Geometric rule check with Answer Set Programming for barrier-free navigation

Marian Kurzawa
Construction Informatics, University of Kassel, Mönchebergstraße 7, 34125 Kassel, Germany
E-mail(s): m.kurzawa@uni-kassel.de

Abstract: The topic of rule-based navigation within buildings is an area of current research. The objective to make a public building accessible for all user groups requires the implementation of complex geometric rule checks. This paper defines exemplary rules based on the German building code DIN 18040-1 and implements rule checks using a declarative programming approach (Answer Set Programming). The results indicate that the declarative implementation has some significant limitations in terms of complexity (linear solver). However, it also offers a promising approach to constraint-based geometric rule checking, which could be a valuable contribution to future research.
Keywords: Rule check, Declarative Programming, Algorithms

Erschienen in Tagungsband 35. Forum Bauinformatik 2024, Hamburg, Deutschland, DOI: 10.15480/882.13504

1 Introduction

In the light of recent developments concerning indoor navigation, the topic of accessibility has assumed greater significance. A number of countries have expressed a desire to ensure, that public buildings are accessible to all user groups. The requirements for such accessibility can be derived from the relevant building codes (in Germany DIN 18040-1 [1]). The range of complexity for such requirements varies considerably, from relatively simple to highly sophisticated and detailed. For instance, the required minimum width of a door is a relatively straightforward confirmation. While the verification of unobstructed maneuvering spaces in front of doors or other points of egress may be straightforward in practice, the formulation of an algorithmic representation of such requirements for a general case is more complex. Consequently, this paper explores the current state of conceptualization for geometric rule checks. In this context, a „rule“ refers to a set of requirements prescribed by the building codes. Rule checks for different complexity classes are developed with a declarative programming approach (Answer Set Programming). Limitations of the used approach are identified, and possibilities for future research in the domain of declarative programming and geometric rule checking are proposed.

1.1 Related work

A number of authors have addressed the definition of rules. Métral, Daponte, Caselli, *et al.* [2] specify rule expressions with mathematical operators and limiting values. The rules can be evaluated for models expressed in RDF, using SHACL or SPARQL. Another approach is presented by Pauwels,

Deursen, Verstraeten, *et al.* [3]. The authors utilize a rule ontology to verify semantic properties, such as the room height, and more complex calculations to assess the acoustic performance of a building. The aforementioned concepts are primarily concerned with semantics. In their work, Eastman, Lee, Jeong, *et al.* [4] present a process for rule definition and discuss example rules on accessibility rule checking. These include the wheelchair turning space in Solibri Office (formerly Solibri Model Checker). Similar checks are conducted by Andrich, Daniotti, Pavan, *et al.* [5], also with Solibri Office. The concepts already incorporate geometric tests. However, as Solibri Office is a proprietary software, there is no publicly available rule definition for such geometric tests. The traversability of spaces is discussed by Bandi and Thalmann [6]. The passability of a space can be determined by discretizing the indoor space into regular rectangular grids. Literature also contains complex discretizations using Voronoi-Diagrams or hexagonal grids [7]-[8]. A more complex approach is developed by Diakité and Zlatanova [9] with the Flexible Space Subdivision (FSS). Spaces are classified into O(bject)-spaces, F(unctional)-spaces and R(emaining free)-spaces, with only the R-spaces beeing considered passable, if there is sufficient space for an A(gent)-space to traverse it. Without respect to navigation Akanmu, Olayiwola, and Olatunji [10] apply a game engine to render geometric tests for maintenance workers. They verify the correct positioning of a ladder at a specific location within the building, taking into account the height of the floor and the ground of the ladder (with a Bounding Box). The majority of the introduced methods can be assigned to imperative object-oriented programming (focus on control flow). Literature lacks declarative implementations (focus on output) for complex geometric rule checking. One of the few declarative approaches to execute rule checks on a building models is presented by Arias, TÖRMÄ, Carro, *et al.* [11]. The authors employ Answer Set Programming (ASP) to filter a building model, using logic expressions. In this approach, the „answers“, which fulfill all given constraints, are collected in answer sets. In addition to verifying a sufficient width of windows within a particular room type, the authors also employ Boolean operations on shapes. Their concept is founded on logical principles, but operates on geometric models. Consequently, it can be extended to check complex geometric rules.

1.2 Methodology

As the known concepts for geometric rule checking focus on imperative programming, the objective of this paper is to test the usability of complex geometric rule checks, using a declarative programming approach. Therefore, the implementation of Arias, TÖRMÄ, Carro, *et al.* [11] is extended to check geometric rules. Exemplary, checkable requirements with different complexity are collected from the German building code DIN 18040-1 [1]. Based on the resulting codes, benefits and limits to the used approach are stated, followed by potentials for future development in this area.

2 Checkable Rules

The following sections present the implementation of inclusive navigation requirements using Answer Set Programming. The checkable rules are derived from building codes. For the tests, different rules with varying degree of complexity are employed. As a definition for the complexity of those rules, this paper refers to Solihin and Eastman [12]. The authors define four complexity classes: rules that require a single or small number of explicit data (class 1), rules that require simple derived attribute

values (class 2), rules that require an extended data structure (class 3), and rules that require a „proof of solution“ (class 4). Based on these classes, different rules are to be tested, which correspond to requirements demanded by the building codes. As discussed by Kurzawa and Kirchner [13], the building codes of several countries exhibit certain similarities. One of those is the clear space in front of a door. The following rules are abstracted from common requirements of the building codes.

2.1 Minimum door width (class 2)

The first checkable rule assumes, that a door must be at least 1.0 m wide, in order to be traversable by a wheelchair user (based on DIN 18040-1). This rule can be classified as class 1, if the width of the door is stored in an object attribute. In this example, the width is derived from the geometry. Consequently, the rule is considered to be class 2.

2.2 Clear door spaces (class 3)

A more complex issue is to ascertain, whether there is sufficient turning space for a wheelchair in front of a door. In this test, a free space of 1.5 x 1.5 m in front of both sides of the door is required (based on DIN 18040-1). In order to maintain the simplicity of the checks, it is assumed that the space is placed in the center in front of the door. In order to ascertain whether the space in front of the door is unobstructed, it is necessary to utilize an auxiliary object for the door space. For a compliant door, the corresponding auxiliary door space should not clash with other objects (in this case walls). The test is classified class 3.

2.3 Accessibility of doors in room (class 4)

The accessibility check of doors from a specific room is considered to be class 4. A rule check entails determining, whether there are any obstructions in the path to the door. For the sake of simplicity, only the straight path from the center of the door space to the center of a room with a convex footprint is considered. In accordance with the provisions of DIN 18040-1, the minimum width of the passageway is 1.2 m. However, the complexity of this rule significantly increases, if concave rooms are taken into account.

3 Answer Set Programming

In this section, the aforementioned rules (see Chapter 2) are implemented, using Answer Set Programming. The spatial reasoner proposed by Arias, TÖRMÄ, Carro, *et al.* [11] serves as the foundation for the tests. The fundamental objective is to identify all volumetric bodies, that comply with predefined constraints without contradictions. The concept employs a linear constraint solver. In the case of orthogonally aligned examples, mathematical operations such as „intersection(...)“ can be evaluated by formulating constraints like $X < X_a$ and $X > X_b$ as intervals and then shifting the borders of the interval of X. However, volumetric bodies are only approximated as the union of convex shapes that are parallel to axes of coordinates. This approach does not consider non-orthogonally aligned examples. The used datasets store the shapes of physical objects by three points. The initial two points delineate intervals along the coordinate axes (see https://gitlab.software.imdea.org/joaquin.arias/spatial/-/tree/master/examples). Similarly, doors and windows are defined by the opening space within a wall. In the orthogonally aligned model, the

third point of the dataset is not used and it is unclear, how this point is defined. Typically, the third point defines the orientation of the volumetric body. In the given examples the third point is situated in proximity to the centroid of the shape. In conclusion non-orthogonally aligned examples result in defective geometries, as illustrated in Figure 1. In the example, an input file containing rotated points is utilized. Although the solver is unable to process non-orthogonally aligned examples, the following rule checks should be applicable to a general non-orthogonally aligned example as well. For these tests, the third point is corrected either as (1,0,0) or (0,1,0). In the subsequent sections, only code snippets are shown and explained. The revised input files, along with the comprehensive developed code, are accessible on https://gitlab.com/designforall/geometric-rule-checking.

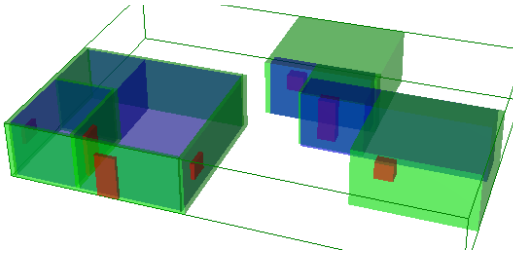

Figure 1: Left: original; right: 45° rotated

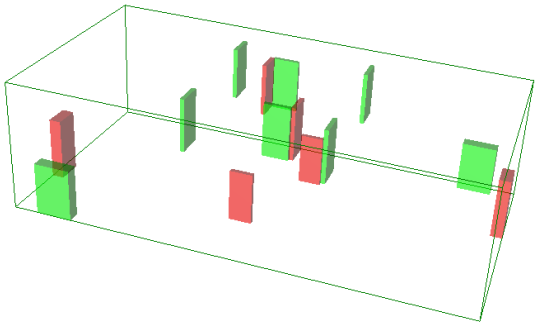

Figure 2: Evaluation of the existing door openings

3.1 Minimum door width (class 2)

In order to ascertain the width of the doors, in a first step the doors have to be filtered (compare Algorithm 1). The width of the door can then be derived from the geometry of the door. As the third point defines the direction of the door, the depth of the door can be determined using the Hesse normal form and coordinate form of a plane. The plane is defined by the first given point p_1 and the directional vector n (here p_3). The width of the door opening is obtained by rotating the directional vector n by 90 degrees. With the rotated direction vector the distance between p_2 and the plane with p_1 the outcome of the equation is the door width between the two points normalized with L (compare Algorithm 1).

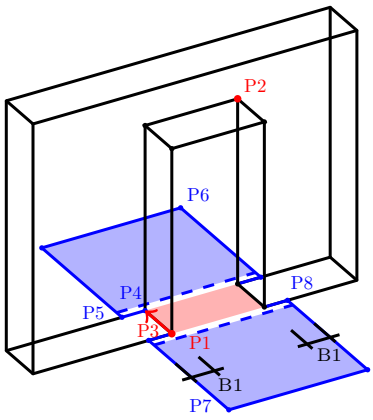

Figure 3: Door space definition via orientation vectors

Algorithm 1: Check of the width of the door openings

```
check_doorwidth :-
    findall(Sh,(
        object('IfcDoor', _, (X1, Y1, Z1), (X2,
            Y2, Z2), (X3, Y3, Z3)),
        L is sqrt(X3 * X3 + Y3 * Y3),
        Width is abs(-Y3*(X2-X1)+X3*(Y2-Y1))/L,
        Width >= 1.0,
        box_shape(point(X1, Y1, Z1), point(X2, Y2
            , Z2), Sh)),
    GoodDoors),
    output([par(GoodDoors,green)]).
```

These mathematical calculations must be evaluated for the geometry of each filtered object. All answers Sh that satisfy the condition ($Width \geq 1.0$) are included in the answer set *GoodDoors*, answers that violate the rule can be gathered analogously in an answer set *BadDoors*. The results are visualized in a X3D file. The objects are colored according to the result of the rule (compare Figure 2).

3.2 Clear door spaces (class 3)

To check door spaces, auxiliary objects must be defined. These are computed based on the direction of the door p_3 (X_3, Y_3). The depth D and the width W of the door opening are to be charged similarly to Chapter 3.1. Subsequently, the points P_4 to P_8 are estimated by translating the given points according to the orientation of the door (see Figure 3 and Algorithm 2). The door spaces from P_5 to P_6 and P_7 to P_8 are returned (see Figure 4). The points P_5 and P_8 are additionally translated by 2 cm away from the door in order to avoid clashes between the door spaces and the door hosting wall.

Algorithm 2: Create door spaces based on direction vectors

```
door_space(X1, Y1, Z1, X2, Y2, Z2, X3, Y3, Sh) :-
    L is sqrt(X3 * X3 + Y3 * Y3),
    D is (X3*(X2-X1)+Y3*(Y2-Y1))/L,      W is abs(-Y3*(X2-X1)+X3*(Y2-Y1))/L,
    B1 is (1.5 - W) / 2,
    Px4 is X1 + D*X3,
    Py4 is Y1 + D*Y3,
    L2 is sqrt((X2-Px4) * (X2-Px4) + (Y2-Py4) * (Y2-Py4)),
    Px5 is Px4 - B1*(X2-Px4)/L2 + 0.02*X3/L,
    Py5 is Py4 - B1*(Y2-Py4)/L2 + 0.02*Y3/L,
    Px6 is X2 + 1.5*X3/L + B1*(X2-Px4)/L2,
    Py6 is Y2 + 1.5*Y3/L + B1*(Y2-Py4)/L2,
    Px7 is Px5 - (1.5+D)*X3/L,
    Py7 is Py5 - (1.5+D)*Y3/L,
    Px8 is X2 - (D+0.02)*X3/L + B1*(X2-Px4)/L2,
    Py8 is Y2 - (D+0.02)*Y3/L + B1*(Y2-Py4)/L2,
    (box_shape(point(Px5, Py5, Z1), point(Px6, Py6, Z2), Sh);
    box_shape(point(Px7, Py7, Z1), point(Px8, Py8, Z2), Sh)).
```

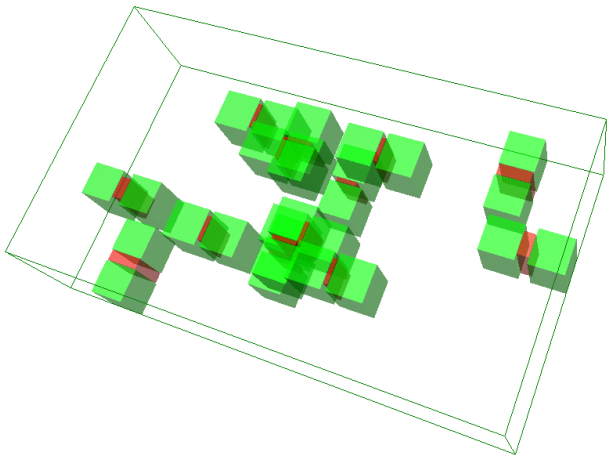

Figure 4: Red: doors, green: door spaces

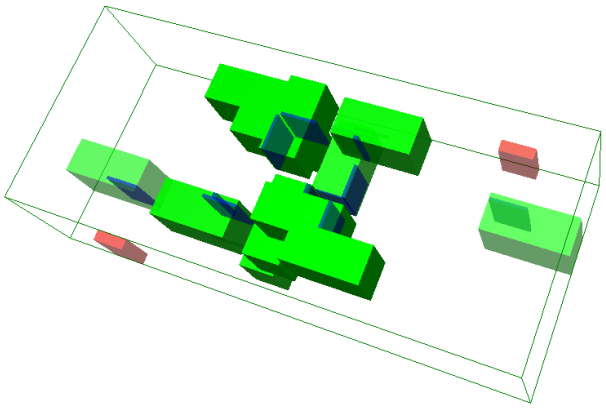

Figure 5: Red: doors with sufficient door spaces, green: simplified door spaces with intersections, blue: intersections with walls

In another query all doors with clashes between door spaces and existing walls are computed. The method *shape_intersect(...)* is described by Arias, TÖRMÄ, Carro, *et al.* [11]. In this example, the evaluation can be executed in a single query, rendering this step unnecessary. The doors that contravene the rule, can be identified, by computing the intersection of the door spaces with walls. If the intersecting area is empty, the door is returned (compare Algorithm 3).
If the identification of the violating objects requires multiple queries, the valid doors can also be identified through the application of mathematical calculations. After computing the intersections of the door spaces with walls (see Figure 5 in blue), corresponding simplified door spaces (one box using solely the points P_5 and P_8, shown in Figure 5 in green) can be determined. The intersection of the simplified door spaces with the doors yields the rule-violating doors. If no intersection between one door and any of the computed simplified door spaces exists, the door passes the test (shown in Figure 5 in red). As illustrated by this example, the approach proposed by Arias, TÖRMÄ, Carro, *et al.* [11] does not yield the bounding values themselves within a box shape. Instead, it provides constraints that define the boundaries of the shape (box shape: [X,Y,Z]; X>X_a, X<X_b, etc.). Consequently, it is not straightforward to access the door to the intersecting door space. In the case of multiple queries it is advisable to include the boundaries of the volumetric body within the return of the method *box_shape(...)*. This facilitates their use in subsequent computations.

Algorithm 3: Detection of doors without sufficient maneuvering space

```
findall(Door,(
    object('IfcWall', _, (Xa, Ya, Za), (Xb, Yb, Zb), _),
    box_shape(point(Xa, Ya, Za), point(Xb, Yb, Zb), Walls),
    object('IfcDoor', _, (X1, Y1, Z1), (X2, Y2, Z2), (X3, Y3, Z3)),
    door_space(X1,Y1,Z1, X2,Y2,Z2, X3,Y3, Doorspace),
    shape_intersect([Walls], [Doorspace], Inter), Inter \= [],
    box_shape(point(X1, Y1, Z1), point(X2, Y2, Z2), Door)),
ProblematicDoors),
```

3.3 Accessibility of doors in room (class 4)

To ascertain whether a door is directly accessible from the center of a space, even orthogonally aligned building objects lead to non-orthogonally aligned paths between the two considered objects. When considering only convex spaces, the path can be computed with two orthogonally aligned cuboids (compare Figure 6). The query is defined for one space. First, walls, doors and spaces are filtered similarly to the implementation in Chapter 3.1. For the selected space and for the door spaces of each door, the two-dimensional centroids are computed. In this orthogonally aligned example with convex spaces, the path from the center of the space to the door space can be computed using two cuboids (see Figure 6). Algorithm 4 illustrates the code to implement a path (first in the x-, then in the y-direction). A second path (first in the y-, then in the x-direction) can be defined analogously. For a door to be accessible from the center of the room, it is necessary that at least one of the two paths (*PathPartA*, *PathPartB*) does not intersect with any walls (compare Algorithm 5).

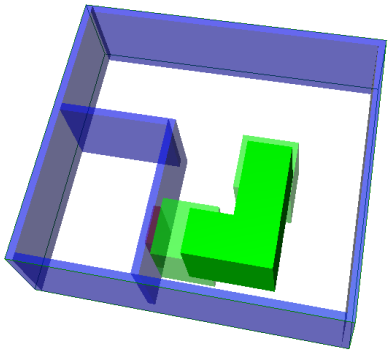

Figure 6: One possible path (*Path1(...)*, green) from the center of the room to the door (red)

Algorithm 4: Compute orthogonally aligned path

```
path1(X1, Y1, Z1, X2, Y2, Sh) :-
    Z2 is Z1+2.1,          B is 1.2,
    Y1l is Y1-B/2,         Y1r is Y1+B/2,
    X2l is X2-B/2,         X2r is X2+B/2,
    Y2r is Y2+B/2,
    (box_shape(point(X1, Y1l, Z1),
        point(X2r, Y1r, Z2), Sh);
    box_shape(point(X2l, Y1l, Z1),
        point(X2r, Y2r, Z2), Sh)).
```

Algorithm 5: Compute, whether an orthogonally aligned path intersects any wall

```
path1(Xm2,Ym2,Z1, Xm,Ym, PathPartA), path2(Xm3,Ym3,Z1, Xm,Ym, PathPartB),
((\+ ( member(Wall, Walls),
    shape_intersect([PathPartA], [Wall], Intersection), Intersection\=[]));
(\+ ( member(Wall, Walls),
    shape_intersect([PathPartB], [Wall], Intersection), Intersection\=[])))
```

4 Discussion

This paper provides a concise overview of current developments in rule checking. The literature offers several object-oriented approaches that utilize semantics. In addition to space discretization geometric tests are less frequently recorded. This paper presents a declarative implementation based on Answer Set Programming, which is derived from the work of Arias, TÖRMÄ, Carro, *et al.* [11]. The constraint-based concept facilitates mathematical operations similar to Constructive Solid Geometry for cuboids. Based on the German building code, three rules for accessibility of paths with different complexity are introduced and implemented. The examples demonstrate that complex geometric rules can be checked using declarative programming. Yet the used implementations reveal significant shortcomings. For instance, the linear solver lacks definitions for non-orthogonally aligned shapes. The assumption of orthogonally aligned volumetric bodies significantly simplifies the complexity of the solver, as there are fewer special cases to be considered and mathematical operations are easier to evaluate. However, as the third example (Chapter 3.3) indicates, navigation problems often require non-orthogonally aligned paths. Therefore, the solver implementation is considered too specific for complex geometric navigation testing.

The definition of a rotation angle of the object would be more appropriate. In general, the declarative approach presents certain challenges. For instance, after filtering all door spaces, the access of the underlying door object within a subsequent query (without geometric calculations) is not supported (see example 2 in Chapter 3.2). Nevertheless, the constraint-based concept of describing volumetric bodies and querying computed intersections is promising for future research. A solver with constraint logic for non-orthogonally aligned geometries (comparable to CSG-transformations) would be advantageous to define more complex shapes.

References

[1] DIN Deutsches Institut für Normung e.V., *DIN 18040-1: Barrierefreies Bauen - Planungsgrundlagen - Teil 1: Öffentlich zugängliche Gebäude*. Berlin: Beuth Verlag, 2010.

[2] C. Métral, V. Daponte, A. Caselli, G. Di Marzo Serugendo, and G. Falquet, "ONTOLOGY-BASED RULE COMPLIANCE CHECKING FOR SUBSURFACE OBJECTS", *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLIV-4/W1-2020, pp. 91–94, 2020. DOI: 10.5194/isprs-archives-XLIV-4-W1-2020-91-2020.

[3] P. Pauwels, D. V. Deursen, R. Verstraeten, *et al.*, "A semantic rule checking environment for building performance checking", *Automation in Construction*, vol. 20, no. 5, pp. 506–518, 2011. DOI: https://doi.org/10.1016/j.autcon.2010.11.017.

[4] C. Eastman, J.-m. Lee, Y.-s. Jeong, and J.-k. Lee, "Automatic rule-based checking of building designs", *Automation in Construction*, vol. 18, no. 8, pp. 1011–1033, 2009. DOI: https://doi.org/10.1016/j.autcon.2009.07.002.

[5] W. Andrich, B. Daniotti, A. Pavan, and C. Mirarchi, "Check and Validation of Building Information Models in Detailed Design Phase: A Check Flow to Pave the Way for BIM Based Renovation and Construction Processes", *Buildings*, vol. 12, no. 2, 2022. DOI: 10.3390/buildings12020154.

[6] S. Bandi and D. Thalmann, "Space Discretization for Efficient Human Navigation", *Computer Graphics Forum*, vol. 17, no. 3, pp. 195–206, 1998. DOI: 10.1111/1467-8659.00267.

[7] I. Afyouni, C. Ray, and C. Claramunt, "Spatial models for context-aware indoor navigation systems: A survey", *Journal of Spatial Information Science*, vol. 4, 2012. DOI: 10.5311/JOSIS.2012.4.73.

[8] J. O. Wallgrün, "Autonomous Construction of Hierarchical Voronoi-Based Route Graph Representations", in *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, vol. 3343, 2004, pp. 413–433. DOI: 10.1007/978-3-540-32255-9_23.

[9] A. A. Diakité and S. Zlatanova, "Spatial subdivision of complex indoor environments for 3D indoor navigation", *International Journal of Geographical Information Science*, vol. 32, no. 2, pp. 213–235, 2018. DOI: 10.1080/13658816.2017.1376066.

[10] A. A. Akanmu, J. Olayiwola, and O. A. Olatunji, "Automated checking of building component accessibility for maintenance", *Automation in Construction*, vol. 114, p. 103 196, 2020. DOI: https://doi.org/10.1016/j.autcon.2020.103196.

[11] J. Arias, S. TÖRMÄ, M. Carro, and G. Gupta, "Building Information Modeling Using Constraint Logic Programming", *Theory and Practice of Logic Programming*, vol. 22, pp. 1–16, 2022. DOI: 10.1017/S1471068422000138.

[12] W. Solihin and C. Eastman, "Classification of rules for automated BIM rule checking development", *Automation in Construction*, vol. 53, pp. 69–82, 2015. DOI: 10.1016/j.autcon.2015.03.003.

[13] M. Kurzawa and J. Kirchner, "Extension of accessibility ontologies by requirements for inclusive indoor navigation", *EG-ICE 2024 conference held on July 3-5, at Universidade de Vigo*, 2024.