself can be crystallographically incorrect), formal statements about data properties might have shown that converting I+ to F− (and not to F+) leads to a logical contradiction. Thus, formal methods can offer an extra tool for ensuring sound results of scientific research (Greengard, 2021).

2. Crystallographic space-group-building algorithms

In crystallographic computations, the task of building a space-group description from a list of all its symmetry operators or reconstructing a space group from just a subset of these operators is common. The need for such reconstruction arises when refining atoms on special positions (Grosse-Kunstleve & Adams, 2002) or computing symmetries of molecules and stoichiometric molecular ensembles in crystals (Gražulis *et al.*, 2015). Algorithms for computations of space groups were published originally as tools for managing space-group symbols (Hall, 1981) and later optimized for use in macromolecular refinement and other crystallographic applications (Grosse-Kunstleve, 1999). In the latter case, source code for the implementation in the C++ programming language is also available.

2.1. Problem description

Informally, the task of reconstructing a symmetry group can be specified as follows. Crystal symmetry can be described by one of the 230 pre-defined space groups (*i.e.* symmetry groups of periodic 3D Euclidean space) (Hahn, 2005). A space group itself is defined as a set of crystal symmetry operators. It has long been established (Fedorov, 1891; Schoenflies, 1892; Downward, 2015) that such a set forms an algebraic group with operator composition as the group operation. The group elements are usually provided in a shorthand notation as coordinate triplets of a general position (Fischer & Koch, 2005) (see Appendix *D* for a brief summary of the set and logic notation used throughout this paper). For instance, the space group $P4$ can be specified by giving a set of four symmetry operators $\{(x, y, z), (-x, -y, z), (-y, x, z), (y, -x, z)\}$ or in the equivalent matrix form. The operator x, y, z is the identity element of any space group. Moreover, all the symmetry space groups are required to be finite.

The space-group-reconstruction algorithm must generate, when presented with a subset of operators from some existing finite group G, a minimal subgroup containing those operators. Thus, presented with the input $\{(-x, -y, z)\}$, the algorithm should generate operators $\{(x, y, z), (-x, -y, z)\}$ describing the space group $P2$, and for the input $\{(-y, x, z), (y, -x, z)\}$ it should return all four operators describing the space group $P4$ (Hahn, 2005).

Intuitively, it is clear that multiplying (*i.e.* composing) the given input symmetry operators with each other and keeping all the obtained products should eventually result in a set closed by multiplication, and that this set should be an algebraic group. Associativity is given by the fact that all the symmetry operators are taken from an existing symmetry group (the underlying group G). Moreover, the finiteness of G guarantees that the resulting set is closed under inverses too. The group operation, however, is not always commutative, so it seems that both left and right products of all symmetry operators must be considered for inclusion.

The task can be reformulated in the following way (Grosse-Kunstleve, 1999). Suppose we have a subgroup H of the underlying symmetry group G. Initially, we choose H as

$\{(x, y, z)\}$, *i.e.* containing only the group identity element. Moreover, we select an arbitrary element, say g, from the given input set. This element is added to H, and then the resulting set is gradually extended with new elements (products of g and the elements of H) until the new minimal subgroup of G containing both H and g is built. The process is repeated until all the input elements (given symmetry operators) are covered.

As a result, the described procedure for constructing a new minimal subgroup of G for the given subgroup H and element g becomes the core part of the space-group-reconstruction algorithm. A pseudocode description of this core procedure is presented in Fig. 1.

The pseudocode for the procedure SimpleBuilderInitial employs the standard syntax for **while**, **for** and **if** statements, while $\leftarrow$ is used for variable assignments. Sets of group elements are implemented as lists, with the `head` operation returning the first list element and the `tail` operation returning the last element. The `append` operation adds the specified element to the end of the list and returns the enlarged list. The group operation is denoted as $\otimes$ in the pseudocode. Finally, the **Require** and **Ensure** headings specify the requirements (conditions) for the procedure inputs and results, respectively.

The pseudocode parts highlighted in red (*i.e.* handling the left products) are added to ensure that the algorithm works even in cases where the group operation is non-commutative. However, it will be shown later (on the basis of a formal proof) that these additions are not necessary and can be omitted, thus making the overall algorithm more efficient.

Require: H – a subgroup of a finite group G with the group operation $\otimes$
Require: g – an element of the finite group G
Ensure: L – a list (without duplicates) of the elements of a new G subgroup
Ensure: L contains both g and the elements of H
1: **procedure** SimpleBuilderInitial(H, g)
▷ Build a space group generated by H and g, initial algorithm
2: $L \leftarrow [h_1, h_2, \ldots, h_n]$ where $\forall i.\ h_i \in H$
3: $L_{\text{new}} \leftarrow [g]$
4: **while** L_{new} is not empty **do**
5: $g' \leftarrow \text{head}(L_{\text{new}})$
6: $L_{\text{new}} \leftarrow \text{tail}(L_{\text{new}})$
7: $L \leftarrow \text{append}(L,\ g')$
8: **for all** $h' \in L$ **do**
9: $g'' \leftarrow h' \otimes g'$
10: $g''' \leftarrow g' \otimes h'$
11: **if** $g'' \notin L \cup L_{\text{new}}$ **then**
12: $L_{\text{new}} \leftarrow \text{append}(L_{\text{new}}, g'')$
13: **end if**
14: **if** $g''' \notin L \cup L_{\text{new}}$ **then**
15: $L_{\text{new}} \leftarrow \text{append}(L_{\text{new}},\ g''')$
16: **end if**
17: **end for**
18: **end while**
19: **return** L
20: **end procedure**

Figure 1
The initial version of the simple space-group-builder (core) algorithm.

One might argue that one single solution of this problem, *e.g.* the implementation provided by Grosse-Kunstleve (1999), should be reused in all subsequent cases. However, using the C language code is not always desirable or even possible, while multi-language programming has not yet been achieved satisfactorily with modern programming tools. Thus, independent implementations in Perl (Wall *et al.*, 2000) or Erlang (Armstrong, 2013) might be useful in cases where these languages provide substantial benefits (for tasks involving much text processing in the case of Perl and for massively parallel tasks in the case of Erlang). Using Java has its own advantages and disadvantages. A 'pure Java' solution enhances portability to different platforms, such as mobile devices, where a Java interpreter is available, but distributing binary code is undesirable because of the variety of architectures.

2.2. General strategy

What we would like to do next is to translate the pseudocode in Fig. 1 into the language of a proof assistant and prove formally that our intuitions about the algorithm and about its different variants are correct. In particular, we would like to prove that lines 10 and 14–16 can be omitted from the algorithm and that it will still generate the correct subgroup even for non-commutative group operations.

Once the proof assistant validates the proof, we have formal assurance of the correctness at least in this formally verified step, and, in our experience, gain additional insights into how the algorithm works. After this, the pseudocode can be used to write code in any language, as we demonstrate in this paper by implementing Perl code for the analysed algorithm. While keeping the program source code and the pseudocode closely aligned does not guarantee the absence of errors, it nonetheless helps to carry over the results of the formal proof in Isabelle/HOL to the source code of the working program. Therefore, the pseudocode representation becomes the common ground between a program to be analysed and its formal verification.

2.3. Core definitions

In our formalization we rely on the standard definition of a group as a mathematical structure $(G, \otimes, \mathbf{1}, ^{-1})$, where G is the carrier set of the group, $\otimes$ is the group operation, $\mathbf{1}$ denotes the group identity element and x^{-1} (for any $x \in G$) stands for the inverse of x. We will use lower-case letters for group elements, while capital letters will be used for groups and their carrier (sub)sets. Depending on the context, a specific capital letter (*e.g.* G) can stand for a group as a structure or its carrier set. A summary of all standard logic and set notation used in this paper is given in Appendix *D*.

The notation $K \leq L$ is used to indicate that a group K is a subgroup of a group L, while $K \otimes L$ denotes a set multiplication between elements of two groups K and L, *i.e.* $K \otimes L = \{k \otimes l \mid k \in K, l \in L\}$. For some group L and element g, the notation $L \otimes g$ stands for a right coset of L and g, *i.e.* $L \otimes g = \{l \otimes g \mid l \in L\}$. A left coset $g \otimes L$ is defined

correspondingly. For some group K and its element $x \in K$, the notation $\langle x \rangle$ stands for a cyclic subgroup of K generated by element x. Finally, x^n, where x is a group element and n is a natural number, represents the nth power of x, such that $x^0 = \mathbf{1}$ and $x^n = x \otimes x^{n-1}$.

Within our formalization, we reserve the capital letters G and H to denote, respectively, the underlying group the algorithm works on and its initial subgroup, while a lower-case letter g will stand for a specific (input) element $g \in G$ that the initial subgroup H should be expanded with. We also assume that the group G is finite.

2.4. Semi-formal analysis

Before we engage ourselves in the analysis of the merits of the formally verified algorithm, let us consider if a more straightforward form of the algorithm can be constructed that is easier to show to be correct. This can be achieved by adding not just the element g'' (the result of a right multiplication of h' by g') but also the left product $g' \otimes h'$. The resulting algorithm is provided in Fig. 1, with extra statements highlighted in red. That this algorithm works as desired, *i.e.* that it produces a group containing H and g, can be seen by examining all possible products of elements $h \in H$ with the element g.

We introduce the following definition to shorten the semi-formal proof of the algorithm in Fig. 1:

Definition 1. A group G element p that has been expressed as a product of two elements from the list L is said to be 'considered for addition', if at some stage of the algorithm execution it was inserted into the list L_{new} at line 12 or line 15 of the algorithm in Fig. 1.

Theorem 1. The algorithm in Fig. 1 always terminates for a finite group G. After termination, it produces a group containing all elements of H and the element g.

Proof. First we show the termination of the algorithm (i), and we then show that the result of the algorithm is a group by proving definiteness of the group operation (ii) as well as closure under both the group operation (iii, iv, v) and the inverse operation (vi, vii).

(i) Since the group G (from which the element g is taken) is finite, the element g has a finite order, *i.e.* $\exists k \in \mathbb{N} \cup \{0\} : g^{k+1} = \mathbf{1}$, where $\mathbf{1} \in H$ is a unit (neutral) element. Since H is also finite, the results of repeated multiplication of elements $h \in H$ and their products with g will eventually start repeating, and will not be added to the list L_{new}. Thus, the algorithm necessarily terminates.

(ii) The definiteness of the multiplication operation for all elements under consideration and the associativity of this operation are ensured by the fact that all elements are taken from some larger group G.

(iii) Let us check if a product of any two elements p and q from L is again in L. If both p and q belong to H, then $p \otimes q \in H$ since H is, by assumption, a subgroup of G. Therefore $p \otimes q \in H \leq L \Rightarrow p \otimes q \in L$.

(iv) Now let us suppose that either p or q, or both of them, are not in H. In that case, one of p or q was added earlier to the list L, and the second was considered for addition. Let us assume, without loss of generality, that p was added earlier. This means that at some iteration of the **while** loop q was multiplied from the right with all elements of L (line 8 of the algorithm in Fig. 1), including the element p. Thus, the product $p \otimes q$ was checked and either found to be already in L or found to be not in L and therefore considered for addition at line 11. But all elements that were considered for addition at some stage were eventually added to the list L in lines 5–7; therefore, in any of the two possible cases, $p \otimes q$ was added to the output list L.

(v) Likewise, under the same conditions as listed in step (iv), the product $q \otimes p$ was checked at lines 10 and 14–16 and either found to be in L or considered for addition. In both cases it was eventually added to L. The conclusion from this step and step (iv) is that, for any two elements from L, their product is in L.

(vi) We now need to check that, for every element $x \in L$, its inverse x^{-1} is also in L. If the element x was from the original group H, $x \in H$, then $x^{-1} \in H$ since H is a group. Let us assume that x is not from the original elements of H. In that case x was considered for addition at some stage of the algorithm execution, either at line 12 or at line 15. It was then added to L at some later iteration and multiplied by all elements in L, including the newly added x. In particular, x^2 is produced and checked at some stage, and is either found to be already in L (lines 11 and 14) or considered for addition (lines 12 and 15). Thus, in any case, it is included in L.

(vii) Now assuming x^{n-1} and all lower powers of x are in L, let us show that x^n is in L. The element x^{n-1} may have been considered for addition at some stage, in which case it was multiplied by the x that was present in L, and the resulting x^n was considered for addition, eventually being added to L. Alternatively, x^{n-1} was never considered for addition, in which case it must have been present in the list L from the very beginning of the algorithm execution, which is only possible if $x^{n-1} \in H$. This means that the product $x \otimes x^{n-1} = x^n$ was already examined when multiplying elements of L by x and either found in L or considered for addition. In all cases eventually $x^n \in L$.

Since all elements in L have finite power, for every x in L there exists such $k \in \mathbb{N} \cup \{0\}$ that $x^{k+1} = \mathbf{1}$, where $\mathbf{1}$ is the unit element of the group. This means that $x^{k+1} = \mathbf{1} = x^k \otimes x \Rightarrow x^k = x^{-1}$, demonstrating that for every element x from the list L its inverse is in the list L.

Since we see that for every p and q from L their products $p \otimes q$ and their inverses p^{-1} (and q^{-1}) are in L, the unit element $\mathbf{1}$ is in L, and the multiplication operation is associative, we conclude that the elements of L form an algebraic group. □

The algorithm in Fig. 1 is thus correct; it contains, however, extra multiplications that might be not necessary (and indeed they are not, as the formal proof using the Isabelle system shows). These extra group multiplications are not just an

Require: H – a subgroup of a finite group G
Require: g – an element of the finite group G, $g \in G$
Ensure: The list L of the operators of a subgroup $L \leq G$ without duplicates
Ensure: L contains both g and the elements of H
1: **procedure** SIMPLEBUILDER(H, g)
▷ Build a space group generated by H and g
2: $L \leftarrow [e, h_1, h_2, \ldots, h_n]$, where $\forall i.\ h_i \in H$
3: $L_{\text{new}} \leftarrow [g]$
4: **while** L_{new} is not empty **do**
5: $g' \leftarrow \text{head}(L_{\text{new}})$
6: $L_{\text{new}} \leftarrow \text{tail}(L_{\text{new}})$
7: $L \leftarrow \text{append}(L,\ g')$
8: **for all** $h' \in L$ **do**
9: $g'' \leftarrow h' \otimes g'$
10: **if** $g'' \notin L \cup L_{\text{new}}$ **then**
11: $L_{\text{new}} \leftarrow \text{append}(L_{\text{new}},\ g'')$
12: **end if**
13: **end for**
14: **end while**
15: **return** L
16: **end procedure**

Figure 2
The optimized simple space-group-builder (core) algorithm.

inelegance: our tests show that adding these extra statements just for the sake of easier proof costs us about 20% extra run time, as demonstrated in Appendix *C*. It is thus highly desirable to remove them, but only if the algorithm can be shown to work correctly afterwards. The simple proof of the closure under the operation is no longer applicable to the optimized algorithm in Fig. 2, however, since multiplication is only one-sided, and the groups under consideration are not necessarily commutative (and H is not always guaranteed to be a normal subgroup, so we cannot even assume that H commutes with g as a subgroup). Thus, we cannot immediately make the induction step (v). If, however, we cannot guarantee that the generated operator set H is closed under left- and right-sided multiplication, we cannot guarantee that H is an algebraic group.

The formal proof in Isabelle shows, though, that even without the commutativity assumption the simplified (optimized) algorithm stays correct. In particular, Lemmas 3 and 4 (Appendix *A*) give us the tools to demonstrate that all group elements are eventually produced. After this proof, the algorithm implementation can be safely simplified (and, as expected, all tests give the same reconstructed groups for both implementations).

3. Formal verification

3.1. Optimized algorithm

In this section we present a formal verification of the presented space-group-builder algorithm. We start by putting forward a high-level formalization of the algorithm and its essential properties, followed by proof sketches of those properties. At the end of the section, Theorem 2 formulates and proves the main correctness property of the algorithm. All of the verified statements were fully formalized and proven within the Isabelle/HOL proof assistant. In Section 3.4 we provide guidelines for mapping between the presented proof sketches and the corresponding formal machine-checked proofs in Isabelle/HOL.

3.2. Additional definitions

Using the core definitions, the space-group-builder algorithm can be represented as the following high-level function:

$$B(H, g) = \begin{cases} H & \text{if } g \in H, \\ R(H, \{g\}) & \text{if } g \notin H, \end{cases} \tag{1}$$

where B stands for the space-group-builder function, H is its initial subgroup and g is the element H has to be expanded with. Finally, the function R represents the recursive part of the algorithm and is defined as

$$R(L, N) = \begin{cases} L & \text{if } N = \emptyset, \\ R(L', N') & \text{if } N \neq \emptyset, \text{ where} \\ & \quad L' = L \cup \{x\}, \quad x \in N, \\ & \quad N' = [N \cup (L' \otimes x)] \setminus L'. \end{cases} \tag{2}$$

The recursive definition of R is to be understood as follows. If the set N is empty, the value of $R(L, N)$ is simply the given set L; otherwise, two new sets are considered: the set L' contains all the elements of L and one arbitrary element x from N, and the new set N' is formed by taking all the elements from the former set N, including all right products of the elements from L' with the chosen element x, and excluding all the elements that are in L'. Essentially this means that the element x is 'transferred' from N to L (thus obtaining L'), and all new right products of L' and x are added to N to construct the new N'. The value of R on these new sets is then computed. This recursive definition captures the behaviour of the algorithm in Fig. 2 at lines 4–14. Since the expression $R(H, \{g\})$ (the generated subgroup) will be essential in the proofs given below, we introduce a shorthand name for it: $R_0 = R(H, \{g\})$.

The presented formalization is more general than the algorithm described in Fig. 2, because here the order in which the elements are moved from the set N to L is arbitrary. To formally demonstrate the correctness of such a generalized algorithm, we need to prove that, for any finite group G with initial subgroup H and arbitrary group element g, the proposed algorithm generates a minimal subgroup that includes both H and g. This property can be formulated as

$$\begin{aligned} &|G| < \infty \wedge H \leq G \wedge g \in G \ \Rightarrow \\ &\quad B(H, g) \leq G \wedge H \cup \{g\} \subseteq B(H, g) \ \wedge \\ &\quad \forall J \leq G : H \cup \{g\} \subseteq J \Rightarrow B(H, g) \subseteq J. \end{aligned} \tag{3}$$

Here $|G|$ stands for the cardinality of the carrier set of the underlying group G.

3.3. Sketch of the proof

In this section we provide a sketch of the proof of the correctness of the space-group-builder algorithm, *i.e.* the proof of the property (3). The complete formal machine-checked proof is provided as an Isabelle/HOL theory in a

supplementary file `CrystalSpaceGroupBuilder.thy`. The provided file is compatible with Isabelle version 2021-1.

The main difficulty in proving the property (3) is to show that the generated subgroup R_0 is closed under the group operation. The algorithm involves only the right coset operation and, therefore, it is not obvious how elements belonging to the left cosets are generated to be included in the resulting subgroup while not considering H to be a normal subgroup. To facilitate the proof, we introduce a set of group elements H_g defined in the following way:

$$H_g = \bigcup_{n=0}^{\infty} H_g^n, \quad \text{where } H_g^n = \begin{cases} H & \text{for } n = 0, \\ H \otimes g \otimes H_g^{n-1} & \text{for } n > 0. \end{cases} \tag{4}$$

We later show that this constructed set H_g is equal to our generated set R_0. This gives us extra means to prove that R_0 is closed under the group operation, by simply replacing it with H_g and exploiting the way H_g was constructed.

Relying on lemmas from Appendix *A*, the main property of the algorithm (3) can be proven as the following theorem.

Theorem 2. For any initial subgroup H of the finite group G and any element g ∈ G, B(H, g) is a minimal subgroup of G containing both H and g, i.e. B(H, g) ≤ G, H ∪ {g} ⊆ B(H, g) and $\forall J \leq G : H \cup \{g\} \subseteq J \Rightarrow B(H, g) \subseteq J$.

Proof. We consider two cases: when $g \in H$ and $g \notin H$. In the first case the proof is immediate, because $B(H, g) = H$, $H \leq G$ and $H \cup \{g\} = H$.

In the second case, *i.e.* when $g \notin H$, we have that $B(H, g) = R_0$. Therefore, we need to show that (*a*) R_0 is a subgroup of G, (*b*) it contains both H and g (*i.e.* $H \cup \{g\} \subseteq R_0$), and (*c*) R_0 is a minimal one of such G subgroups (*i.e.* $\forall J \leq G : H \cup \{g\} \subseteq J \Rightarrow R_0 \subseteq J$).

To show (*a*), we need to prove in turn that (i) R_0 is a subset of G, (ii) it contains $\mathbf{1}$, (iii) it is closed under the group operation and (iv) it is closed under the inverse operation. We now consider those cases (i)–(iv) one by one:

(i) We have $R_0 \subseteq G$, since all the elements of R_0 are produced by applying the group operation on elements in G.

(ii) We have $\mathbf{1} \in R_0$, because $\mathbf{1} \in H$ and $H \subseteq R_0$ by Lemma 6.

(iii) We have that R_0 is closed under the group operation, because $R_0 = H_g$ by Lemma 11 and H_g is closed under the group operation by Lemma 4.

(iv) We have that R_0 is closed under the inverses, since, for each $x \in R_0$, we have $\langle x \rangle \subseteq R_0$ by Lemma 7, and for finite groups $x^{-1} \in \langle x \rangle$.

To show (*b*), we need to prove $H \cup \{g\} \subseteq R_0$. Since $R_0 = R(H, \{g\})$, applying Lemma 6 for $R(H, \{g\})$ immediately gives us the desired result $H \cup \{g\} \subseteq R_0$.

Finally, we show (*c*). We have $R_0 = H_g$ by Lemma 11 and $\forall J \leq G : H \cup \{g\} \subseteq J \Rightarrow H_g \subseteq J$ by Lemma 5. Therefore R_0 is a minimal subgroup containing H and g. □

3.4. Proof in Isabelle/HOL

The formalization of the algorithm and the machine-checked proof of its correctness property (3) is done in Isabelle2020/HOL. The formalization and the proofs are designed as a theory `CrystalSpaceGroupBuilder` that extends the `group` and `subgroup` theories from the standard `HOLAlgebra` theory and introduces an assumption for the group to be finite.

The overall procedure $B(H, g)$ is formalized as a definition `builder` and the recursive part of the procedure $R(L, N)$ is stated as a function `builderRec`. In order to proceed with the proofs of the termination and other properties, preconditions for the `builderRec` function are introduced as a definition `builderPre`, requiring L and N to be disjoint subsets of G. Lemma `builderRec_inv_preCnd` shows that this precondition is maintained as an invariant by `builderRec`. Theorem 2 is proven in Isabelle as theorem `builder_produces_subgroup` together with lemmas `builderRec_produces_subgroup` and `builderRec_m_closed`.

Incidentally, the formal Isabelle proof for the algorithm in Fig. 2 could be adapted to verify the algorithm in Fig. 1 as well. The closure under the group operation, the main property considered in Theorem 2, for the less efficient algorithm is formulated and proven as Lemma 12 in Appendix *B*. This lemma makes the definition of H_g (4) along with Lemmas 3, 4 and 8–11 not needed in this case. Its formal proof in Isabelle/HOL is provided in the supplementary file `CrystalSpaceGroupBuilderLR.thy`.

4. Results

The formal proof carried out in the Isabelle/HOL proof assistant demonstrated that the theory derived from the pseudocode presented in Fig. 2 allowed us to prove the desired property of the algorithm, namely that the list of operators produced by the algorithm is always a group. This theory can now be re-used to validate further versions of the algorithm. A slight modification of this Isabelle theory, for instance, allowed us to formally verify that the semi-formal proof in Section 2.2 and the algorithm presented there are correct as well. In the future, since the analysed SimpleBuilder procedure forms the core of this optimized algorithm, this theory can be used to formally verify further optimizations of the space-group-builder algorithm, for example, factorization of a space group into inversion centre, centring translations and representative rotations (Grosse-Kunstleve, 1999).

With the formal assurance that the algorithm in Fig. 2 is correct, we can now remove additional group multiplications from the initial algorithm presented in Fig. 1. Checking both implementations, we find that this modification alone improves the run time by about 20% (see Appendix *C* for run-time data). This is a worthwhile improvement, given also that the more efficient algorithm is shorter and simpler.

The verified algorithm was used to implement and double-check an optimized space-group-builder algorithm in the *cod-tools* package (Gražulis *et al.*, 2015; Merkys *et al.*, 2016, 2021; Vaitkus *et al.*, 2021). This implementation is now available for multiple uses such as detection of special positions, validation of symmetry operators in CIFs, checking inputs for

the Crystallography Open Database and determining symmetry groups of molecules that are on special positions (Gražulis *et al.*, 2015), among other applications. The implementation has been also shown to be extremely useful in the derivation of chemical structures from crystallographic data (Quirós *et al.*, 2018).

5. Discussion

Ideally, we would like to perform all steps of program generation from the original specification to the working machine code automatically, so that the possibility of human error is reduced. Unfortunately, the current state of the art in compiler construction does not permit full formal verification of program code written in an arbitrary programming language. Formal verification of algorithms is implemented as an extension over the C# language (Spec#; Barnett *et al.*, 2005) and a subset of the Ada language (SPARK; Chapin & McCormick, 2015). However, even in these cases full verification automation for more complicated algorithms is rarely achievable (*i.e.* proofs of some verification conjectures are returned as not completed, thus leaving uncertainty about the overall correctness). Moreover, the language of verification conjectures is typically quite limited, covering just standard mainstream mathematics and logic. Better automation can be expected for domain-specific languages, where dedicated provers can rely on pre-defined and proven beforehand statements from a specific problem area. Our approach can be seen as a step in that direction, relying on a mathematical theory (built in the Isabelle/HOL proof assistant) for a particular class of crystallographic algorithms. The theory can be gradually extended and easily reused, thus continuously increasing the automation level for formal algorithm verification.

In this approach, a mathematically rich algorithm is first encoded in human-readable procedural pseudocode, using mathematical notation (as in Fig. 2). From this notation, two codes are derived by a human programmer using conventional programming techniques: one is a functional code in a proof assistant language (for example Isabelle/HOL, which we have used in our case) and the other is a procedural language code suitable for compilation and execution on computers (in our case, Perl code is used). In fact, the Perl code in our case was written before the pseudocode; so, actually, the opposite direction of information flow is also possible. An important point, however, is that the code transformations done manually are plain and simple enough that the probability of mistake made by a human programmer is minimized. All complexity should be offset to the formal code in the proof assistant's language, which is then verified rigorously using computer-assisted formal methods.

From a crystallographer's perspective, a proof assistant like Isabelle provides the means to verify an existing proof automatically. A proof is encoded in a formal language and all logic inferences are traced back to the basic axioms, thus leaving no room for implicit assumptions, misunderstandings or human errors in this process. Even though the language of the proof assistant is rich and requires a certain effort to learn, it looks like yet another programming language (like Fortran, Prolog or C++), with which crystallographers engaged in crystallographic software development are surely familiar. Once the formal proof is created by a mathematician and verified in a proof assistant, a programming crystallographer can double-check their source code against the assumptions and consequences of the proven statements (lemmas and theorems), thus increasing the likelihood that the program under consideration functions as expected.

The main intellectual challenge is to construct a formal proof in a proof assistant after the required property to be proven has been (semi-formally) formulated by a crystallographer. Here the contribution of a skilled mathematician is crucial. Of course, there is no reason why a crystallographer could not master the formal proof construction methods, like many crystallographers have mastered programming skills in regular programming languages, but constructing a proof is arguably a greater challenge than verifying the proof and applying its results to check the program code. Unlike regular programming, which requires certain knowledge of the underlying computing engine, constructing a proof in Isabelle would require deepening one's understanding of the underlying mathematical logic and some experience in formal proof methods and functional programming languages. Since there is no general algorithm to construct a proof for an arbitrary theorem, practical skills in choosing a productive proof strategy become important. More details about the proof environment and proof construction process can be found in Appendix *E*.

Of course, formal verification methods are not a panacea. They only work provided that the original formal specification is formulated correctly and that there are no errors in the proof assistant itself. The latter is very likely, since proof assistants are reviewed and used by many mathematicians, so undergo careful scrutiny, and the former, although possible, should be minimized by keeping pseudocode simple and aligned with the proof assistant theorem formulation as closely as possible.

The method presented in this paper is fairly general and can be applied to any crystallographic or scientific algorithm. More complicated algorithms, of course, will have to be modularized, calling simpler algorithms as subroutines, and their formal verification should use Isabelle sub-theories describing those subroutines. We hope that with time, as proof assistants become more 'user friendly', more algorithms can be analysed in the proposed way, and formal proofs could become a standard method of presentation for scientific algorithms and a way to reason about them.

6. Conclusions

We have demonstrated that formal verification methods can be applied to prove correspondence to specifications for crystallographic algorithms. The formal verification techniques using Isabelle and HOL appear to be useful in practical crystallographic programming, even though complete automatic verification of the program source code is not feasible

for most programming environments. Algorithms checked with formal methods can be used in actual implementations, as demonstrated by our implementation of the discussed space-group-builder algorithm in the *cod-tools* software package.

APPENDIX A
Auxiliary theorems

This section contains the proofs of several lemmas that are used to prove the main property (3) of the algorithm. Specifically, we show that set H_g is closed under the group operation, function $R(L, N)$ is monotonically increasing with respect to its arguments, set R_0 contains all the cyclic subgroups generated by its elements (and, therefore, it is closed under the inverses), R_0 also includes the left and right cosets represented by its elements, and H_g is actually equal to the resulting set R_0.

We start with an essential property of the H_g^n sets that directly follows from the way they are constructed.

Lemma 3. For any natural numbers m and n, $H_g^m \otimes H_g^n = H_g^{m+n}$.

Proof. We prove the goal by induction on m. For the base case $m = 0$, we need to show that $H_g^0 \otimes H_g^n = H_g^n$. Since $H_g^0 = H$ and H is closed under the group operation (hence $H \otimes H = H$), we can rewrite the goal from the left-hand side: $H_g^0 \otimes H_g^n = H \otimes H \otimes g \otimes H_g^{n-1} = H \otimes g \otimes H_g^{n-1} = H_g^n$.

For the induction step $m = i + 1$, we assume $H_g^i \otimes H_g^n = H_g^{i+n}$ and need to show that $H_g^{i+1+n} = H_g^{i+1} \otimes H_g^n$. Since $H_g^{i+1} = H \otimes g \otimes H_g^i$, we can rewrite the goal from the right-hand side as follows: $H_g^{i+1} \otimes H_g^n = H \otimes g \otimes H_g^i \otimes H_g^n = H \otimes g \otimes H_g^{i+n} = H_g^{i+1+n}$. □

Now we can easily prove that H_g is closed under the group operation.

Lemma 4. H_g is closed under the group operation, i.e. $\forall x, y \in H_g : x \otimes y \in H_g$.

Proof. Assume $x \in H_g$ and $y \in H_g$. According to (4), there exist some natural numbers m and n such that $x \in H_g^m$ and $y \in H_g^n$. This means that $x \otimes y \in H_g^m \otimes H_g^n$. Since $H_g^m \otimes H_g^n = H_g^{m+n}$ by Lemma 3 and $H_g^{m+n} \subseteq H_g$ for any m and n, this immediately proves that $x \otimes y \in H_g$. □

Moreover, H_g is contained in all G subgroups that include H and g.

Lemma 5. H_g is included in all G subgroups containing H and g, i.e. $\forall J \leq G : H \cup \{g\} \subseteq J \Rightarrow H_g \subseteq J$.

Proof. We assume J is a subgroup containing both H and g. To prove $H_g \subseteq J$, we need to show that, for any $x \in H_g$, we have that $x \in J$. For each $x \in H_g$, by definition of H_g in (4), there exists n such that $x \in H_g^n$. Thus, to finish the proof, it is sufficient to show that $\forall n : H_g^n \subseteq J$. We proceed by induction on n.

For the base case, we have $H_g^0 = H \subseteq J$ by the lemma assumption. For the inductive step, we additionally assume $H_g^n \subseteq J$ and prove that $H_g^{n+1} \subseteq J$. We have that $H_g^{n+1} = H \otimes g \otimes H_g^n$ by definition (4). H, g and H_g^n are all in J according to the assumptions. Moreover, J is a subgroup (thus, it is closed under the group operation). Therefore, $H_g^{n+1} \subseteq J$. □

Next, we proceed with one of the essential properties of $R(L, N)$ by showing that it represents a monotonically increasing function:

Lemma 6. For any disjoint G subsets L and N, the resulting set $R(L, N)$ contains both L and N, i.e. $L \cup N \subseteq R(L, N)$.

Proof. It is easy to show that L and N stay disjoint in the recursive definition of $R(L, N)$. The remaining proof is by induction on $R(L, N)$. The base case [when $N = \emptyset$ and $R(L, N) = L$] is trivial as $L \cup N = L = R(L, N)$.

For the inductive case [when $N \neq \emptyset$ and $R(L, N) = R(L', N')$], we assume that $L' \cup N' \subseteq R(L', N')$ and need to show that $L \cup N \subseteq R(L', N')$. From the definition (2) and the fact that L and N are disjoint, we see that $L' \cup N'$ can be simplified to $L \cup N \cup (L' \otimes x)$. Therefore, since $L \cup N \subseteq L' \cup N'$ and $L' \cup N' \subseteq R(L', N')$, by transitivity we get $L \cup N \subseteq R(L', N')$. □

Next we focus on the generated set R_0 and prove that R_0 contains all the cyclic subgroups generated by its elements as well as left and right cosets represented by its elements.

Lemma 7. For all elements $x \in R_0$, the corresponding cyclic groups $\langle x \rangle$ are in the result, i.e. $\forall x \in R_0 : \langle x \rangle \subseteq R_0$.

Proof. It is easy to show that, for all $x, y \in R_0$, we have either $x \otimes y \in R_0$ (if $x \in L$ when moving y from N to L') or $y \otimes x \in R_0$ (if $y \in L$ when moving x) or both were in H.

Let us assume $x \in R_0$. Then we prove $\langle x \rangle \subseteq R_0$ by showing that any power $x^n \in R_0$ by induction on n. For the base case $n = 0$, $x^0 = \mathbf{1} \in R_0$ since $\mathbf{1}$ belongs to the initial subgroup $H \subseteq R_0$.

For the inductive case n, we assume $x^n \in R_0$ and need to prove $x^{n+1} \in R_0$. As noted above, either $x \otimes x^n \in R_0$ or $x^n \otimes x \in R_0$. In any case, $x^{n+1} \in R_0$. As a result, proof by induction shows that $\forall n : x^n \in R_0$ and therefore $\langle x \rangle \subseteq R_0$. □

Lemma 8. For the given initial subgroup H, the algorithm generates all the right cosets represented by the elements in R_0, i.e. $\forall x \in R_0 : H \otimes x \subseteq R_0$.

Proof. Initially $L = H$. H always remains in L by Lemma 6. Since H is a group, it is closed under the group operation and therefore $\forall x \in H : H \otimes x = H \subseteq R_0$.

Additionally, we have to consider elements moved from N to L. For each such element x, $H \otimes x$ will be contained in $N' \cup L'$ because $L' \otimes x$ is added to $L' \cup N'$ and $H \subseteq L'$. Thus $H \otimes x \subseteq L' \otimes x \subseteq R_0$. □

Now we prove the dual result about the left cosets. Here we cannot proceed along the lines of Lemma 8, because the algorithm only uses the group operation in such a way that H is always on the left side (*i.e.* producing right cosets for the elements of H).

Lemma 9. The algorithm generates all the left cosets represented by the elements in R_0, i.e. $\forall x \in R_0 : x \otimes H \subseteq R_0$.

Proof. Assume that $x \in R_0$. According to Lemma 8, $H \otimes x \subseteq R_0$ for each $x \in R_0$. Moreover, from standard group theory, we have $x \otimes H = \{y^{-1} : y \in H \otimes x^{-1}\}$.

Since $x^{-1} \in \langle x \rangle$, we have that $x^{-1} \in R_0$ by Lemma 7. Moreover, this means that $H \otimes x^{-1} \subseteq R_0$ by Lemma 8. For any y such that $y \in H \otimes x^{-1}$, the element $y^{-1} \in \langle y \rangle$ and thus $y^{-1} \in R_0$ again by Lemma 7. As a result, we prove that $x \otimes H \subseteq R_0$. □

The above lemmas do not state that all the cosets needed to form the generated subgroup are present in the result, *i.e.* there could be a coset that is not represented in the result by any element and is thus missing in R_0.

Next we proceed with showing that the generated group R_0 is equal to the constructed set H_g. First we show that for each $x \in R_0$ there exists $g \otimes x \in R_0$ and thus $H_g^n \subseteq R_0 \Rightarrow g \otimes H_g^n \subseteq R_0$.

Lemma 10. For each element $x \in R_0$, the result contains x multiplied by g from the left, i.e. $\forall x \in R_0 : g \otimes x \in R_0$.

Proof. Let us consider three alternative cases: $x \in H$, $x = g$ and $x \in R_0 \setminus (H \cup \{g\})$. For the first case, we have $g \in R_0$ by Lemma 6 and its left coset $g \otimes H$ is contained in R_0 by Lemma 9; thus $g \otimes x \in R_0$.

Next consider that $x = g$. According to Lemma 7, the result R_0 contains $\langle x \rangle$ for all $x \in R_0$. Thus it contains $g^2 = g \otimes g = g \otimes x$.

Let us consider the remaining case when $x \in R_0 \setminus (H \cup \{g\})$. The element g is added to the resulting list L' by the first algorithm iteration because it is the only element in the original set N. All the other elements are added to the resulting set L' after it, as elements of the cosets $(L \cup \{x\}) \otimes x$. Since $g \in L$ at this point, the elements $g \otimes x$ are added to $L' \subseteq R_0$. □

Now we are ready to prove that the sets R_0 and H_g are actually equal.

Lemma 11. The set R_0 is equal to the constructed set H_g, i.e. $R_0 = H_g$.

Proof. First we show that $R_0 \subseteq H_g$. In order to do that, we start by proving a more general property $\forall L, N : L \cup N \subseteq H_g \Rightarrow R(L, N) \subseteq H_g$ by induction on $R(L, N)$. For the base case (when $N = \emptyset$), $R(L, N)$ is equal to L, and the property trivially follows from its assumption.

For the inductive case, we assume $L \cup N \subseteq H_g$. Moreover, we have the inductive assumption that the property is preserved by the recursive step. To finish the proof, it is sufficient to show that $L' \cup N' \subseteq H_g$. The only elements added to $L' \cup N'$ are the coset elements $(L \cup \{x\}) \otimes x$. Since $L \subseteq H_g$ and $x \in H_g$, then $L \cup \{x\} \subseteq H_g$ and by Lemma 4 $(L \cup \{x\}) \otimes x \subseteq H_g$. This implies that $L' \cup N' \subseteq H_g$.

$R_0 \subseteq H_g$ is a special case of the proven general property, provided we can show that $H \cup \{g\} \subseteq H_g$. We do that by proving separately that $H \subseteq H_g$, because $H = H_g^0 \subseteq H_g$, and $\{g\} \subseteq H_g$, because $g = \mathbf{1} \otimes g \otimes \mathbf{1} \in H_g^1 \subseteq H_g$.

Next we show that $H_g \subseteq R_0$ by induction on H_g^n. For the base case ($n = 0$), we have $H_g^0 = H \subseteq R_0$ by Lemma 6.

For the induction step ($n = i$), we have the induction assumption $H_g^i \subseteq R_0$ and need to prove that $H_g^{i+1} \subseteq R_0$. By Lemma 10 we have $g \otimes H_g^i \subseteq R_0$ and then we get $H \otimes g \otimes H_g^i \subseteq R_0$ by Lemma 8. Since $H \otimes g \otimes H_g^i = H_g^{i+1}$, we conclude that $H_g^{i+1} \subseteq R_0$. □

APPENDIX *B*
Proof of closure under the group operation for the initial version of the algorithm

Lemma 12. R_0 produced by the algorithm in Fig. 1 is closed under the group operation, i.e. $\forall x, y \in R_0 : x \otimes y \in R_0$.

Proof. We assume $\forall x, y \in L : x \otimes y \in L \cup N$ and proceed by induction on $R(L, N)$. For the base case, when $N = \emptyset$ we have $L \cup N = L$ and thus by assumption $x \otimes y \in L = R(L, N)$.

For the induction step we need to show $\forall x, y \in L' : x \otimes y \in L' \cup N'$, where $L' = L \cup \{n\}$ with $n \in N$ as defined in the definition (2) and $N' = (N \cup (L' \otimes n) \cup (n \otimes L')) \setminus L'$ is adjusted to represent the algorithm in Fig. 1. If $x, y \in L$, the result is immediate because $x \otimes y \in L \cup N$ by assumption and $L \cup N \subseteq L' \cup N'$ by Lemma 6. Otherwise $x = n \wedge y \in L$ or $x \in L \wedge y = n$ or $x = y = n$. In all of these cases $x \otimes y \in L' \cup N'$ because $L' \otimes n$ and $n \otimes L'$ are both added to the set N'. □

APPENDIX *C*
Run times of the different space-group-builder algorithm versions

To evaluate the speed-up caused by replacing the algorithm in Fig. 1 with the one in Fig. 2, we have performed five successive runs, each run comprising 100 reconstructions from all operators of the space group 230 ($Ia\bar{3}d$) on an unloaded computer with 7.7 Gi RAM (2.6 Gi RAM used) and an Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60 GHz running at 1.7 GHz, under Linux Mint 20.1 ×86_64 OS, using the distribution's default Perl interpreter (v5.30.0). Both algorithms were implemented in a single `insert_symop` method from the `SimpleBuilder.pm` file which is provided as part of supplementary file `jl5034sup2.zip`. Switching between the two implementations is done by setting the `COD_TOOLS_USE_DOUBLE_MULT_SIMPLE_BUILDER` environment variable to false (for the algorithm in Fig. 2) or true (for the algorithm in Fig. 1). The run time for the algorithm in Fig. 1 was 81.08 ± 0.18 s, while that for the algorithm in Fig. 2 was 63.78 ± 0.17 s (21% run time improvement).

APPENDIX *D*
A brief summary of logic and set notation

A concise but sufficiently precise introduction to set theory can be found in the book by Halmos (1960), a classical text

Table 1
Summary of logic and set notation used in the article.

Q and R are arbitrary statements, and A and B are arbitrary sets. The symbol $\neg$ binds stronger than other symbols, *i.e.* expression $\neg R \wedge Q$ is interpreted as $(\neg R) \wedge Q$, not as $\neg(R \wedge Q)$. Parentheses are used to indicate operation precedence where they are needed to avoid ambiguity.

Symbol	Meaning
iff	If and only if.
$\stackrel{\text{def}}{=}$	Equal by definition; the left side is defined to be equal to the right side.
$\wedge$	Logical 'and'; $Q \wedge R$ is true iff both Q and R are true.
$\vee$	Logical 'or'; $Q \vee R$ is true iff at least one of Q or R is true.
$\neg$	Logical 'not'; $\neg Q$ is true if Q is false and is false if Q is true.
$\Rightarrow$	Logical implication; $Q \Rightarrow R$ is true iff R is true or Q is false, *i.e.* $Q \Rightarrow R \stackrel{\text{def}}{=} R \vee \neg Q$.
$\Leftrightarrow$	Logical equivalence; $Q \Leftrightarrow R$ is true iff R and Q are either both true or both false: $Q \Leftrightarrow R \stackrel{\text{def}}{=} (R \wedge Q) \vee (\neg R \wedge \neg Q)$. For instance, $(R \Leftrightarrow Q) \Leftrightarrow ((R \Rightarrow Q) \wedge (Q \Rightarrow R))$.
$\forall x : P(x)$	For all x the predicate $P(x)$ is true.
$\exists\, x : P(x)$	There exists x such that the predicate $P(x)$ is true.
$A = \{a, b, c, d\}$	Set A is composed of elements a, b, c and d
$\{x \mid P(x)\}$	A set of all elements x for which the predicate $P(x)$ is true.
$B = \{x \in A \mid P(x)\}$	Set B is the set of all elements from the set A for which the predicate $P(x)$ is true.
$x \in A$	Element x belongs to set A.
$x \notin A$	Element x does not belong to set A; $x \notin A \stackrel{\text{def}}{=} \neg(x \in A)$.
$\emptyset$	The empty set – the unique set that contains no elements; $\forall x : x \notin \emptyset$.
(a, b)	An ordered pair with elements a and b; $(a, b) \stackrel{\text{def}}{=} \{\{a\}, \{a, b\}\}$ (Kuratowski, 1921). An important property holds: $(a, b) = (c, d) \Leftrightarrow ((a = b) \wedge (b = d))$.
(a, b, c)	An ordered triple; $(a, b, c) \stackrel{\text{def}}{=} (a, (b, c))$.
$A \cup B$	Set union – a set of all elements that belong to either A or B (or both); $A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \vee x \in B\}$.
$A \cap B$	Set intersection – a set of all elements that belong to both A and B; $A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \in B\}$.
$A \setminus B$	Set difference – a set of elements from A that are not in B; $A \setminus B \stackrel{\text{def}}{=} \{x \in A \mid x \notin B\}$.
$A \subseteq B$	A is a subset of B – all elements of A are also elements of B; $A \subseteq B \stackrel{\text{def}}{=} \forall x : (x \in A \Rightarrow x \in B)$.
$A = B$	Sets A and B are equal, *i.e.* they are the same set; $A = B \stackrel{\text{def}}{=} \forall x : (x \in A \Leftrightarrow x \in B)$. For instance, $A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A)$.
$(-x, -y, -z)$	A symmetry operator represented as an ordered triple of general position coordinate expressions.
$\{(x, y, z), (-x, -y, z)\}$	A set of symmetry operators.

that has enjoyed numerous republications (Halmos, 2017). More formal and complete treatment is given by Paul Bernays (1958). Modern notation follows contemporary discrete mathematics textbooks (Rosen, 2012), some openly available online (Doerr & Levasseur, 2021*a*,*b*). The set notation used in the paper is summarized in Table 1.

APPENDIX *E*
Developing a proof in Isabelle

Formal proofs of program correctness, even for moderately sized programs, can become long and involved, full of complex details. This situation naturally calls for some kind of automation. Assistance may be provided by an interactive tool (called a 'theorem prover' or a 'proof assistant') which records and maintains a proof as it is constructed step by step. Such a tool ensures the high accuracy and soundness needed in complex detailed mathematical proofs. Furthermore, mechanized logics (which these tools rely on) cannot allow any 'hand-waving' over matters of syntax or semantics.

The underlying tool logic can be easily extended by the user. These extensions are organized into units called 'theories', which contain a number of definitions, axioms and proven logical statements ('theorems' and 'lemmas'). Proofs of these theorems are facilitated by pre-defined 'proof methods' ('tactics'), encoding sound logical inferences that can be applied to a potential theorem ('goal') to ascertain its validity.

Isabelle is one such interactive proof assistant. It checks the proof correctness, but the proof itself should be provided by the user. The level of detail required in the proof depends on the employed automation. The Isabelle proof assistant has two main sources of automation: the available proof methods and the proof search engine, called Sledgehammer. The former allow us to automate proof construction using given facts (*i.e.* the proof hypotheses and previously proven theorems) and the latter allows us to automate the search for other usable facts and proof methods for a given goal. In the following example we show the basic workflow used while developing the proof. The workflow is typical and is not tied to the space-group-builder algorithm.

As an example we take a fragment of the `builderRec_produces_subgroup` theorem which corresponds to the cases (i)–(iv) in Theorem 2. At this point in the proof, the goal is to show that $R \leq G$ where $R = \mathit{builderRec}\ H\ \{g\}$ is the result of the space-group-builder algorithm. This goal in Isabelle is formulated as

```
1: have "subgroup R G"
2:     sorry
```

The keyword **have** asserts a new fact. We can attempt to prove this fact automatically by launching the Sledgehammer tool after pointing a cursor right after the assertion. This statement is too complex for Isabelle or Sledgehammer to prove automatically, and thus we can mark it temporarily as unproven using the keyword **sorry**. This allows us to work on other parts of the proof before proving this assertion.

To proceed with this proof, we have to provide more details on how to decompose it. In this case we have to consider the *subgroup* definition and split the proof into four sub-goals corresponding to the properties composing it:

```
1: have "subgroup R G"
2: proof -
3:     have R_subset: "R ⊆ carrier G" sorry
4:     moreover have R_m_closed: "⋀x y. ⟦x ∈ R; y ∈ R⟧ ⟹ x ⊗ y ∈ R" sorry
5:     moreover have R_one_closed: "𝟭 ∈ R" sorry
6:     moreover have R_m_inv_closed: "⋀x. x ∈ R ⟹ inv x ∈ R" sorry
7:     ultimately show "subgroup R G" by (simp add: subgroup_def)
8: qed
```

Here we have **sorry** replaced by **proof -** ... **qed**. The dash here instructs Isabelle to avoid using the default proof method and use the one provided in the proof directly. The body of this block should contain a series of intermediate assertions followed by the **show** command, proving the original goal. To check whether the introduced intermediate assertions are enough to prove the goal, we temporarily omit their proof with the keyword **sorry** and concentrate on the **show** part. The final goal is proven by the simplification method `simp` augmented with four properties of a group and the definition of the *subgroup*. With the above-mentioned structure in place, the proof (after the **by** keyword) for the original goal is proposed by Sledgehammer automatically.

Having the main goal proven, we should revisit all the assertions we postponed and prove them one by one. For example, to prove closure under the group operation (marked by the name `R_m_closed`), we remove the keyword **sorry** and provide a suitable set of facts and a proof method. In this case, having the lemma `builderRec_m_closed` already formulated and proven, the proof is automatically found by Sledgehammer:

```
moreover have R_m_closed: "⋀x y. ⟦x ∈ R; y ∈ R⟧ ⟹ x ⊗ y ∈ R"
    using builderRec_m_closed g pre R by metis
```

A similar approach is used to construct the remaining proof in a hierarchical way.

References

Armstrong, J. (2013). *Programming Erlang: Software for a Concurrent World*. Dallas, Raleigh: Pragmatic Bookshelf.

Barnett, M., DeLine, R., Fändrich, M., Jacobs, B., Leino, R., Schulte, W. & Venter, H. (2005). *VSTTE 2005: Verified Software: Theories, Tools, Experiments*, pp. 144–152. Berlin, Heidelberg: Springer-Verlag.

Chang, G. (2003). *J. Mol. Biol.* **330**, 419–430.

Chapin, P. C. & McCormick, J. W. (2015). *Building High Integrity Applications with SPARK*. Cambridge University Press.

Doerr, A. & Levasseur, K. (2021*a*). *Applied Discrete Structures*, https://faculty.uml.edu/klevasseur/ADS2.

Doerr, A. & Levasseur, K. (2021*b*). *Applied Discrete Structures*, https://discretemath.org/ads-latex/ads.pdf.

Downward, M. (2015). *Found. Chem.* **17**, 275–287.

Fedorov, E. S. (1891). *Zap. Min. Obshch. (Trans. Miner. Soc.)*, **28**, 1–146.

Fischer, A. & Koch, E. (2005). *International Tables for Crystallography*, Vol. A, *Space-Group Symmetry*, edited by Th. Hahn, Section 11.1.1, p. 810. Heidelberg: Springer.

Gražulis, S., Merkys, A., Vaitkus, A. & Okulič-Kazarinas, M. (2015). *J. Appl. Cryst.* **48**, 85–91.

Greengard, S. (2021). *Commun. ACM*, **64**, 13–15.

Grosse-Kunstleve, R. W. (1999). *Acta Cryst.* A**55**, 383–395.

Grosse-Kunstleve, R. W. & Adams, P. D. (2002). *Acta Cryst.* A**58**, 60–65.

Hahn, Th. (2005). Editor. *International Tables for Crystallography*, Vol. A, *Space-Group Symmetry*. Heidelberg: Springer.

Hall, S. R. (1981). *Acta Cryst.* A**37**, 517–525.

Halmos, P. R. (1960). *Naive Set Theory*. New York: Van Nostrand Reinhold Company.

Halmos, P. R. (2017). *Naive Set Theory*. Mineola: Dover.

Kuratowski, C. (1921). *Fund. Math.* **2**, 161–171.

Merkys, A., Vaitkus, A., Butkus, J., Okulič-Kazarinas, M., Kairys, V. & Gražulis, S. (2016). *J. Appl. Cryst.* **49**, 292–301.

Merkys, A., Vaitkus, A. & Gražulis, S. (2021). *cod-tools*, https://www.crystallography.net/archives/2021/software/cod-tools/.

Nipkow, T., Wenzel, M. & Paulson, L. C. (2002). *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag.

Paul Bernays, A. A. F. (1958). *Axiomatic Set Theory*, Studies in Logic and the Foundations of Mathematics, Vol. 34. Amsterdam: North-Holland Publishing Company.

Paulson, L. C. (1986). *J. Log. Program.* **3**, 237–258.

Quirós, M., Gražulis, S., Girdzijauskaitė, S., Merkys, A. & Vaitkus, A. (2018). *J. Cheminform*, **10**, 23.

Rosen, K. H. (2012). *Discrete Mathematics and Its Applications*. McGraw-Hill.

Schoenflies, A. M. (1892). *Krystallsysteme und Krystallstruktur*. Leipzig: Teubner. https://ia902706.us.archive.org/13/items/krystallsysteme00schogoog/krystallsysteme00schogoog.pdf.

Shaffer, P. A., Schomaker, V. & Pauling, L. (1946). *J. Chem. Phys.* **14**, 648–658.

Vaitkus, A., Merkys, A. & Gražulis, S. (2021). *J. Appl. Cryst.* **54**, 661–672.

Wall, L., Christiansen, T. & Orwant, J. (2000). *Programming Perl*. Sebastopol: O'Reilly.