

5 Implementation

The purpose of this chapter is to provide more detailed information about implemented approaches, to document developed software artefacts and to explain how encountered problems influenced certain design decisions. Section 5.1 discusses the implementation of CAD interface which allows assembly sequence planning (ASP) using STEP assembly models. The module's functionality is demonstrated on an industrial example. Section 5.2 describes how exact solver methods are applied to solve scheduling problems resulting from geometrical constraints of the assembly and resource constraints of the involved assembly system.

5.1 CAD interface

The implementation of CAD interface is based on the method described in [Pin16b] and also makes use of CATIA software suite developed by Dassault Systèmes. The programs described in this section are developed in Visual Basic .NET (VB.NET) to access CATIA scripting API, which provides functionalities to export product-related data, manipulate geometric bodies and execute clash analysis. AND/OR graph related procedures in Sections 5.1.4 and 5.1.5 are implemented in Python.

5.1.1 Demonstrative assembly

A demonstrative product is presented for better understanding of implemented programs. It is a free-to-share model of a single-stage centrifugal pump taken from GrabCAD website. Analysed centrifugal pump contains 23 elementary components including 11 fasteners such as nuts and bolts. The specification tree as displayed in CATIA can be seen on the left side of Fig. 5.1.

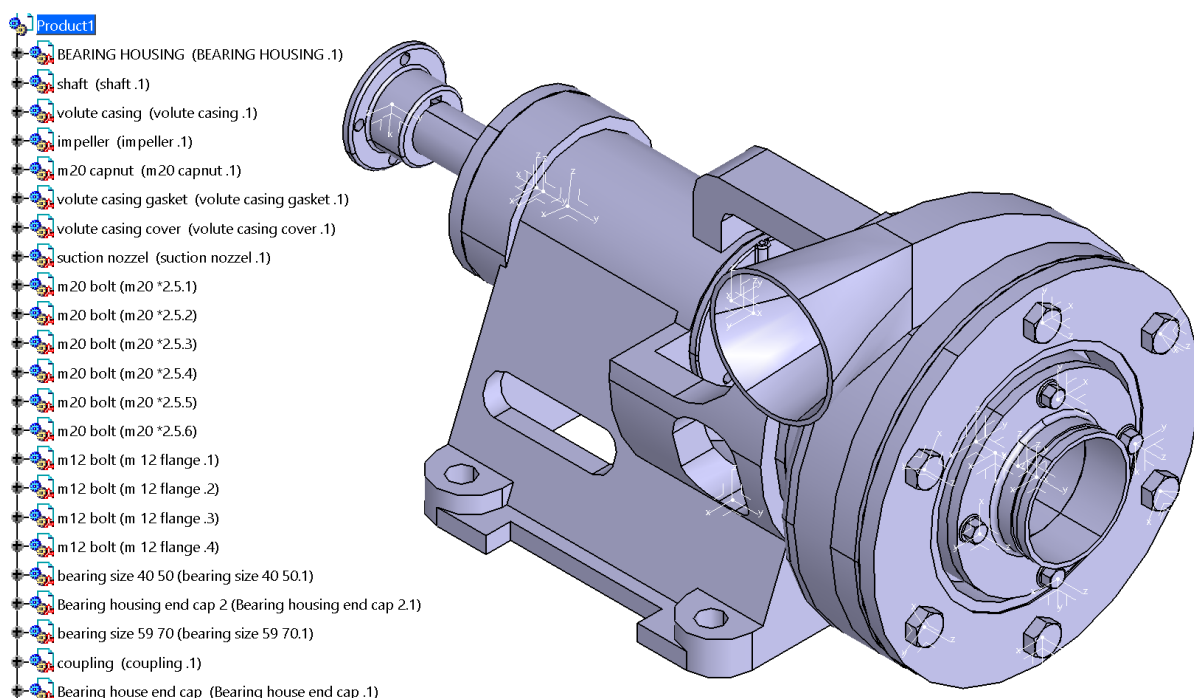


Figure 5.1: Single-stage centrifugal pump

5.1.2 Assembly tiers method

CAD model preparation

A correct model of the analysed product stored in a STEP file is a prerequisite for assembly tiers determination. Such a STEP file has to be opened by the user in CATIA first. After successful import of CAD file into the program, its bill of materials or specification tree is shown next to the 3D rendering. It is important to consider that complex assemblies may have multiple hierarchy levels in the specification tree. Methodology proposed by Pintzos et al. only considers subassemblies on the first level of the tree as components relevant to assembly tiers determination [Pin16b, p. 1047]. However, this methodology significantly reduces the utility of assembly tiers determination because the operation sequences to build up the top-level assemblies from their components are neglected. To overcome this limitation, a VB macro called PrepareProductStructure was written that dissolves hierarchical product structures and puts all elementary parts on the top level of specification tree. The list shown below is the optional procedure of creating a new CAD file without hierarchical product structure, stored as a CATProduct file.

1. Open STEP assembly file in CATIA
2. Save a copy of assembly file as CATProduct
3. Restart CATIA
4. Open previously saved CATProduct file
5. Run PrepareProductStructure VB macro, resulting in new file 'Assembly.CATProduct'
6. Run AssemblyTiers algorithm from VB.NET application

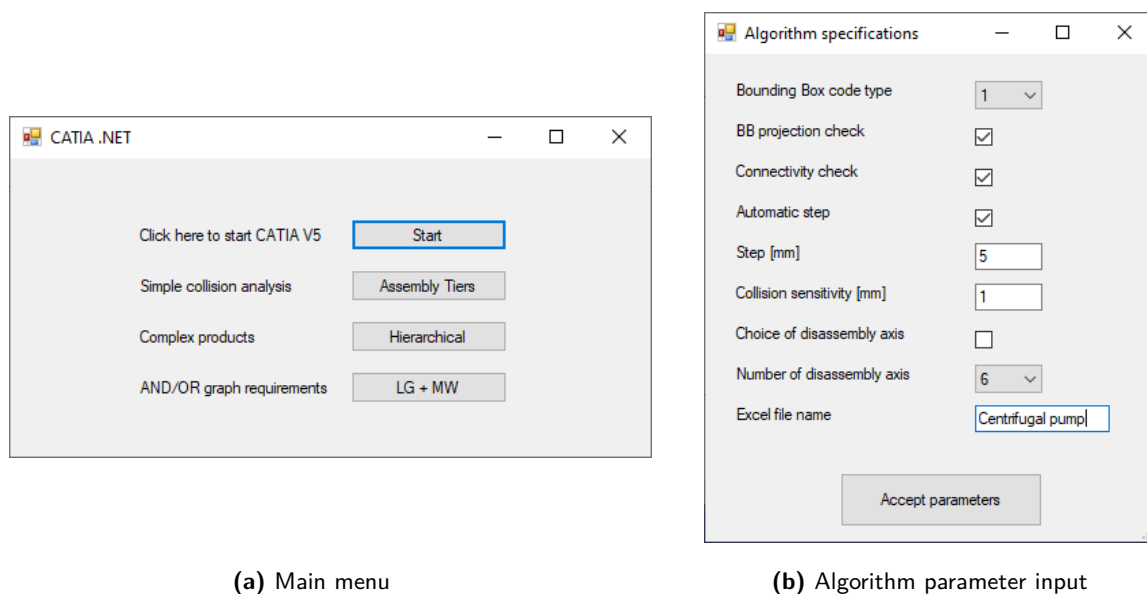


Figure 5.2: CATIA interface GUI

User input and product preprocessing

After the CAD file is prepared, developed VB.NET application can be started from Microsoft Visual Studio© or as an executable binary. In the beginning, a menu with four options appears (s. Fig. 5.2a). If CATIA is not running yet, clicking the first button in the main menu will open the application, where a CAD file has to be opened manually. Assembly tiers program is started by clicking the second button in the main menu. An input form appears, where the user can specify parameters for algorithm execution (s. Fig. 5.2b). Algorithm parameters are explained later together with the affected program steps.

Once the parameters are chosen, a reference to CATIA Component Object Model (COM) must be created. This is achieved by command `CATIA = GetObject(, "CATIA.Application")`. Next, an object of type `ProductDocument` is retrieved, which represents the product structure of opened assembly: `document = CATIA.ActiveDocument`. This object has a property called `Product` in the API, which contains all relevant data about analysed assembly. It is passed to `ExtractProducts` function in order to obtain an `ArrayList` of all components (variable `cAllProducts`). The user gets notified about the total number of parts contained in the assembly, including fasteners.

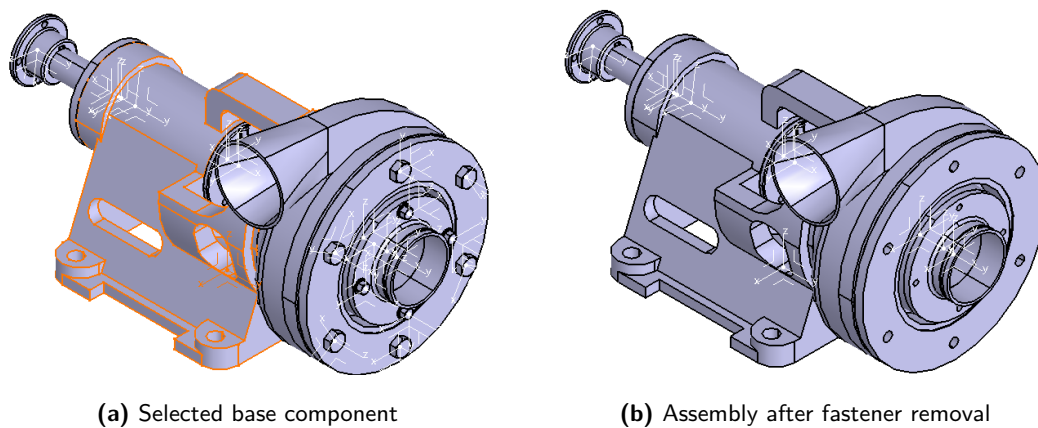


Figure 5.3: Assembly preparation steps

In the next step, the user is requested to select base components which are assumed to be pre-assembled and stationary. This is required to ensure that base components are the ones that can be attached to existing surroundings and foundations which are not present in the STEP model. As per Fig. 5.3a, Bearing housing is the base component of the centrifugal pump. Further model preparation follows the approach proposed in [Pin16b], where fasteners are removed from the assembly based on their names in the specification tree. The result of this step is shown in Fig. 5.3b. Original approach achieves separation of components into fasteners and assembly parts by keyword matching, identifying keywords 'bolt', 'screw', 'clip', 'wedge', 'pin', 'nut' and 'washer' case-independently. This method is expanded in this work to recognise fasteners not only by the mentioned keywords, but also using the names of common industrial norms. For instance, screws can be detected if norms ISO 4762, DIN 912 or DIN 933 appear in the part number. The list of keywords is also extended for nuts (norms ISO 4161, DIN 934, DIN 439) and washers (DIN 9021, DIN 125, DIN 127). Programmatically the removal of fasteners is realised by the function `DeactivateFasteners` which iterates over the list of all identified components and checks parts' attribute `PartNumber` for keywords. If there are no matches, such a part is considered relevant for disassembly simulation and appended to `cRelevantProducts`

list. In other case, an API function is used to deactivate the fastener in CATIA, thus excluding it from collision detections. The user is then informed about the number of parts that will be involved in collision analysis.

Bounding box and removal distances determination

Before disassembly simulation can begin, it is necessary to know how far each part has to be moved from its original position for successful disassembly. A disassembly trial is successful if and when the removed part can continue its movement outside the assembly without ever colliding with other parts. Bounding box (BB) concept is especially relevant in this context. Bounding box of a geometric shape can be defined as a cuboid wherein the shape is inscribed, meaning that no points belonging to the shape lie outside the bounding box and the planes comprising the BB are perpendicular to the axes of a given axis system. A bounding box can be constructed by calculating six extremum points of a geometric shape: minimum and maximum points along each of three principal axes. The distances of these six points from the origin of part axis system fully describe the bounding box.

Fig. 5.4 demonstrates geometric relations between coordinate systems (CS), extrema planes and bounding boxes (BB) in an assembly.

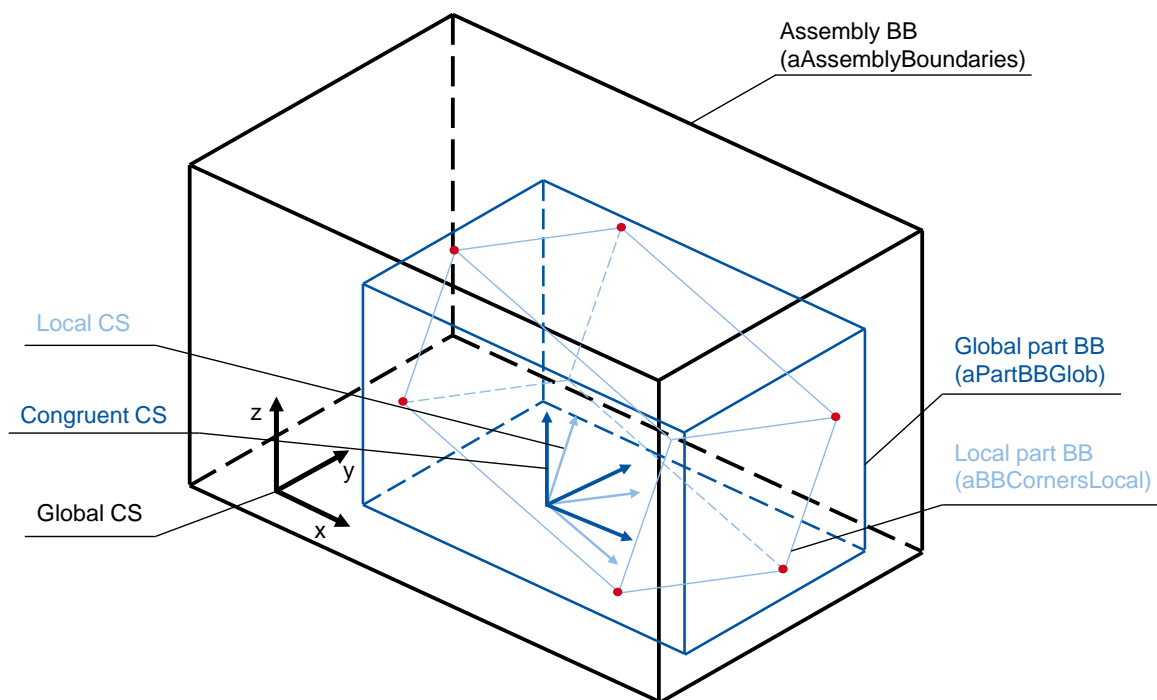


Figure 5.4: Assembly coordinate systems and bounding boxes

Six extremum points in the local axis system span the local part BB by defining extremal planes (light blue cuboid in Fig. 5.4). To ensure correct detection of a successful disassembly attempt along both global and local axes and to avoid complex geometrical calculations after each movement step, a so called global part BB has to be created. If the directions of local and global coordinate systems coincide, then local and global bounding boxes will also be the same. However, if the local axis system is rotated with respect to global CS as in Fig. 5.4, the local part BB will be inscribed in the global part BB. Six of eight corner points of local part

BB (marked red) determine the locations of extremal planes in global part BB. Notice that the coordinate systems of local and global BBs are called local and congruent coordinate systems, respectively. The congruent CS has the same origin as local CS, but its axes are directed parallel to the axes of global CS.

A separate function `GenerateBoundingBox` is called for each relevant (not fastener) part to get the information about its BB and to update the whole assembly's BB. This function receives three arguments: `partDocument1 As PartDocument`, `objProduct As Product`, `i As Integer`. First, the name of `PartDocument` is checked for extension `.CATPart`, which is required for correct execution of geometric operations. If the object type is valid, its attribute `Part` is assigned a variable `part1`. In order to create extremum points, an instance of `HybridShapeFactory` is saved in `hybridShapeFactory1` variable. In the implemented algorithm, API function `AddNewExtremum` of class `HybridShapeFactory` is applied, which can find the extremum points of each part in its local coordinate system, i.e. such points that have the biggest or the smallest coordinates along local axis. For further analysis, extremum points in local CS are transformed into global, or absolute, coordinates `absCoord`.

An important method for coordinate transformations is called `Coord_Transform` in the code. It receives a three-component vector relative to some CS and transforms it into CS of the parent `Product`. This operation can be done recursively until the original vector is transformed into global axis system. Let $\mathbf{x} \in \mathbb{R}^3$ be the point coordinates with respect to global axes, $\mathbf{x}' \in \mathbb{R}^3$ coordinates of the same point in the local axis system and $\mathbf{x}_0 \in \mathbb{R}^3$ translation vector between global and local CS. With rotation matrix $\mathbf{R} \in \mathbb{R}^{3 \times 3}$ representing the orientation of local axes, the transformation realised by `Coord_Transform` is given by Eq. 5.1.

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{R}^{-1} \mathbf{x}' \quad (5.1)$$

Once all six extremum points with respect to local CS are determined, eight corner points of local part BB can be constructed by recombining values. Let x'_{max} , y'_{max} and z'_{max} be the distances of local part BB extremal planes from the local CS origin along positive directions of axes. Similarly, let x'_{min} , y'_{min} and z'_{min} be the extrema in negative directions. In local coordinates, the eight corners populating `aBBCornersLocal` 8×3 array are the following:

$$\begin{aligned} &(x'_{max}, y'_{max}, z'_{max}) \\ &(x'_{max}, y'_{max}, z'_{min}) \\ &(x'_{max}, y'_{min}, z'_{max}) \\ &(x'_{max}, y'_{min}, z'_{min}) \\ &(x'_{min}, y'_{max}, z'_{max}) \\ &(x'_{min}, y'_{max}, z'_{min}) \\ &(x'_{min}, y'_{min}, z'_{max}) \\ &(x'_{min}, y'_{min}, z'_{min}) \end{aligned}$$

In the next logical step, six planes comprising global part BB have to be determined relative to congruent CS. To do so, each corner point from `aBBCornersLocal` is transformed into global coordinates and the position

vector of part's own axis system origin is subtracted. In order to determine the six outermost corners of local part BB (marked red in Fig. 5.4), the points with maximum and minimum coordinate components have to be found. The whole implemented procedure is shown in Alg. 1:

Algorithm 1: GenerateBoundingBox

Result: aPartBBGlob of current part, updated aAssemblyBoundaries

```

1 Initialise aPartBBGlob of current part with  $-\infty$  for maximums and  $+\infty$  for minimums;
2 foreach Local BB corner point do
3   absCoord  $\leftarrow$  transform corner point into global coordinates; CCC  $\leftarrow$  subtract part's origin position
   to get corner coordinates in congruent CS;
4   if  $x, y$  or  $z$  of absCoord  $>$  corresponding maximum in aAssemblyBoundaries then
5     Replace corresponding value in aAssemblyBoundaries with new maximum;
6   end if
7   if  $x, y$  or  $z$  of absCoord  $<$  corresponding minimum in aAssemblyBoundaries then
8     Replace corresponding value in aAssemblyBoundaries with new minimum;
9   end if
10  if  $x, y$  or  $z$  of CCC  $>$  corresponding maximum in aPartBBGlob then
11    Replace corresponding value in aPartBBGlob with new maximum;
12  end if
13  if  $x, y$  or  $z$  of CCC  $<$  corresponding minimum in aPartBBGlob then
14    Replace corresponding value in aPartBBGlob with new minimum;
15  end if
16 end foreach

```

As a result, aPartBBGlob stores the distances from congruent CS origin to global part BB faces and aAssemblyBoundaries is updated by any new global assembly extrema. After processing each relevant part the removal distances can be easily calculated. Removal distance vector of a part is defined as a vector $\Delta \mathbf{x} = (\Delta x_+, \Delta x_-, \Delta y_+, \Delta y_-, \Delta z_+, \Delta z_-)^T$, with

$$\Delta x_+ = X_{max} - \tilde{x}_{min} \quad (5.2)$$

$$\Delta x_- = X_{min} - \tilde{x}_{max} \quad (5.3)$$

$$\Delta y_+ = Y_{max} - \tilde{y}_{min} \quad (5.4)$$

$$\Delta y_- = Y_{min} - \tilde{y}_{max} \quad (5.5)$$

$$\Delta z_+ = Z_{max} - \tilde{z}_{min} \quad (5.6)$$

$$\Delta z_- = Z_{min} - \tilde{z}_{max} \quad (5.7)$$

where

$\Delta x_+, \Delta y_+, \Delta z_+$ are removal distances in positive axis directions

$\Delta x_{-}, \Delta y_{-}, \Delta z_{-}$ are removal distances in negative axis directions

$X_{max/min}, Y_{max/min}, Z_{max/min}$ are the assembly boundaries (aAssemblyBoundaries)

$\tilde{x}_{max/min}, \tilde{y}_{max/min}, \tilde{z}_{max/min}$ are the distances from congruent CS to global part BB faces (aPartBBGlob)

A part is disassembled when any component of part position vector $\mathbf{x}_0 = (x_0, y_0, z_0)^T$ exceeds or becomes less than the corresponding removal distance. A function named `ProductReachedFinalPosition` is implemented to get a CATIA Product's `Position` attribute and call its member function `GetComponents`, which returns a twelve-element array containing nine elements of local CS rotation matrix and three position coordinates. These three elements can then be efficiently checked against previously determined removal distances. In case that a part reaches a position where it is guaranteed to be removed from the assembly the function returns `True`.

The original assembly tiers method by Pintzos et al. requires definition of movement step by which relevant parts are translated every iteration of virtual disassembly algorithm. The authors claim to have applied user-defined steps covering the minimum width of the examined components in their tests on different product designs [Pin16b, pp. 1047, 1055]. The application implemented as a part of this thesis gives user a choice between automatically calculated step and its manual input (see 'Automatic step' checkbox and 'Step [mm]' input field in Fig. 5.2b). For automatic step calculation an empiric formula

$$s = \left\lceil \frac{\left((X_{max} - X_{min}) \cdot (Y_{max} - Y_{min}) \cdot (Z_{max} - Z_{min}) \right)^{1/3}}{50} \right\rceil \quad (5.8)$$

is applied to set step length s by dividing the geometric mean of assembly BB dimensions by a factor of 50 and rounding the result to the closest integer value.

Main algorithm implementation

In the following, comprehensive textual explanations are aided by pseudocode blocks to formulate movement and stationary phases of the assembly tiers algorithm, collision handling procedure and remaining helper algorithms. First, a mathematical notation is required to define occurring variables and sets.

Let $\Pi = \{p_0, \dots, p_{N-1}\}$ be the analysed product consisting of N parts relevant for collision detection. Single parts are denoted by p_i where $i \in \{0, 1, \dots, N-1\}$. $B \subset \Pi$ is the set of preselected base components which are not being moved during virtual disassembly trials but nevertheless can cause clashes. Initial orientations and positions of parts are stored in $\mathbf{X}^* \in \mathbb{R}^{N \times 12}$ matrix so that parts can be returned in their original states whenever a collision prevents further movements. Removal distances vectors mentioned above can get combined into $\Delta \mathbf{X} \in \mathbb{R}^{N \times 6}$ matrix for convenient information extraction:

$$\Delta \mathbf{X} = \begin{bmatrix} \Delta \mathbf{x}_0^T \\ \Delta \mathbf{x}_1^T \\ \vdots \\ \Delta \mathbf{x}_{N-1}^T \end{bmatrix} \quad (5.9)$$

Parameter input dialogue of the program (s. Fig. 5.2b) contains a drop-down element 'Number of disassembly axis' which determines whether a part will be moved along only six global axis directions or additionally along six local axis directions. When the counter of already tested directions $j \in \mathbb{N}$ equals the chosen number of disassembly axes $J \in \{6, 12\}$, it indicates that next part can be processed. Manipulation directions can be mapped to the index j via set $D = \{x_{gl}, y_{gl}, z_{gl}, -x_{gl}, -y_{gl}, -z_{gl}, x_{loc}, y_{loc}, z_{loc}, -x_{loc}, -y_{loc}, -z_{loc}\}$, so that they can be referred to as D_j .

Another two important parameters of the algorithm are step distance s and collision sensitivity ε . A penetration of one geometric body into another is assumed to prevent further disassembly movement whenever the depth of such penetration exceeds ε . As collision detection in CATIA is realised between two Group instances, they are represented by sets $G_1, G_2 \subset \Pi$.

Furthermore, it is necessary to deactivate parts in CATIA as they get successfully disassembled. They comprise a set of parts $\Theta \subset \Pi$, which are not considered in collision detections and are not displayed in CATIA once they are removed from assembly within a tier. In order to identify the parts that need to be assigned a tier and be deactivated, parts are appended to $\Phi \subset \Pi$ – set of parts that could be successfully disassembled along at least one direction. Set of moveable parts $M \subset \Pi$ contains all components that have to be manipulated in the current algorithm iteration. Apart from that, set $L \subseteq \Pi$ is required for connectivity check mode (s. Fig. 5.2b) and defines the subset of parts that is examined for connectivity between its elements.

Three more variables are necessary for complete definition of algorithm's outputs. First, precedence matrix $\mathbf{S} \in \mathbb{R}^{N \times N}$ stores information about allowed assembly operation orders. $\mathbf{S}_{i,k} = 1$ if part i has a direct precedence relation to k and $\mathbf{S}_{i,k} = 0$ otherwise. Second, assembly tiers as defined in [Pin16b] are sets of parts that can be assembled simultaneously, so that each part $p_i \in \Pi$ can be assigned a tier number. Assembly tier numbers begin with 0 for base components and increase as operations can only be executed at later stages. Tiers vector $\mathbf{t} \in \mathbb{R}^N$ stores the assignment of tier numbers to parts, so that \mathbf{t}_i is the tier of part p_i . Last but not least, allowed disassembly directions are to be recorded for each part. This is realised by disassembly directions matrix $\mathbf{Y} \in \mathbb{R}^{N \times 12}$, where $\mathbf{Y}_{i,j} = 1$ if part p_i can be removed from assembly along direction D_j and $\mathbf{Y}_{i,j} = 0$ otherwise.

Variables in code are given names that differ from introduced symbols for better code readability. Table A.1 contains the mapping of mathematical symbols to VB.NET code variables. To avoid linear searches of current part in the sets of successfully disassembled, deactivated and moveable parts (Φ , Θ and M) boolean arrays `bDisassembled`, `bDeactivated` and `bMoveable` of size N are introduced in the source code. Thus, membership of a part in these sets can be checked efficiently by getting the corresponding boolean value in an array.

Apart from variable definition, `AssemblyTiers` implementation in VB.NET relies on CATIA API functions. First of all, the stepwise movement of parts is realised by calling `Apply(iTransformationArray)` function on the `Product.Move` property, whereby single parts are referred to as `Product` objects in CATIA. The transformation array contains nine elements of a rotation matrix and three – of a translation vector. Secondly, every collision detection is performed by manipulating a new `Clash` object. Its `ComputationType` is set to `catClashComputationTypeBetweenTwo` of `SPATypeLib.CatClashComputationType` enumerator. Depending on the task at hand, `InterferenceType` is `catClashInterferenceTypeClearance` or `catClashInterferenceTypeContact` of `SPATypeLib.CatClashInterferenceType` enumerator. Interferences between geometric bodies are saved as `Conflicts` property of the `Clash` object after `Clash.Compute()` function is called.

While the general outline of `AssemblyTiers` algorithm is given in Section 2.2.6, its concrete implementation is explained with Alg. 2 - 7. The preprocessing steps of deactivating fasteners and calculating removal distances for each part of an assembly are followed by the movement phase of the algorithm, during which active manipulation of parts and collision handling take place. After the movement phase, optional choice of disassembly directions for parts with multiple unobstructed removal axes is given to the user. Next, precedence relations between parts in sequential tiers are determined based on liaisons. Finally, data recorded in context of virtual disassembly are transformed into assembly data.

`AssemblyTiers` movement phase starts with initialisation of required counters and sets. In the beginning N parts including preselected base elements are present in the assembly. Parts get removed in CATIA in the course of progressing disassembly, so that variable I keeps track of remaining parts. Tier counter l is initialised with 1 and gets incremented as new groups of parts can be removed from the product. Each part deactivated after stage l is assigned tier l . However, the tiers recorded this way represent disassembly stage order and need to get reversed afterwards. Note that base components always retain tier 0 which requires no reversion. The tiers vector \mathbf{t} is instantiated as a null vector of length N . Counter variable i is used to obtain parts from set of all components II by index. The i_{cycle} counter monitors how many parts are tested at each disassembly stage. The total number of parts to be checked in each tier decreases from stage to stage. Since each part has to be moved in different directions, counter j determines which one to take and starts at 0 ($D_0 = x_{gl}$). Disassembly directions matrix \mathbf{Y} is populated with ones at start, so that certain entries can be overwritten with 0 whenever a part's movement is obstructed. Set of parts L becomes relevant whenever the option 'Connectivity check' is activated (s. Fig. 5.2b). In the beginning this set equals the entire set of relevant parts. Moveable components M of an assembly in the initial state contains all parts except bases.

Algorithm 2: AssemblyTiers movement phase

Input : $\Pi, \Delta\mathbf{X}, \mathbf{X}^*, s, \varepsilon, J, N, B$

Output: $\mathbf{S}, \mathbf{t}, \mathbf{Y}$

```

1  $I \leftarrow N$ ;
2  $l \leftarrow 1$ ;
3  $i, i_{cycle}, j \leftarrow 0$ ;
4  $L \leftarrow \Pi, M \leftarrow \Pi \setminus B, \mathbf{t} \leftarrow \mathbf{0}$ ;
5 while  $I > |B|$  do                                     // Tiers loop
6   while  $i_{cycle} < I - |B|$  do                             // Components loop
7     if  $p_i \in M$  then                                       // Current part is moveable
8       if  $SubassemblyIsConnected(L \setminus \{p_i\}) = \text{True}$  then
9         // Populate collision detection groups
10         $G_1 \leftarrow \{p_i\}$ ;
11         $G_2 \leftarrow \Pi \setminus p_i \cup \Theta$ ;
12        while  $j < J$  do                                     // Directions loop
13          while  $ProductReachedFinalPosition(p_i) = \text{False}$  do
14             $\text{MoveProduct}(p_i, s, D_j)$ ;
15            if  $\text{CollisionDetected}(G_1, G_2, \varepsilon) = \text{True}$  then
16               $\text{CollisionHandling}$ ;
17            end if
18          end while
19           $\text{Move } p_i \text{ to } \mathbf{X}_i^*$ ;                               // Return current part to initial position
20           $j \leftarrow j + 1$ ;                                   // Take next disassembly direction
21        end while
22      end if
23       $i_{cycle} \leftarrow i_{cycle} + 1$ ;                         // Update counter of checked parts
24    end if
25     $i \leftarrow i + 1$ ;                                       // Get next part in assembly
26     $j \leftarrow 0$ ;                                           // Disassembly direction indices start with 0
27  end while
28  TiersUpdate;
29  RecalculateRemovalDistances;
30   $l \leftarrow l + 1$ ;                                       // Increment tier counter
31   $i \leftarrow 0$ ;                                           // Begin next tier cycle with the first part
32   $i_{cycle} \leftarrow 0$ ;                                     // Reset counter of checked parts
33 end while

```

The contents of tiers loop in Alg. 2 establish the main procedure that runs until only base components are left in the assembly ($I = |B|$). This loop includes processing of each remaining moveable part, during which disassembly possibilities are determined. First, membership of part p_i in the set of moveable parts M is checked. In the case that p_i is a base component or already deactivated, part index i is increased by 1 and a new part iteration begins. Otherwise, the stability of assembly without p_i has to be examined. In simpler terms, it is necessary to know whether removing a certain part will divide the product in two subassemblies, one of which includes no base components and is thus unsupported, i.e. cannot be built upon base parts.

The discussed connectivity check is realised in `SubassemblyIsConnected` function. Consider the liaison graph of an exemplary product shown in Fig. 5.5. Assuming that p_i is part 6 (volute casing cover), `SubassemblyIsConnected` would try to remove the corresponding node and all its edges from the liaison graph to examine whether the remaining graph is connected, i.e. there is a path from any node to any other node. Removing part 6 would break up liaison graph connectivity since there would be no paths from part 7 to the rest of the graph.

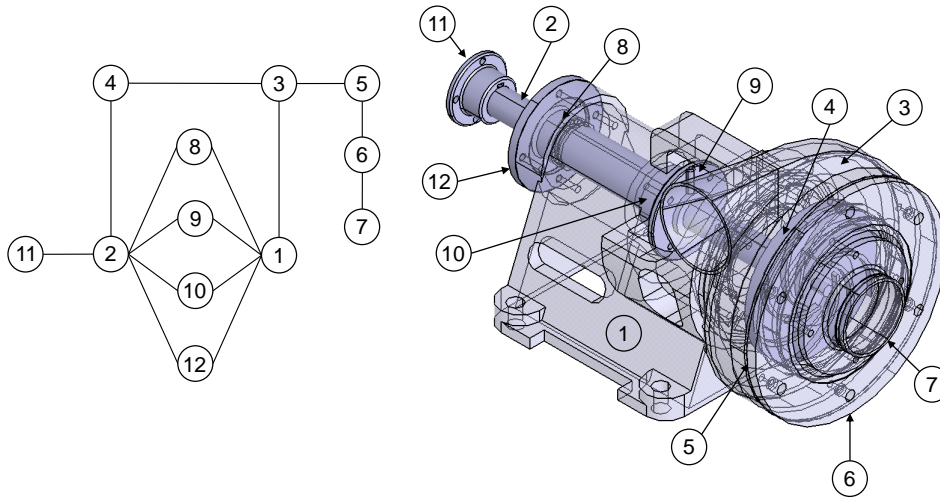


Figure 5.5: Liaison graph of the centrifugal pump

Alg. 3 describes the procedure responsible for subassembly connectivity checking. An important requirement for this algorithm omitted in the main pseudocode Alg. 2 is the extraction of an entire assembly's liaison graph using CATIA contact detection functions. The data structure employed to represent the graph in a computer-readable form is the symmetrical adjacency matrix of liaison graph $A \in \mathbb{R}^{N \times N}$ with $A_{i,k} = 1$ if there is a mechanical contact between parts p_i and p_k and $A_{i,k} = 0$ otherwise. Contacts of parts to themselves are not allowed: $A_{i,i} = 0$.

There are three stages in `SubassemblyIsConnected` algorithm. Firstly, a submatrix of A is constructed using the indices of parts in $L \setminus \{p_i\}$. Secondly, the graph represented by resulting submatrix A' is explored via depth-first search (DFS) technique starting from the node with the smallest index. An array of visited nodes \mathbf{v} is populated with boolean values indicating whether a path leading to a node exists. Finally, the algorithm iterates over \mathbf{v} and returns `False` as soon as a node is found which cannot be visited. This indicates that the graph encoded by A' is not connected. In the case that all nodes can be visited from the starting node, `True`

is returned.

Algorithm 3: SubassemblyIsConnected

```

Input :  $\Lambda, L, p_i$ 

Output: True if liaison subgraph with nodes  $L \setminus \{p_i\}$  is connected, False otherwise

1  $\sigma \leftarrow$  list of part indices  $q$  with  $p_q \in L \setminus \{p_i\}$ ;
2  $c \leftarrow |L \setminus \{p_i\}|$ ; // Count of parts in  $L$  without  $p_i$ 
3  $\mathbf{v}_n \leftarrow \text{False} \quad \forall n \in [0, c-1], n \in \mathbb{N}$ ; // No nodes are visited first
4  $\Lambda' \in \mathbb{R}^{c \times c}$ ;
5 for  $m \leftarrow 0$  to  $c-1$  do
6   for  $k \leftarrow 0$  to  $c-1$  do
7      $\Lambda'_{m,k} \leftarrow \Lambda_{\sigma_m, \sigma_k}$ ; // Construct liaison subgraph adjacency matrix
8   end for
9 end for
10 DFS( $\Lambda', \mathbf{v}, 0$ ); // Explore liaison subgraph nodes
11 for  $n \leftarrow 0$  to  $c-1$  do
12   if  $\mathbf{v}_n = \text{False}$  then
13     return False;
14   end if
15 end for
16 return True;

```

If a part can be removed from assembly without breaking up liaison graph connectivity, two groups of parts have to be defined between which CATIA should detect collisions which may occur during part movements. The first group G_1 only contains the current part p_i . The second group can be created differently depending on 'BB projection check' setting (s. Fig. 5.2b). BB projection check functionality is introduced to decrease the computational cost of collision analysis by reducing the number of parts against which clash detection is executed. Second group of parts contains all parts except p_i and deactivated parts Θ if BB projection check is not activated. In the other case only parts with bounding boxes that have overlapping projections with BB projections of p_i are appended to the second group. BBs are projected along the global assembly axes. If BB projections of two parts do not overlap in any disassembly direction, then no collisions between the two parts are possible during disassembly along principal axes. Thus, computationally expensive intersection analysis between such geometric bodies can be avoided. The function shown in Alg. 4 is used to determine which parts p_k are to be added to the second collision detection group.

Algorithm 4: BoundingBoxesOverlap**Input** : i, k **Output** : True if BB projections of p_i and p_k overlap in at least one global axis direction, False otherwise

```

1 if  $\tilde{x}_{min,i} \geq \tilde{x}_{max,k} \vee \tilde{x}_{max,i} \leq \tilde{x}_{min,k}$  then
2   if  $\tilde{y}_{min,i} \geq \tilde{y}_{max,k} \vee \tilde{y}_{max,i} \leq \tilde{y}_{min,k}$  then
3     if  $\tilde{z}_{min,i} \geq \tilde{z}_{max,k} \vee \tilde{z}_{max,i} \leq \tilde{z}_{min,k}$  then
4       return False;
5     end if
6   end if
7 end if
8 return True;

```

The actual disassembly movement procedure commences after collision groups are defined. The part of interest is moved stepwise in direction D_j until it reaches a distance at which a collision-free removal from assembly is guaranteed or a collision occurs between G_1 and G_2 that blocks further movement in the same direction. Either way, p_i is returned to its initial position in global coordinates and the next disassembly direction is checked.

Instructions that are to be followed in case of a collision are grouped into `CollisionHandling` method (s. Alg. 5). First of all, the obstacles from G_2 on the way of p_i need to be determined. Depending on the obstacles' tiers different actions are taken. Consider that base components as well as not yet removed parts have tier 0. A collision with one of these components prohibits assembly movement in the same direction as before. Thus, an entry in disassembly directions matrix \mathbf{Y} is made to denote that p_i cannot be removed along axis D_j . Since further movement is blocked, the innermost while-loop of the main algorithm is exited, whereafter the part is put back into its initial position. The same behaviour is triggered by any collision that happens during the first tier iteration. The second rule ensures that precedence relations are recorded between parts of consecutive tiers even if there are no direct liaisons between such parts, because assembly can be obstructed by parts from a previously removed tier. For instance, consider the assembly of impeller (part 4) in the centrifugal pump product (s. Fig. 5.5) when the volute casing cover (3), the shaft (2) and the volute casing gasket (5) are already assembled. The assembly of part 4 is obstructed if part 6 is fixed upon the product subassembly. In fact, parts 4 and 6 belong to sequential tiers but do not have any direct liaisons between them (s. liaison graph in Fig. 5.5). Not applying the second rule in Alg. 5 would result in severe information loss and would disregard the requirement to assemble part 4 before 6. It is important to note that a collision with a part from the previous tier does not block further movements of p_i and is only relevant for recording of precedence relations.

Algorithm 5: CollisionHandling

```

1 foreach obstacle  $p_k$  do
2   if  $t_k = 0 \vee l = 1$  then                                     // Further movement is blocked
3      $Y_{i,j} \leftarrow 0$ ;                                       // Record obstructed disassembly direction
4     go to Alg. 2 ln. 18;                                       // Interrupt disassembly movement
5   end if
6   if  $t_k = l - 1 \wedge l \neq 1$  then                             // Collision with parts from the previous tier
7      $S_{i,k} \leftarrow 1$ ;                                       // Record precedence relation, continue movement
8   end if
9 end foreach

```

Whenever part p_i reaches disassembly-safe position, True is assigned to Φ_i . After every disassembly direction is tested for a part, Φ_i provides information whether there is at least one obstacle-free disassembly direction for the part of interest. In the connectivity check mode, this information is used to remove p_i from L , thus reducing the subset of parts which is considered in subassembly connectivity test for the next part, p_{i+1} . This measure ensures that subassembly connectivity is preserved after a tier iteration, because even if removing single parts does not disintegrate the product, disassembly of multiple such parts simultaneously could still result in unsupported subassemblies. This can be illustrated with a simple five-part wheel assembly shown in Fig. 5.6. Let part 1 be the base component. AssemblyTiers algorithm with activated connectivity check mode can determine that liaison graph connectivity is preserved after removing part 2 (one of the wheel supports). This part can also be freely disassembled in an outward movement along the wheel axle 4. Set of parts relevant for connectivity checks L is reduced to $\{1, 3, 4, 5\}$ even though wheel support 2 stays active in the collision detection scene. The next candidate for disassembly, wheel support 3, cannot be removed because it would result in subassembly $\{4, 5\}$ being disconnected from the base 1. Without intermediate update of L the algorithm would allow disassembly of parts 2 and 3 in the first tier. Removing either part 2 or part 3 alone does not lead to unsupported axle and wheel. However, removing them both in the same tier does.

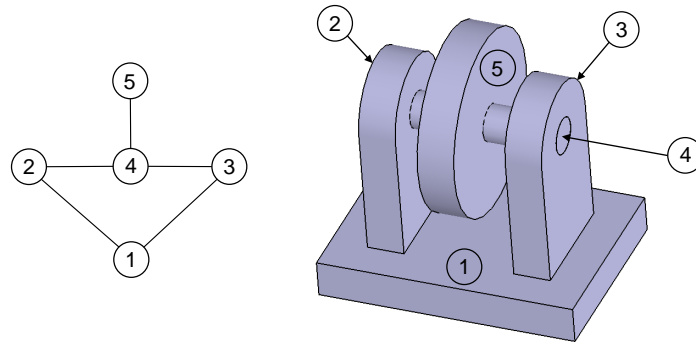


Figure 5.6: Wheel assembly for connectivity check illustration

Deactivation of parts takes place at TiersUpdate stage after iterations through all available parts (Alg. 2 ln. 27). The methods involved in TiersUpdate are shown in Alg. 6 and can be divided into three steps. First of all, newly disassembled parts are assigned the current tier l , highlighted in CATIA for better visualisation and

extracted from the set M to be skipped in the next tier. Parts disassembled in the previous tier are deactivated and hidden in CATIA. Secondly, if only base components are left after the previous step, all non-base parts are deactivated directly and the outermost tier loop of `AssemblyTiers` can be exited. Last but not least, the counter of remaining parts I is compared to the temporary variable I_{temp} which stores the number of parts before deactivation. In case that no parts can be removed (and I does not decrease), remaining non-base components are deactivated and the tiers loop is left.

Since the removal of components can reduce the dimensions of assembly's BB, removal distances of parts along certain directions can also shorten. Thus, recalculating the removal distances is practical because it can reduce the simulation time needed to extract parts from an assembly. This update function is called at Alg. 2 ln. 28.

`RecalculateRemovalDistances` method is responsible for updating $(X_{max}, X_{min}, Y_{max}, Y_{min}, Z_{max}, Z_{min})$ assembly bounding box by iterating through the entire set of assembly's parts II and calculating the global assembly BB only accounting for parts that are not members of Θ (deactivated components). This process is followed by recalculation of removal distances using unchanged global part BBs $\tilde{\mathbf{X}}$ and updated assembly

minimum and maximum coordinates.

Algorithm 6: TiersUpdate

```

1   $I_{temp} \leftarrow I$ ;
2  for  $i \leftarrow 0$  to  $N - 1$  do                                     // Record tiers
3      if  $p_i \in B$  then
4           $t_i = 0$ ;                                             // Base components always have tier 0
5      else
6          if  $\Phi_i = True \wedge t_i = 0$  then                     // Disassembled parts without an assigned tier
7               $t_i \leftarrow l$ ;
8               $I \leftarrow I - 1$ ;                             // Decrease the counter of parts left in assembly
9              Change visuals of  $p_i$  to indicate disassembled tier;
10              $M \leftarrow M \setminus \{p_i\}$ ;                 // Disable movement of this part in next tier
11         end if
12         if  $\Phi_i = True \wedge t_i = l - 1 \wedge l > 1$  then // Disassembled parts from the previous tier
13             Deactivate  $p_i$  in CATIA;
14              $\Theta \leftarrow \Theta \cup \{p_i\}$ ;
15         end if
16     end if
17 end for
18 for  $i \leftarrow 0$  to  $N - 1$  do
19     if  $I = |B| \wedge t_i = l$  then                             // All parts disassembled
20         Deactivate  $p_i$  in CATIA;
21          $\Theta \leftarrow \Theta \cup \{p_i\}$ ;                     // Deactivate all parts from last tier directly
22     end if
23 end for
24 if  $I = I_{temp}$  then                                         // No parts could be removed in this tier
25     for  $i \leftarrow 0$  to  $N - 1$  do
26         if  $t_i = 0 \wedge p_i \notin B$  then                     // Remaining non-base components
27             Deactivate  $p_i$  in CATIA;
28              $\Theta \leftarrow \Theta \cup \{p_i\}$ ;
29         end if
30     end for
31     exit AssemblyTiers movement phase;
32 end if

```

To continue the main algorithm cycle in the next stage, tier counter l is incremented by 1, part index and counter of checked parts are reset to 0 (s. Alg. 2 ln. 29-31). Processing the last tier is followed by AssemblyTiers stationary phase (Alg. 7), during which recorded disassembly directions and tiers are transformed into usable

output and additional precedence relations between components are determined.

Algorithm 7: AssemblyTiers stationary phase

```

1  $l_{max} \leftarrow l - 1;$  // Highest recorded tier
2  $\bar{j} \in \mathbb{N} : D_{\bar{j}} = -D_j;$  // Index of the opposite direction
3  $K = \{(a, b) : A_{a,b} = 1, a > b\};$  // Part index pairs representing liaisons
4 for  $i \leftarrow 0$  to  $N - 1$  do
5   if  $p_i \notin B$  then // Non-base components
6     Activate  $p_i$  in CATIA;
7   end if
8   if  $t_i \neq 0$  then
9      $t_i \leftarrow l_{max} + 1 - t_i;$  // Reverse all tiers except for base parts
10  end if
11   $y \leftarrow 0, \quad y \in \mathbb{R}^J;$  // Temporary assembly directions vector
12  for  $j \leftarrow 0$  to  $J - 1$  do
13    if  $Y_{i,j} = 1$  then
14      // Reverse disassembly directions to obtain assembly directions
15       $y_{\bar{j}} \leftarrow 1;$ 
16    end if
17  end for
18   $Y_i \leftarrow y;$ 
19 end for
19 foreach  $(a, b) \in K$  do
20   // Precedence relations of components belonging to sequential tiers
21   if  $t_a = t_b - 1$  then
22      $S_{a,b} \leftarrow 1;$ 
23   end if
24   if  $t_b = t_a - 1$  then
25      $S_{b,a} \leftarrow 1;$ 
26   end if
26 end foreach

```

Vector t stores parts' disassembly tiers at main algorithm runtime, i.e. parts that can be removed first are assigned smaller tier values. However, the main purpose of the algorithm is to create a meaningful representation of assembly logic, which means that the logical order has to be reversed in accordance with 'assembly-by-disassembly' strategy. This procedure (Alg. 7 ln. 9) is implemented as shown in [Pin16b, p. 1051]. Next, the disassembly directions matrix Y is converted to store feasible assembly axes by means of swapping values of opposite directions if one of them equals 1. A dictionary is instantiated beforehand which supplies the algorithm with opposite axes mapping (s. Tab. 5.1). The dictionary provides the opposite axis index \bar{j} for a given disassembly axis j .

D_j	$+x_{gl}$	$+y_{gl}$	$+z_{gl}$	$-x_{gl}$	$-y_{gl}$	$-z_{gl}$	$+x_{loc}$	$+y_{loc}$	$+z_{loc}$	$-x_{loc}$	$-y_{loc}$	$-z_{loc}$
j	0	1	2	3	4	5	6	7	8	9	10	11
\bar{j}	3	4	5	0	1	2	9	10	11	6	7	8

Table 5.1: Index mapping for opposite direction retrieval

In the final step of AssemblyTiers stationary phase remaining precedence relations between parts belonging to sequential tiers are determined by analysing contacts. Every pair of parts with a liaison connecting them is checked for the difference between tiers. Whenever it equals 1, a directed edge in the assembly precedence graph is created pointing from the part with a lower tier to the other component.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	1	0	0	0	1	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	1
9	0	1	1	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	1	0

(a) 'Precedence Matrix' worksheet

	1	2	3	4	5	6	7	8
1	Product	+X	+Y	+Z	-X	-Y	-Z	Assembly Tier
2	BEARING HOUSING .1							0
3	shaft .1	0	1	0	0	0	0	3
4	volute casing .1	0	0	0	0	1	0	3
5	impeller .1	0	0	0	0	1	0	4
6	volute casing gasket .1	0	0	0	0	1	0	4
7	volute casing cover .1	0	0	0	0	1	0	5
8	suction nozzel .1	0	0	0	0	1	0	6
9	bearing size 40 50.1	0	1	0	0	0	0	4
10	Bearing housing end cap 2.1	0	0	0	0	1	0	2
11	bearing size 59 70.1	0	0	0	0	1	0	1
12	coupling .1	0	1	0	0	0	0	6
13	Bearing house end cap .1	0	1	0	0	0	0	5

(b) 'Assembly Directions' worksheet

Figure 5.7: AssemblyTiers output as an Excel spreadsheet

Assembly tiers method is concluded by writing precedence matrix S , assembly directions Y and tiers t to a Microsoft Excel spreadsheet. The implemented program automatically creates a reference to a new Excel COM object by calling the function `CreateObject("Excel.Application")`. Therein, a `Workbook` object is instantiated. The first Excel Sheet added to the workbook is called 'Precedence Matrix' and the second – 'Assembly Directions'. The first sheet contains values of S (Fig. 5.7a), while the second combines information

about part names, allowed assembly directions and tiers (Fig. 5.7b).

5.1.3 AND/OR graph prerequisite extraction

In accordance with the conceptual framework presented in Fig. 4.2, two alternatives are to be implemented as basis for scheduling. The alternative B requires preparation of liaison and moving wedge (MW) matrices for subsequent AND/OR hypergraph creation. Throughout this section, movement step s , collision sensitivity ε and bounding box projection check settings are assumed to be the same as in `AssemblyTiers` program. Due to the definition of MW matrices given in [SS02], only movements along global axes are considered. The actual derivation of an AND/OR graph is treated in section 5.1.4 because it is not part of developed CATIA interface but is outsourced to another software module which operates on data provided by the CAD interface.

Generation of an assembly AND/OR graph is based on respecting stability and geometrical feasibility predicate of assembly operations (requirements ASP2 and ASP3 in Tab. 3.2). In contrast to assembly tiers method, subassemblies are allowed to be built and manipulated as a whole. Considering this, a valid algorithm must be able to determine whether a subassembly can be created from its components and then brought into the assembly collision-free and respecting the stability predicate at the same time. Similar to assembly tiers method, liaison graph connectivity check provides means of ensuring subassembly stability in a simplified form, i.e. assuming that every liaison is a rigid, dynamically stable connection. In order to account for the entirety of possible collisions that can occur between parts and subassemblies moving along main assembly directions, MW matrices in accordance with [SS02] are applied in this thesis.

Prior to relevant data acquisition fasteners are deactivated using `DeactivateFasteners` function. Liaison graph extraction is realised by the same method as in `AssemblyTiers` and provides Δ as output. The result is written into a separate Excel file which can be later accessed by the AND/OR generation algorithm. For instance, the liaison graph representing contacts between components of the centrifugal pump assembly is saved as `Centrifugal_pump_Liaisons.xlsx`.

Generating bounding boxes of components in an assembly is necessary to efficiently determine when a part can be safely disassembled. This preprocessing step takes place before any part is manipulated by the algorithm. Moving wedge extraction resembles the movement phase of `AssemblyTiers` algorithm but shows different

behaviour on collisions. The logical outline of MovingWedge algorithm is presented in Alg. 8.

Algorithm 8: MovingWedge

Input : $\Pi, \Delta\mathbf{X}, \mathbf{X}^*, s, \varepsilon, N$

Output: MW

```

1  $MW_{i,k}(\delta) \leftarrow 1 \quad \forall i, k \in \{0, \dots, N-1\}, \quad \forall \delta \in \{+x, +y, +z\};$ 
2 for  $i \leftarrow 0$  to  $N-1$  do
    // Populate collision detection groups
3    $G_1 \leftarrow \{p_i\};$ 
4    $G_2 \leftarrow \Pi \setminus p_i;$ 
5   for  $j \leftarrow 0$  to 2 do                                     // Directions loop
7     while  $ProductReachedFinalPosition(p_i) = False$  do       // Movement loop
9        $MoveProduct(p_i, s, D_j);$ 
10      if  $ProductReachedFinalPosition(p_i) = True$  then
11         $Move\ p_i\ to\ \mathbf{X}_i^*;$                                 // Return part to initial position
12        break;                                                // Take next movement direction
13      end if
14      if  $CollisionDetected(G_1, G_2, \varepsilon) = True$  then
15        foreach obstacle  $p_k$  do
16          if  $j = 0$  then
17             $MW_{k,i}(+x) \leftarrow 0;$ 
18          else if  $j = 1$  then
19             $MW_{k,i}(+y) \leftarrow 0;$ 
20          else if  $j = 2$  then
21             $MW_{k,i}(+z) \leftarrow 0;$ 
22          end if
23        end foreach
24      end if
25    end while
26  end for
27 end for
  
```

MW matrix elements are initialised with 1, which corresponds with the assumption that no parts obstruct any movements of other parts. The purpose of MovingWedge algorithm is to discover which assembly operations are not possible given the geometric forms of a product's components. The limits on assembly directions are then to be recorded by overwriting corresponding MW values with 0. Similar to the assembly tiers method, two collision detection groups are created for each part, which is then moved along three principal global axes, while recording occurring clashes. However, the part continues its movement until it reaches a disassembly-safe distance (indicated by a boolean output of `ProductReachedFinalPosition` method). An important property of MW matrices is used to reduce the duration of collision analysis and the amount of data to be saved:

$$MW(\delta)^T = MW(-\delta) \quad (5.10)$$

Equation 5.10 allows a moving wedge matrix for one axis to contain information about both positive and negative assembly directions along that axis, so that a single matrix for each principal axis is sufficient to record all relevant collision data. This property results from the invariance of collisions to relative movement of parts along the same axis but in different directions. To put it in simpler terms, if a part p_i moving in the positive axis direction δ collides with another stationary part p_k , it implies that the same collision also occurs if the part p_k is moved in the negative direction of the same axis $(-\delta)$ and p_i is fixed:

$$MW_{i,k}(\delta) = MW_{k,i}(-\delta) \quad \forall i, k \in \{0, \dots, N-1\} \quad (5.11)$$

Thus, it is not necessary to move each part in both positive and negative axis directions. The information about relative movements of parts and their collisions can be obtained by examining simulated disassembly movements in the positive axis directions. Note that MW matrices contain information about assembly movements and it implies permuting indices i and k in Alg. 8 ln. 14-20.

Prerequisites for generating an assembly AND/OR graph are ready after MW data are written to a dedicated Excel file with three worksheets 'MW_x', 'MW_y' and 'MW_z'. Next section presents two alternative algorithms developed to transform liaison and MW data into a complete AND/OR graph.

5.1.4 AND/OR graph generation

The algorithms related to AND/OR generation are implemented in Python 3 and make use of various libraries to reduce development time. First of all, the liaison and MW data need to be read. For this purpose, Pandas data analysis library is applied [Reb20]. An instance of `ExcelFile` is created by the specified path to allow data extraction from Excel. Next, 'Liaison Matrix' worksheet is parsed into a `DataFrame` object by Pandas' `parse()` function. The first five rows of the read `DataFrame` are printed in the console for debugging purposes. The same procedure is followed for MW matrices contained in their respective worksheets. Liaison dataframe is stored in `liaison_df` variable, MW data is saved in a dictionary of dataframes `mw_dfs`, which can be retrieved by their names ('MW_x', 'MW_y' or 'MW_z'). Secondly, the entire product Π is represented by a list of parts `prod`, which contains 1-based indices of parts: `prod = list(range(1, len(liaison_df) + 1))` for more intuitive output and visualisation at later stages of the algorithm.

Since no performance benchmarks are present in the literature, two alternative approaches for AND/OR derivation are programmed in this thesis for examination: top-down and bottom-up.

Top-down approach

This method was developed by Thomas in the dissertation [Tho08] and its application in self-optimising assembly systems based on cognitive technologies is mentioned in [Bre12, p. 911]. The core idea of the top-down approach is to apply 'assembly-by-disassembly' strategy beginning with the entire product, but instead of removing single parts as in the assembly tiers method, pairs of all subsets are examined for collision-free

separation. Valid disassembly operations are appended to the AND/OR graph and resulting subassemblies are evaluated and tested in further disassembly attempts. [Bre12]

The top-down procedure is a Python script called `and_or.py`, which can be launched in a conventional Python IDE, e.g. Spyder. After the described data import an empty list of nodes Ω is created so that resulting subsets of parts can be stored and skipped if they have already been 'visited' by the recursive procedure. This recursively executed algorithm is called `and_or()` in the code and requires four arguments: a subset of parts to disassemble $\Pi' \subseteq \Pi$, liaison matrix A , MW matrices and the list of visited nodes Ω . HyperNetX library [Bat18] is employed for graph storage and optional visualisation, so that string representations of subassemblies are required. The entire top-down `and_or()` function is presented in the pseudocode Alg. 9.

The procedure begins with generating all possible partitions Π'' of the provided subassembly Π' in two non-empty sets:

$$\Pi''(\Pi') = \{\{\pi_1, \pi_2\} : \pi_1 \cup \pi_2 = \Pi', \pi_1, \pi_2 \neq \emptyset\} \quad (5.12)$$

For example, a set $\{1, 2, 3\}$ has three such partitions: $\{\{1, 2\}, \{3\}\}, \{\{1, 3\}, \{2\}\}, \{\{2, 3\}, \{1\}\}$. Applying `bin_partitions()` function to Π' yields every partition containing two subassemblies each. Consider that every element of the initial part subset Π' can be assigned 0 or 1 to denote membership of the element in one of the two resulting subassemblies. The entire set of partitions can then be enumerated by integers $0 \leq n \leq 2^{|\Pi'|} - 1$ encoded in binary system as shown in Tab. 5.2. Furthermore, since a permutation of subassemblies π_1 and π_2 still results in the same initial part subset, an enumeration with a half of the numbers is sufficient. Since empty subsets are not meaningful for assembly planning, the first and the last rows in Tab. 5.2 are irrelevant. To conclude, numbers $1 \leq n \leq 2^{|\Pi'|} - 1$ suffice for the partition encoding.

n	1	2	3	π_1	π_2
0	0	0	0	1, 2, 3	–
1	0	0	1	1, 2	3
2	0	1	0	1, 3	2
3	0	1	1	1	2, 3
4	1	0	0	2, 3	1
5	1	0	1	2	1, 3
6	1	1	0	3	1, 2
7	1	1	1	–	1, 2, 3

Table 5.2: Set partition encoding via binary number enumeration

Algorithm 9: AND/OR top-down approach**Input** : $\Pi', \Lambda, MW, \Omega$ **Output**: Assembly AND/OR hypergraph

```

1  $\Pi'' \leftarrow \text{bin\_partitions}(\Pi')$ ;
2 foreach  $\{\pi_1, \pi_2\} \in \Pi''$  do
3   if  $\text{is\_stable}(\pi_1, \Lambda) = \text{False} \vee \text{is\_stable}(\pi_2, \Lambda) = \text{False}$  then
4     continue;
5   end if
6    $Y \leftarrow \emptyset$ ; // Set of collision-free assembly direction indices
7   for  $j \leftarrow 0$  to 5 do
8      $c \leftarrow 0$ ; // Counter of collision-free part pairs for direction  $D_j$ 
9     if  $j < 3$  then // Positive axis directions
10       $MW' \leftarrow MW(D_j)$ ;
11      for  $(p_i, p_k) \in \pi_1 \times \pi_2$  do
12         $c \leftarrow c + MW'_{i,k}$ ;
13      end for
14    else // Negative axis directions
15       $MW' \leftarrow MW(D_{j-3})$ ;
16      for  $(p_i, p_k) \in \pi_1 \times \pi_2$  do
17         $c \leftarrow c + MW'_{k,i}$ ;
18      end for
19    end if
20    if  $c = |\pi_1| \cdot |\pi_2|$  then // Assembly direction is collision-free
21       $Y \leftarrow Y \cup D_j$ ;
22    end if
23  end for
24  if  $|Y| > 0$  then // There is at least one collision-free assembly direction
25    Save hyperedge  $(\Pi', \pi_1, \pi_2)$ ;
26    if  $\pi_1 \notin \Omega$  then
27       $\Omega \leftarrow \Omega \cup \pi_1$ ;
28       $\text{and\_or}(\pi_1, \Lambda, MW, \Omega)$ ;
29    end if
30    if  $\pi_2 \notin \Omega$  then
31       $\Omega \leftarrow \Omega \cup \pi_2$ ;
32       $\text{and\_or}(\pi_2, \Lambda, MW, \Omega)$ ;
33    end if
34  end if
35 end foreach

```

Every partition consisting of two sets $\{\pi_1, \pi_2\}$ has to be checked for stability predicate and geometrical feasibility. At first, connectivity of both subassemblies is tested by executing a depth-first search on their corresponding liaison subgraphs and determining whether they are connected. Whenever one of the subassemblies is not connected, the next partition possibility is analysed.

Before testing the geometrical feasibility of putting subassemblies π_1 and π_2 together, an empty set of collision-free assembly directions Y is initialised. Determining whether π_1 can be assembled to π_2 along a certain direction requires analysis of MW submatrices for involved axis and subassemblies. A collision between two groups of parts can only occur if a part from the first group collides with a part from another group. Vice versa, a subassembly can be attached to another subassembly collision-free if any pair of parts contained in opposite groups does not cause any clashes during assembly. This condition can be checked by counting ones in the corresponding submatrix of MW for the axis of interest and comparing the sum c to the total number of part pairs $|\pi_1| \cdot |\pi_2|$. If the numbers are equal, the assembly axis is appended to the set of collision-free directions. A new assembly operation is recorded for partitions which have at least one such direction.

Hyperedges in the AND/OR graph are represented by 3-tuples, in which the first element is the part subset that is created by the assembly operation and the remaining two elements are the components or subassemblies that are combined to form II' . Each of subassemblies π_1 and π_2 are added to the set of visited nodes Ω if their decomposition has not been analysed, which is followed by a recursive call of `and_or()` function on unvisited subassemblies.

Once the complete graph is created by the procedure, it is accessible as an instance of `Hypergraph` class, which can be visualised as a rubber-band diagram. Each rubber-band embraces three subassembly nodes and represents an assembly operation. Rubber-bands are given distinct colours and IDs. The graph itself is serialised using Python module `pickle` and can be retrieved by process graph generation and scheduling programs.

Bottom-up approach

In contrast to the top-down AND/OR graph generation, Gulivindala et al. propose an approach that generates assembly sequences starting from single parts and gradually creating bigger subassemblies at higher levels. Their approach aims at finding a parallel assembly sequence with the smallest number of assembly levels using a heuristic method. [Gul20]

In this thesis, a similar approach at generating the entire assembly AND/OR graph is implemented to account for influences from the assembly system which can make a simple heuristic suboptimal. A separate Python script called `and_or_bottom_up.py` is created for this purpose. The presented bottom-up algorithm operates on the same input data as in the top-down algorithm implementation. In the context of the bottom-up approach, Ω represents the list of subassemblies created by the algorithm so far. At first, the list of created subassemblies is initialised with single parts. Existing subassemblies are combined on higher levels l to form bigger subassemblies. At the beginning, only pairs of parts can be created, while an increasing number of combinations is possible on higher levels because of the growing set of created subassemblies.

Several conditions are to be checked in order to create new combinations of parts and subassemblies (see Alg. 10 In. 8). Firstly, the total number of parts in two subassemblies cannot exceed the total product part count,

because it implies that there are same parts present in both subassemblies. Secondly, even if the first condition holds, the eventual intersection of the two subsets of parts π_1 and π_2 needs to be examined. If no same parts appear in the two subsets, these subassemblies are put together and tested for connectivity. Furthermore, the subassemblies are checked for existence of collision-free assembly directions. Notwithstanding the examined conditions, a subassembly may still be created that prevents further necessary assembly operations, i.e. there are parts that can no longer be assembled in any directions once a certain subassembly is built, which is to be avoided. In the case that all previously mentioned conditions are satisfied, a new assembly hyperedge is created in the same manner as in the top-down algorithm.

Algorithm 10: AND/OR bottom-up approach

Input : Π, Λ, MW, Ω

Output: Assembly AND/OR hypergraph

```

1  $\Omega \leftarrow \{\{p_i\} : p_i \in \Pi\};$  // Populate set of created subassemblies with single parts
2  $k_{start} \leftarrow 0;$  // Smallest index of the second component subassembly
3 for  $l \leftarrow 2$  to  $N$  do // Maximum number of assembly levels is  $N$ 
4    $\Omega' \leftarrow \Omega;$  // Temporary updated set of created subassemblies
5   for  $i \leftarrow 0$  to  $|\Omega|$  do // First component subassembly index
6     for  $k \leftarrow k_{start}$  to  $|\Omega|$  do // Second component subassembly index
7       if  $i < k$  then // Skip repeated assembly operations
8         if  $\pi_1 = \Omega_i$  is assemblable to  $\pi_2 = \Omega_k$  then
9            $\Pi' \leftarrow \pi_1 \cup \pi_2;$ 
10          Save hyperedge  $(\Pi', \pi_1, \pi_2);$ 
11          if  $\Pi' \notin \Omega'$  then // Resulting subassembly is new
12             $\Omega' \leftarrow \Omega' \cup \Pi';$  // Append new resulting subassembly
13        end for
14    end for
15     $k_{start} \leftarrow |\Omega|;$  // Reset second subassembly index to avoid examined combinations
16     $\Omega \leftarrow \Omega';$  // Rewrite set of created subassemblies with updated version
17 end for

```

5.1.5 Process graph generation

The method presented in [Ehm19] is applied in order to construct an assembly specific process graph from AND/OR as a basis for scheduling. The main difference from the cited work is the form in which AND/OR is stored and used by the process graph construction algorithm. In contrast to the transition matrix, the developed algorithm works with AND/OR represented as a dictionary wherein each hyperedge ID is assigned a 3-tuple of subassemblies. The notation of indices and sets found in [Ehm19, p. 93] is applied with modifications to reflect the assembly logic. The explanations of important elements such as splitting and decision nodes can be found in Section 2.2.5.

The implemented algorithm is a part of `and_or_mip.py` script and iterates over assembly operations $k \in O_j$

present in the AND/OR graph and retrieves the subassembly created by k . The output subassembly is denoted as h . A new decision node with the same index as k is appended to the set R_j whenever the created subassembly h is not in the list of explored subassemblies. Operations $k' \in O_j$ that use h as a component are added to the set of outgoing operations R_{jr}^{out} for the decision node $r = k \in R_j$ and all operations k' that assemble h are added to the set of ingoing operations R_{jr}^{in} . Next, a splitting node s can be created for each ingoing operation k' , if both components h' involved in k' have at least one operation k'' that assembles each of them. The newly created splitting node s represents the possibility of simultaneous execution of such k'' . Analogously to the method developed by Ehm, dummy operations k^* need to be created wherever there are multiple operations k'' going into the splitting node s or multiple further outgoing assembly operations. The general outline of the algorithm is shown in Alg. 11.

Algorithm 11: Process graph construction

Input : Assembly AND/OR hypergraph with operations O_j

Output: $R_j, S_j, R_{jr}^{in}, R_{jr}^{out}, S_{js}^{in}, S_{js}^{out}$

```

1 foreach  $k \in O_j$  do
2    $h \leftarrow$  subassembly created by operation  $k$ ;
3   if  $h$  not explored then
4     Create decision node  $r = k \in O_j$ ;
5     Append  $h$  to explored subassemblies;
6   end if
7   foreach  $k' \in O_j$  do
8     if  $h$  is a component in  $k'$  then
9       Append  $k'$  to  $R_{jr}^{out}$ ;
10    else if  $h$  is output of  $k'$  then
11      Append  $k'$  to  $R_{jr}^{in}$ ;
12       $h'_{1/2} \leftarrow$  components in  $k'$ ;
13      if  $h'_{1/2}$  can be assembled  $\wedge k = k'$  then
14        Create splitting node  $s \in S_j$ ;
15        Append all  $k''$  to  $S_{js}^{in}$  that assemble  $h'_{1/2}$  (if  $k''$  is the only way to assemble  $h'_{1/2}$  and
          only one operation uses  $h'_{1/2}$  as component);
16        Append  $k'$  to  $S_{js}^{out}$ ;
17        Insert a dummy operation if there are  $>1$  ingoing operations creating or outgoing
          operations using  $h'_{1/2}$ ;
18         $s \leftarrow s + 1$ ;
19      end if
20    end if
21  end foreach
22 end foreach
23 Replace operations with dummies where needed;
24 Remove redundant decision nodes (logical copies of splitting nodes);

```

Fig. 5.8 illustrates how an AND/OR graph is transformed into a collection of decision and splitting nodes, preserving the logical assembly structure. A six-part product with a total of nine potential assembly operations is chosen for easier demonstration of the algorithm. In the first iteration, operation $k = 1$ results in the first decision node $r = 1$. This node has a special characteristic that it has no outgoing assembly operations, because operation 1 is the final assembly step and creates the final product.

Analysing $k = 2$ results in another decision node $r = 2$, which has an outgoing operation 1 and two ingoing operations 2 and 3. This node illustrates a choice between two alternative ways to assemble $h = \{1, 2, 4, 5, 6\}$. Then, examining the ingoing operation $k = k' = 3$ shows that both $h'_1 = \{1, 4, 5\}$ and $h'_2 = \{2, 6\}$ contain more than a single part and have assembly operations creating them. The first splitting node $s = 1$ is instantiated. Since the subassembly $\{1, 4, 5\}$ has two operations that create it (operations 5 and 6) and $\{2, 6\}$ is used as a component by operations 3 and 4, no real ingoing assembly operations are appended to $S_{j,1}^{in}$ at this step. However, dummy operations 10* and 11* are inserted.

During the iteration with $k = 4$ a corresponding decision node $r = 4$ is added to the process graph. The subassembly $h = \{1, 2, 5, 6\}$ is a component of operation $k' = 2$, which is then added to $R_{j,4}^{out}$. Operation $k' = 4$ assembles $\{1, 2, 5, 6\}$ and is thus appended to $R_{j,4}^{in}$. Both component subassemblies $h'_1 = \{2, 6\}$ and $h'_2 = \{1, 5\}$ can be assembled but are also both used in more than one further assembly operation each. Thus, a new splitting node $s = 2$ is created with a single outgoing operation 4. No real ingoing operations are inserted at this stage. Instead, dummies 12* and 13* are added.

The iteration $k = 5$ makes a new decision node $r = 5$ with ingoing operations 5 and 6 and an outgoing operation 3. This operation is recognised as redundant with the outgoing operation 3 from $S_{j,1}^{out}$. This redundancy is eliminated in the end of the whole procedure by replacing operation 3 in $R_{j,5}^{out}$ with the dummy 10*. Proceeding in the next iteration $k = 6$, the algorithm does not create a new decision node because the subassembly $h = \{1, 4, 5\}$ has already been explored in the previous iteration. Since operation $k = k' = 6$ uses part $\{5\}$ as a component, which is not created by any assembly operation, no new splitting nodes are required at this step.

The rest of the operations $k = 7, 8, 9$ are processed in the same manner. Note the importance of the final replacement of operations with dummies, during which the set of outgoing operations $R_{j,8}^{out} = \{4, 5\}$ is changed to $\{12^*, 5\}$ and $R_{j,9}^{out} = \{3, 4\}$ to $\{11^*, 13^*\}$.

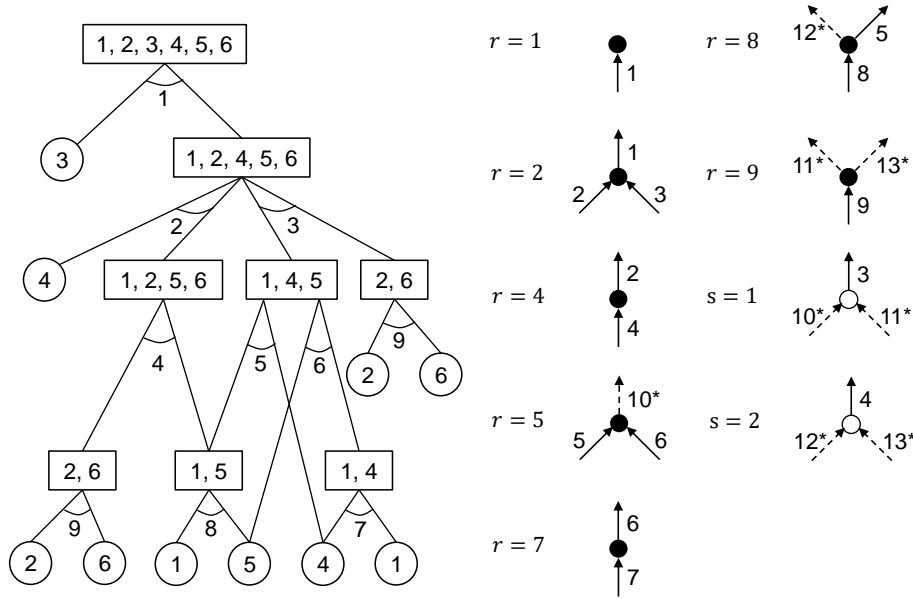


Figure 5.8: Transformation of an AND/OR graph adapted from [Ehm19, p. 92] into decision and splitting nodes of a process graph

5.2 Scheduling module

Two separate scripts are dedicated to scheduling with different assembly process models. The script for scheduling using precedence graphs is presented in Section 5.2.1 and can be found in `assembly_tiers_scheduling.py`. A detailed explanation of `and_or_mip.py` script is given in Section 5.2.2. While the program for scheduling based on precedence graphs does not require any additional information processing before starting the optimisation, scheduling based on AND/OR graphs requires preliminary translation into a process graph, which is handled in the previous section.

The process data necessary for scheduling are the characteristics of operations that need to be executed. First of all, the duration of each operation is assumed to be deterministic and proportional to the volume of component bounding boxes extracted previously by the CAD interface. Furthermore, the durations pt are integers between 0 and 10, which can stand for different time units depending on the coarseness of the schedule and the complexity of assembly operations. With $V_{BB,k}$ being the volume of the bounding box of part p_k or the sum of BB volumes of parts contained in a subassembly, the duration is calculated by the formula

$$pt_k = \left\lceil \frac{V_{BB,k} - \min_k V_{BB,k}}{\max_k V_{BB,k} - \min_k V_{BB,k}} \cdot 9 + 1 \right\rceil \quad (5.13)$$

Secondly, the part weights are stored in the vector ρ as integer numbers of kilograms. The values of ρ are distributed between 500 and 1500 kg in similarly to Eq. 5.2. In the test scenario, two types of cranes are available with load capacities of 1000 kg and 5000 kg, respectively. For two cranes, this information is formalised in the tuple $\lambda = (1000, 5000)$. Thirdly, the number of workers requested by each operation is an integer between 1 and 3 stored in the vector u . The values of u are deterministic and also proportional to BB volumes

of parts. The total number of workers available on-site c cannot be exceeded at any time interval. Using the mentioned data, it is possible to expand the standard RCPSP by additional constraints to reflect the restrictions relevant for on-site large-scale assembly. The standard RCPSP and additional constraints are formulated as MIPs which are then solved by Gurobi optimiser. The implementation of MIP constraints is covered in the book 'Mixed Integer Linear Programming with Python' by Santos and Toffolo [ST20], which provides the basis for the scheduling programs developed in this thesis.

5.2.1 Scheduling with precedence graphs

The precedence graph and the assembly tiers of product parts need to be extracted from Excel first. To do so, the string variable `product_name` is used to create a path to the file of interest automatically. The function `read_precedence_matrix()` opens the Excel file and parses the 'Precedence Matrix' worksheet into a new Pandas DataFrame object called `precedence_df`. The assembly tiers are extracted similarly by `read_assembly_tiers()`, where the object `at_df` stores the data of the 'Assembly Directions' worksheet. In the current implementation, the Excel files are assumed to be found in the folder `data` in the same directory as the Python script. The relative path to the required files can be modified in the code, if changes in the project structure are needed in the future.

Once the data is imported in Python, the precedence matrix needs to be transformed into a list of edges to match the constraint implementation from [ST20]. First, the applied function `edge_list()` takes `precedence_df` as an argument and calculates column and row sums of the precedence matrix. Then, a directed edge $(i + 1, j + 1)$ ¹ is created for each matrix element with $S_{i,j} = 1$ and appended to the edge list. After that, a dummy start operation 0 is to be connected with operations that have no predecessors. Such operations can be identified by the column sums of the precedence matrix that equal 0, i.e. no edges point at these operations. Dummy end operation $N + 1$ is connected to all operations that have no outgoing edges, which is determined by the corresponding row sums that equal 0. The function returns the `precedence_graph` object containing the set of directed precedence relations.

The wrapper function `solve()` is called after process data generation and import. Maximum load capacities of machines λ and the total number of available workers c are initialised in the body of the `solve()` function. The number of on-site cranes and manipulators is determined automatically by getting the number of elements in the λ array. A Gurobi `Model` object is created to store variables, constraints and execute various data processing operations. The planning horizon \mathcal{T} is a Python `range` object that contains a sequence of integers between 0 and the sum of all operation durations, corresponding to the worst-case scenario of no parallel operations in the schedule. The set of operations \mathcal{O} is represented by integers from the interval $\{0, \dots, N + 1\}$. The cranes and manipulators are enumerated in the set \mathcal{M} . The precedence graph of the assembly job is represented by the set of directed edges \mathcal{S} .

Based on the binary programming formulation proposed by Pritsker [CITATION!], binary decision variables are used to assign operations their time slots. In contrast to Pritsker's formulation, binary decision variable $x_{ktm} = 1$ if operation k starts in the beginning of time period t on machine m . The end times of operations

¹Matrix indices in the imported DataFrame start with 0, but the first assembly operation has index 1 in the used notation

are integer variables e_k . The makespan of an assembly schedule is denoted by an integer variable C_{max} . The RCPSP input data and variables can be summarised as follows:

\mathcal{O}	set of operations \mathcal{O}_k with $k \in \{0, \dots, N+1\}$
pt_k	integer duration of operation k , where $pt_k \in \{0, \dots, 10\}$
ρ_k	weight of the component to be transported during operation k in kilograms
\mathcal{M}	set of machines (cranes and manipulators)
λ_m	load capacity of machine $m \in \mathcal{M}$ in kilograms
u_k	number of workers requested by operation k
c	total capacity of workers on-site
\mathcal{T}	planning horizon $\{0, \dots, \sum_{k=0}^{N+1} pt_k\}$: set of possible processing times for operations
\mathcal{S}	set of precedences between operations $(k, l) \in \mathcal{O} \times \mathcal{O}$
x_{ktm}	boolean, 1, if operation k starts at time t on machine m
e_k	integer end time of operation k
C_{max}	makespan

The MIP for scheduling with precedence graphs can then be formulated as follows:

$$\min \quad C_{max} = \max_k e_k \quad (5.14a)$$

$$\text{s.t.} \quad e_k = \sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} (t + pt_k) \cdot x_{ktm} \quad \forall k \in \mathcal{O} \quad (5.14b)$$

$$\sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ktm} = 1 \quad \forall k \in \mathcal{O} \quad (5.14c)$$

$$\sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} t(x_{ltm} - x_{ktm}) \geq pt_k \quad \forall (k, l) \in \mathcal{S} \quad (5.14d)$$

$$(\lambda_m - \rho_k) \cdot \sum_{t \in \mathcal{T}} x_{ktm} \geq 0 \quad \forall k \in \mathcal{O}, m \in \mathcal{M} \quad (5.14e)$$

$$\sum_{k \in \mathcal{O}} \sum_{m \in \mathcal{M}} \sum_{t'=\max(0, t-pt_k+1)}^{t+1} u_k x_{kt'm} \leq c \quad \forall t \in \mathcal{T} \quad (5.14f)$$

$$\sum_{k \in \mathcal{O}} \sum_{t'=\max(0, t-pt_k+1)}^{t+1} x_{kt'm} \leq 1 \quad \forall t \in \mathcal{T}, m \in \mathcal{M} \quad (5.14g)$$

The optimisation objective is to minimise the makespan (5.14a), which is defined as the latest completion time of assembly operations (5.14b). The objective is set in Gurobi by the command `model.setObjective(C_max, GRB.MINIMIZE)`. Constraints are added to the MIP Model using Gurobi function `addConstr()`. Condition 5.14c ensures that each operation is assigned only one starting time t on only one machine m in the planning horizon \mathcal{T} . The precedence constraint 5.14d implies that the earliest start time of a successor operation l is after the predecessor operation k is finished. Condition 5.14e restricts the choice of machines based on their load capacities. The resource constraints 5.14f and 5.14g ensure that the capacity of workers is not exceeded at any time and that each machine can only process one operation at a given time slot.

The optimal schedule is generated by calling the `model.optimize()` method, which generates a solution log in the console at runtime. Once the model is solved, the values of variables different from zero are printed out by `model.printAttr('X')` method, unless the specified model is infeasible or unbounded. Python matplotlib visualisation library [Cas20] is used to view the schedule in a Gantt chart for better readability of results, and the precedence graph is plotted with NetworkX graph visualisation library [HSS08].

5.2.2 Scheduling with AND/OR-derived process graphs

The procedure implemented in `and_or_mip.py` is an alternative way of assembly sequence planning and scheduling that makes use of the process graph. In contrast to the assembly tiers method, subassemblies are allowed to be built. In general, AND/OR-derived process graphs contain a much bigger number of possible operations than precedence graphs. Nevertheless, only a subset of alternative assembly operations needs to be chosen in order to create a valid assembly sequence. The involved assembly system model is generated analogously as in the previous section and includes the worker capacity constraint as well as the machine load capacity constraint. The process data are generated according to the method described in the introduction to the scheduling module.

Since the structure of process graphs is more complex than that of precedence graphs, the input data and variables are added or adjusted to represent the changes in logic:

\mathcal{O}	set of operations \mathcal{O}_k in the process graph
pt_k	integer duration of operation k , where $pt_k \in \{0, \dots, 10\}$
ρ_k	weight of the component to be transported during operation k in kilograms
\mathcal{M}	set of machines (cranes and manipulators)
λ_m	load capacity of machine $m \in \mathcal{M}$ in kilograms
u_k	number of workers requested by operation k
c	total capacity of workers on-site
\mathcal{T}	planning horizon $\{0, \dots, T = \sum_{k=0}^{N+1} pt_k\}$: set of possible processing times for operations
P_j	set of task pairs (k, l) with direct precedence in the process graph
R_j	set of decision nodes for product j
S_j	set of splitting nodes for product j
$R_{jr}^{in}, R_{jr}^{out}$	set of ingoing and outgoing tasks of decision node $r \in R_j$
$S_{js}^{in}, S_{js}^{out}$	set of ingoing and outgoing tasks of splitting node $s \in S_j$
x_{ktm}	boolean, 1, if operation k starts at time t on machine m
e_k	integer end time of operation k
C_{max}	makespan

The full MIP for scheduling with AND/OR-derived process graphs is shown below:

$$\min \quad C_{max} = \max_k e_k \quad (5.15a)$$

$$\text{s.t.} \quad e_k = \sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} (t + pt_k) \cdot x_{ktm} \quad \forall k \in \mathcal{O} \quad (5.15b)$$

$$\sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ktm} \leq 1 \quad \forall k \in \mathcal{O} \quad (5.15c)$$

$$\sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} t(x_{ltm} - x_{ktm}) \geq pt_k \cdot \sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ktm} - T \left(1 - \sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ltm} \right) \quad \forall (k, l) \in P_j \quad (5.15d)$$

$$\sum_{k \in R_{jr}^{in}} \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} x_{ktm} = 1 \quad \forall r \in R_j, r = 1 \quad (5.15e)$$

$$\sum_{k \in R_{jr}^{in}} \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} x_{ktm} = \sum_{l \in R_{jr}^{out}} \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} x_{ltm} \quad \forall r \in R_j, r > 1 \quad (5.15f)$$

$$\sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} x_{ktm} = \sum_{t \in \mathcal{T}} \sum_{m \in \mathcal{M}} x_{ltm} \quad \forall s \in S_j, k \in S_{js}^{out}, l \in S_{js}^{in} \quad (5.15g)$$

$$(\lambda_m - \rho_k) \cdot \sum_{t \in \mathcal{T}} x_{ktm} \geq 0 \quad \forall k \in \mathcal{O}, m \in \mathcal{M} \quad (5.15h)$$

$$\sum_{k \in \mathcal{O}} \sum_{m \in \mathcal{M}} \sum_{t'=\max(0, t-pt_k+1)}^{t+1} u_k x_{kt'm} \leq c \quad \forall t \in \mathcal{T} \quad (5.15i)$$

$$\sum_{k \in \mathcal{O}} \sum_{t'=\max(0, t-pt_k+1)}^{t+1} x_{kt'm} \leq 1 \quad \forall t \in \mathcal{T}, m \in \mathcal{M} \quad (5.15j)$$

The optimisation objective of this MIP model is the same as that of precedence graph-based assembly scheduling, to minimise the makespan, i.e. the latest completion time of operations. Constraint 5.15c is a modification of the analogous constraint in precedence graph-based scheduling (5.14c) which accounts for the fact that not all operations $k \in \mathcal{O}$ need to be chosen for the schedule. The next condition, 5.15d, is responsible for restricting the earliest starting times of successor operations. This constraint is relaxed whenever the successor operation l is not chosen for the schedule ($\sum_{m \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{ltm} = 0$). The set of direct precedence relations in the process graph P_j contains all pairs of ingoing and outgoing operations of each node. Constraints 5.15e - 5.15g are modifications of constraints 7 - 9 from [Ehm19], which synchronise ingoing and outgoing flows at decision nodes and represent parallel operations at splitting nodes. Conditions 5.15h and 5.15i are deployed to check load capacities of cranes and on-site worker capacities. The last constraint, 5.15j, ensures that machines are never used by more than one operation at the same time.

A Assembly tiers implementation

A.1 Table of variables

Symbol	Code variable	Description
Π	cRelevantProducts	Entire product without fasteners
N	cRelevantProducts.Count	Number of relevant parts
p_i	cRelevantProducts.Item(int_i)	Part by index i
B	cBaseProducts	Set of base components
\mathbf{X}^*	aInitPos	Matrix of initial part positions
$\Delta \mathbf{X}$	aRemovalDistances	Removal distance matrix
$\tilde{\mathbf{X}}$	aPartBBGlob	Bounding box matrix
i	int_i	Part index
j	int_j	Counter of tested disassembly directions
I	intI	Counter of parts left in assembly
J	intJ	Number of disassembly directions to be tested
D	d1	Set of movement directions
s	intStep	Movement step distance
ε	dCollSens	Collision sensitivity
$G_{1/2}$	group1/2	Groups of parts for clash analysis
Θ	bDeactivated	Set of deactivated parts
M	bMoveable	Set of moveable parts
Φ	bDisassembled	Set of parts with at least one free disassembly direction
L	connectivityCheckNodeIndices	Set of parts examined in connectivity checks
\mathbf{S}	precedenceMatrix	Precedence matrix
l	intTier	Tier counter
\mathbf{t}	aTiers	Tiers vector
\mathbf{Y}	disassDir	Disassembly directions matrix

Table A.1: Mathematical symbols and code variables in AssemblyTiers algorithm