KnitKit: A flexible system for machine knitting of customizable textiles

Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, Sai-Kit Yeung

KnitKit: A flexible system for machine knitting of customizable textiles

GEORGES NADER, Panasonic RnD Center Singapore Singapore University of Technology and Design,
YU HAN QUEK, Singapore University of Technology and Design
PEI ZHI CHIA, Singapore University of Technology and Design
OLIVER WEEGER, Technical University of Darmstadt, Singapore University of Technology and Design
SAI-KIT YEUNG, Hong Kong University of Science and Technology

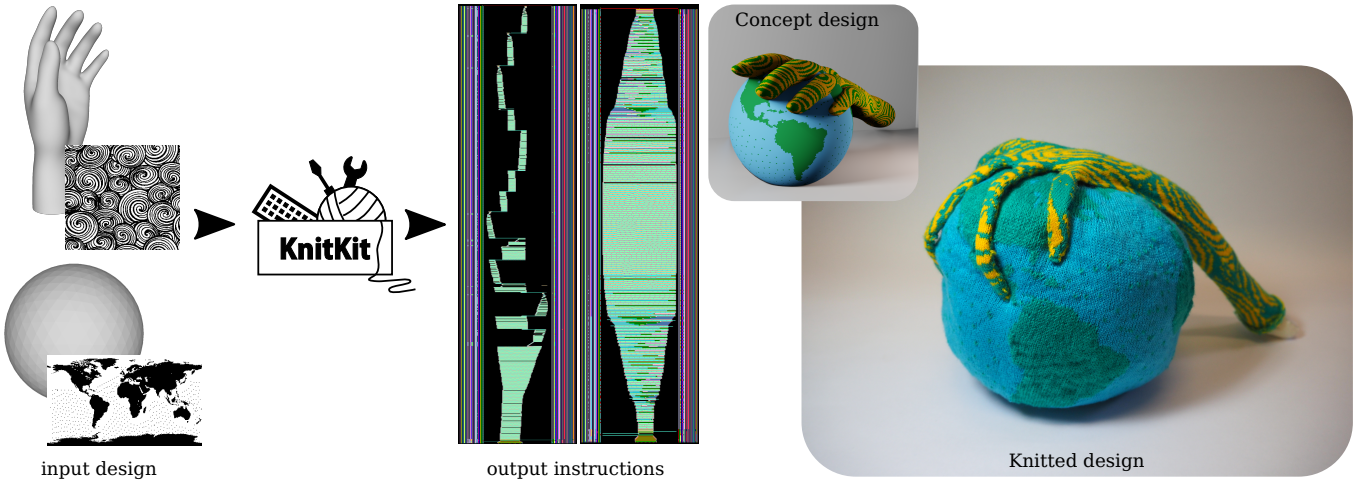

Fig. 1. The *KnitKit* system decouples the high-level design aspect of producing knitted textiles from the complexities and low-level specificity of knitting machines by generating machine knitting instructions from an input 3D geometry and a texture. This enables high-level design of knitting properties, i.e., geometry, yarn types and stitch patterns. Here, we CNC knitted a glove and a globe.

In this work, we introduce *KnitKit*, a flexible and customizable system for the computational design and production of functional, multi-material, and three-dimensional knitted textiles. Our system greatly simplifies the knitting of 3D objects with complex, varying patterns that use multiple yarns and stitch patterns by separating the high-level design specification in terms of geometry, stitch patterns, materials or colors from the low-level, machine-specific knitting instruction generation. Starting from a triangular 3D mesh and a 2D texture that specifies knitting patterns on top of the geometry, our system generates the required machine instructions in three major steps. First, the input is processed and the *KnitNet* data structure is generated. This graph structure serves as an abstract interface between the high-level geometric and knitting configuration and the low-level, machine-specific knitting instructions. Second, a graph rewriting procedure is applied on the *KnitNet* that produces a sequence of abstract machine actions. Finally, the low-level machine instructions are generated by adapting those abstract actions to a specific machine context. We showcase the potential of this computational approach by designing and fabricating a variety of objects with complex geometries, multiple yarns, and multiple stitch patterns.

CCS Concepts: • **Applied computing** → Computer-aided manufacturing.

Additional Key Words and Phrases: Machine knitting, functional textiles

ACM Reference Format:
Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, and Sai-Kit Yeung. 2021. KnitKit: A flexible system for machine knitting of customizable textiles. *ACM Trans. Graph.* 40, 4, Article 64 (August 2021), 16 pages. https://doi.org/10.1145/3450626.3459790

Authors' addresses: Georges Nader, Panasonic RnD Center Singapore, Singapore University of Technology and Design,, georges_nader@sg.panasonic.com; Yu Han Quek, Singapore University of Technology and Design; Pei Zhi Chia, Singapore University of Technology and Design, peizhi_chia@sutd.edu.sg; Oliver Weeger, Technical University of Darmstadt,, Singapore University of Technology and Design, weeger@cps.tu-darmstadt.de; Sai-Kit Yeung, Hong Kong University of Science and Technology, saikit@ust.hk.

1 INTRODUCTION

Recent advances in fiber and yarn technologies, as well as digital and three-dimensional (3D) textile manufacturing, have created increasing potential for technical textiles and performance fabrics not only in fashion and apparel industries, but also in engineering applications, e.g., in the aerospace, automotive, architecture, bio-medical, and defence sectors [Chen 2015; Leong et al. 2000]. Functional properties such as electric, magnetic and thermal conductivity, light sensitivity, mechanical stiffness, or impact resistance can now be embedded in a fabric by using advanced yarn materials and complex knitting designs that involve varying stitch patterns.

Furthermore, computer numerical control (CNC) knitting machines are now capable of holistically fabricating 3D textiles with a

wide range of shapes, textures, and materials [Underwood 2009]. In particular, CNC knitting allows users to combine 3D shape-ability with the flexibility to locally embed yarn materials and vary stitch types, parameters, and patterns, which provides unprecedented design freedom for the development and customization of functional textiles [Abel et al. 2012]. However, it is very challenging and time consuming to realize these high-level knitting designs, since they have to be translated to a low-level interface that operates at the complexity of individual stitches and machine instructions. Thus, users, i.e., textile designers, must develop a detailed understanding of the machine knitting process in order to realize their designs. In addition, the difficulty of *programming* the machine, i.e., building the low-level machine instructions, tends to increase disproportionately with the complexity of the geometry and the functional specifications. This can make the fabrication process very laborious and thus limits the practical knitting output of these machines to relatively simple objects.

In this work, we aim to make the design and fabrication of functional textiles more practical and accessible even to non-expert users. To do so we:

- Propose the *KnitKit* system – a flexible and customizable pipeline for machine knitting. It proceeds by transforming a high-level geometry and knitting configuration into abstract knitting actions that are later translated to low-level instructions.
- Define the *KnitNet* data structure – a row-based directed graph representation that describes the structure of knittable objects. It allows the easy representation of objects made with varying yarns and stitch patterns. This structure serves as an abstract interface between the high-level knit design and the low-level knitting instructions.
- Present a novel scheduling algorithm that generates the low-level knitting instructions from the *KnitNet* by (1) generating a set of abstract actions, then (2) translating the actions into concrete, machine-specific low-level instructions.

2 BACKGROUND

Knitted fabrics consist of one or more continuous yarns that loop through existing loops to create stitches. These stitches form an inherent row-column structure, where the rows, i.e., the *courses*, emerge from yarn-wise connections and the columns, i.e., the *wales*, emerge from loop-wise connections. Various types of stitches can be realized through different looping methodologies and patterns can be created by combining different stitch types over several courses and wales. Furthermore, due to the different geometric dimensions of stitch types and the possibility to merge and split wales, knitted fabrics are not necessarily locally flat. This enables the realisation of complex 3D shapes with a single, continuously knitted piece of fabric [Underwood 2009].

2.1 V-bed knitting machines

V-bed weft knitting machines are the most versatile type of knitting machines, since they enable the fabrication of seamless garments with multiple yarns and varying patterns [Power 2015]. They consist of two main elements: (1) two beds of needles facing each other forming an inverted V-shape, and (2) a system of *yarn carriers* that move parallel to the needle beds, feeding yarns to the needles. Each needle is capable of holding one or more *loops* of yarn at a time and can be independently actuated to perform a small set of primitive operations. In general, the knitting process consists of forming new loops by selectively actuating specific needles on the bed to pull new yarn through the existing loops. The existing loops then descend through the gap between the beds. The process is repeated and the aggregating rows of loops form a piece of fabric. The needles can also perform *tuck*, *drop*, or *transfer* operations, and the beds can be *racked* sideways relative to each other for a variety of results. Thus, creating a knitted object with these machines boils down to executing an extensive series of operations, each of which would either move the yarn across the needle bed or actuate specified needles to perform any of the aforementioned operations. These operations are in practice also coupled by carriage and cam mechanisms. Additionally, there are various secondary factors that can influence the knitting process, such as pull-down and tension control settings. For a more detailed and technical description of machine knitting, we refer the reader to [Spencer 2001].

Commercial CNC V-bed knitting machines such as the ones of Shima Seiki or Stoll [Seiki; Stoll] are usually accompanied by a computer-aided manufacturing software that provides a visual interface for *programming* the machine at a low-level of abstraction. For Shima Seiki machines, this is the KnitPaint data format. These interfaces allow the user to specify knitting instructions in terms of arrangements of stitch types or patterns. Thus, they require highly trained and skilled experts to painstakingly translate complex high-level textile designs into machine-readable instructions. This puts a practical limit on the capabilities of these machines, as realizing complex textiles becomes a time-consuming and difficult task.

2.2 Related work

A lot of research has been devoted to the modeling and fabrication of cut and sew garments [Li 2018; Wolff and Sorkine-Hornung 2019]. Most of the work tackling knitted objects has mainly focused on the visualization [Aliaga et al. 2017; Meißner and Eberhardt 1998; Wu and Yuksel 2017] and simulation [Cirio et al. 2016; Leaf et al. 2018; Yuksel et al. 2012] aspects. In general, they are not suited for fabrication, since the generated knit patterns are not guaranteed to be knittable. For instance, the *stitch mesh* framework [Wu et al. 2018] provides a powerful interface for modeling knitted fabrics. While it ensures that the structure is topologically valid, i.e., does not unravel, it does not guarantee its machine knittability. This is mainly because the framework neglects inconsistencies in the underlying knitting directions.

In the past few years, researchers have recognized the lack of high-level design tools for the fabrication of knitted textiles. McCann *et al.* [2016] presented a CAD-like tool where users manipulate simple primitives, such as tubes and sheets, to design knitted objects. Machine compatible instructions are then generated by an algorithm that automatically schedules bed layouts. Kaspar *et al.* [2019] expanded on this CAD-like tool idea. In addition to composing simple primitives, their system includes a domain-specific language for patterning and a novel pattern representation which allows the user to knit garments with various stitch patterns. While these

tools are easier to use than commercial software, they still require a designer to manually configure the available primitives. Others have tried to generate knitting instructions directly from a triangular mesh. Igarashi *et al.* [2008] presented a system for generating hand-knitting instructions, which converts a manually segmented model into separate regions of parallel winding strips which are then sampled at constant intervals. However, this method cannot be applied to machine knitting primarily due to scheduling constraints. Wu *et al.* [2019] extended the *stitch mesh* framework [Wu et al. 2018; Yuksel et al. 2012] to produce knittable structures. They also presented a scheduling algorithm that generates step-by-step knitting instructions for a given model. These techniques, however, rely on hand-knitting capabilities and do not take into consideration the constraints imposed by the machine. Popescu *et al.* [2018] described a system for generating machine-knittable 3D objects from quad meshes. However, their system still requires users to manually segment the model. Narayanan *et al.* [2018] introduced a method for automatically generating machine knitting instructions from an input mesh. Their approach relies on a user-defined time function that encodes the desired knitting direction. This time function guides a sequential quad-dominant remeshing algorithm that produces a geometric mesh structure compatible with machine knitting. This mesh is then passed to a tracing algorithm that generates the corresponding machine instructions. Narayanan *et al.* [2019] extended this work by embedding certain classes of primitive knitting instructions into the mesh faces. This allowed them to design a user-friendly system that is capable of visually creating knitting programs in a 3D design interface. More recently, Wu *et al.* [2021] adopted a different approach to handle the knitting of complex 3D structures. They introduced an automatic cutting algorithm that turns any 3D surface into a knittable topological disc. The resulting knittable surface is then fed to a machine instruction generation module based on the *stitch mesh* framework [Wu et al. 2018; Yuksel et al. 2012].

While these methods certainly make the use of knitting machines more accessible and allow for very fine-grained control of the output, they do not scale well to larger and more complex input patterns due to the need for manual stitch-level editing. They also do not lend themselves well to parameterization of a high-level specification, as the working domain is limited to primitives on the order of single stitches. For example, to knit an object with a ribbed pattern, one has to manually specify alternating wales of knit and purl stitches, and deal with various conditions caused by geometric irregularities. This means that subsequent scaling or tweaking of the geometry would require the user to re-draw the desired pattern from scratch. Thus, producing complex patterns that involve the interaction of stitches with their neighbors in both wale and course directions is impractical in general.

Despite this progress, handling 3D objects constituted of complex stitch patterns and multiple yarns remains an open issue, since they also require the complex management of multiple yarn carriers. Taking these aspects into consideration is important for the realization of functional knitted objects. More specifically, we present a novel row-based directed graph representation of knittable objects that acts as an abstract interface between the low-level machine instructions and the high-level design. This allows us to develop a highly customizable system and a scheduling algorithm that are capable of handling complex knit structures and multi-yarn objects.

3 OVERVIEW

The design of our machine knitting system is guided by the following objectives:

(1) **Geometric independence:** The input geometry should remain independent from various knitting parameters such as number of yarns and stitch patterns.
(2) **Machine independence:** While the generated instructions are specific to a particular machine, the algorithm used to generate them should be flexible enough to support different hardware.
(3) **Customizability:** Users should be able to customize the output machine instructions according to their needs. This allows the framework to support an extendable library of knitting styles and stitch patterns.

A general overview of the *KnitKit* system, which implements these design objectives, is presented in Fig. 2. Our knitting system functions as follows: We start by passing the input design to a geometry analysis framework that builds the *KnitNet* data structure. This input consists of a triangular 3D mesh along with a vector field that represents the geometry to be knitted and the desired knitting direction, respectively. Furthermore, texture data can be attached to the input geometry in order to customize the knitting operation in terms of yarn materials and stitch patterns.

The geometry analysis framework begins with parameterization followed by a remeshing operation that transforms the input geometry into a quad-dominant mesh. The edges of the resulting mesh are then assigned to an orientation consistent with the input vector field and grouped into two categories: wales and courses. Additionally, the knitting configuration is extracted from the attached textures and stored into the vertices. In the case of a 3D input geometry, we segment the geometry into two parts, each corresponding to one needle bed of the knitting machine. Finally, we perform a clustering operation that groups adjacent vertices of the quad-dominant mesh into rows. With this information, the *KnitNet* data structure is generated. It encodes the topology and target knitting structure and contains all the necessary information to generate the corresponding machine instructions.

Second, we process the *KnitNet* structure and **generate the low-level machine instructions**. This step starts by generating a sequence of actions from the *KnitNet*. These actions conceptually describe the operations required for the machine to knit the input geometry. This is achieved with a graph rewriting approach that builds this sequence in an iterative manner using a set of rules. The resulting sequence of actions is then transformed into a set of low-level machine operations using a library of routines. These routines adapt the high-level actions according to the machine physical state and configuration. Finally, these operations are transcribed into the native data format of the specific machine being used. Both the graph transformation rules and the routine library can be modified and extended by the user. This makes our knitting system highly flexible and customizable. However, customizing both the rules and the routine library requires knowledge in machine knitting thus is considered to be a task for machine specialists. It is important to

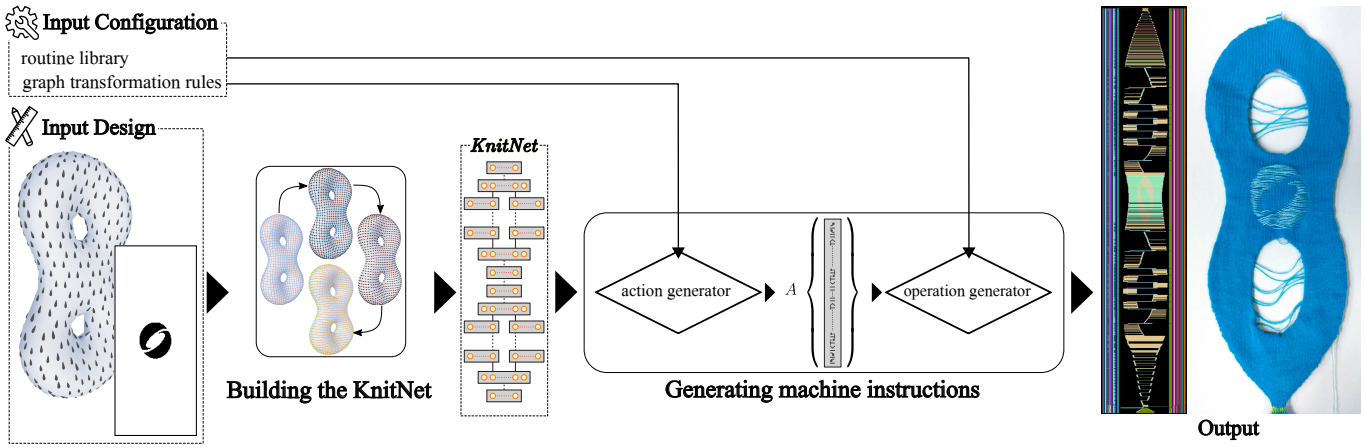

Fig. 2. Overview of the *KnitKit* system. First, an input design given by a textured 3D mesh is converted into a topological *KnitNet* data structure. Using a user-customizable library of graph transformation rules and routines, this data structure is processed via an intermediate abstract action stage into machine instructions, which can be loaded into a knitting machine (visualized here in KnitPaint) to fabricate a customized knitted textile (Size: 120 x 180 mm).

note that the graph rules and the routine library are independent from any specific input design. This means that they can be reused to generate machine instructions for any arbitrary input design represented by a *KnitNet* supported by the given graph rules and routine library.

Splitting the *KnitKit* into two independent stages makes the system more accessible and achieves our objectives of developing a geometry and machine independent, customizable machine knitting system. Non-expert users can focus on the design aspect by configuring the knitting process from a high-level conceptual point of view, i.e., modeling the input geometry using standard 3D modeling software or configuring the knitting design in term of yarns and stitch types using 2D textures. On the other hand, machine specialists can concentrate on building a robust set of graph rewriting rules and the routine library used to generate the knitting instructions to support a large class of objects and patterns.

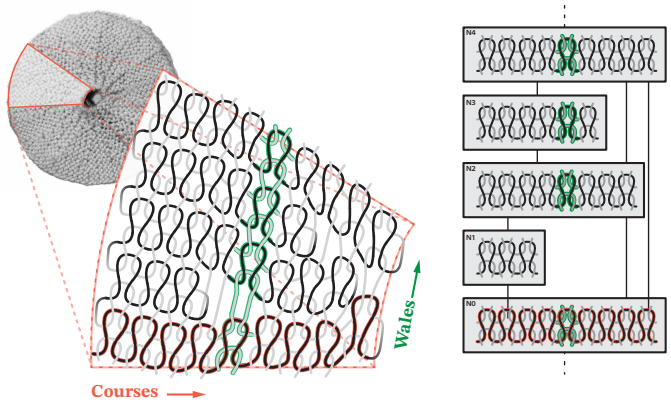

Fig. 3. The row-wise topological structure of a knitted object is captured by the *KnitNet* graph. Each course of a knitted object is represented by a node that contains a list of stitches, the KN Node. The course-wise connections are represented as directed edges connecting one node to another, the KN Edges.

4 THE *KNITNET* DATA STRUCTURE

The *KnitNet* is a directed acyclic graph that captures the topological structure of a knitted object, see Fig. 3. The nodes of the *KnitNet* roughly correspond to the courses of the knitted object and the edges describe the wale-wise connections between courses. We define the *KnitNet* with the following elements:

KN Node. The nodes of the *KnitNet* each contain a set of ordered vertices that are connected in a course-wise manner. It is described as follows:

- `id`: a unique identifier.
- `template`: a user-defined field that represents a certain knitting pattern. It can include an arbitrary number of physical yarns of different colors or materials. For example a Fair Isle colorwork pattern is defined by a `template` containing two (or more) yarns of different colors.
- `vertices`: an ordered list, where each vertex generally represents a stitch of the knit structure.

Additionally, our framework allows for local configuration of the `template`. To that end, each `vertex` is represented by:

- `id`: a unique identifier.
- `type`: a user-defined field used to configure the corresponding template. For instance, it can be a single key attribute that maps to a certain stitch pattern (Single Jersey, Rib, Full Cardigan, etc.), or in the case of a Fair Isle template, it can be a flag that designates which yarn is carried.

KN Edge. The edges between two nodes of the *KnitNet* correspond to the course-wise connections between adjacent KN Nodes. It is possible to have multiple connections from and to the same KN Node. However, they cannot include overlapping vertices. It is represented as follows:

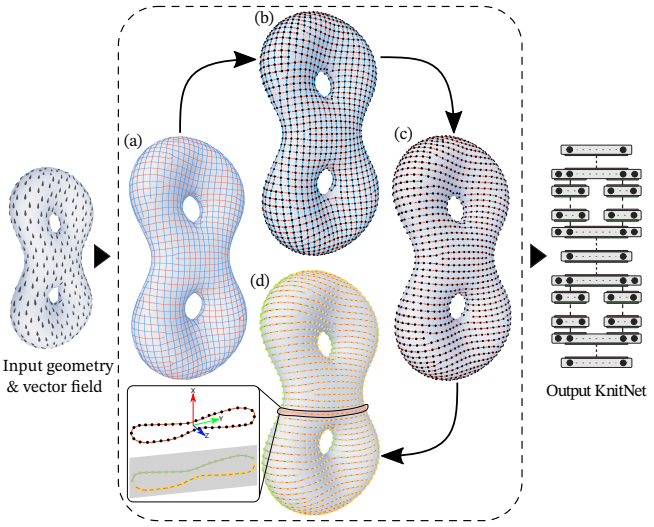

Fig. 4. Illustration of the various steps of building the *KnitNet* from an input 3D mesh and vector field, here for a double torus: (a) 2D stripe parameterization computed from the input geometry and vector field. (b) Quad dominant mesh M_Q with its edges classified as wale (blue) and course (red). (c) Vertex clusters obtained by tracing the vertices of M_Q along the course edges. (d) In the case of a 3D geometry, a segmentation step is performed that attributes each vertex of the quad-dominant mesh M_Q to a needle bed.

- `source/destination`: A reference for the `id` of the incoming and outgoing `KN Node`.
- `interval`: indicates the starting and ending vertex `id` of a connection.

5 BUILDING THE *KNITNET*

The process of building the *KnitNet* begins with a parameterization operation that produces orthogonal stripes with equal spacing. This is followed by a remeshing step that produces an adequate quad-dominant mesh. The edges of this mesh are consistently oriented in accordance with the input knitting direction and are classified into two groups: course-edges and wale-edges. In case the knitted geometry requires multiple needle beds, e.g., 3D geometry, a segmentation operation is performed that divides the mesh into two and assigns each half to the corresponding bed. Finally, the vertices of the remeshed model are grouped and ordered into consecutive connected nodes to obtain the *KnitNet* structure. This process is illustrated in Fig. 4 and described in detail below.

5.1 Directional quad-dominant remeshing

At its core, a knitted structure can be modeled by a non-conforming quad-dominant mesh $M_Q = (S, E)$, where the vertices S and edges E correspond to a set of stitches and their respective connections. In this case, T-junctions represent *short rows* and *internal increases* or *decreases*, which are essential for shaping, see Fig. 5. In order to model a knitted structure, the resulting quad-dominant mesh must meet the following specifications:

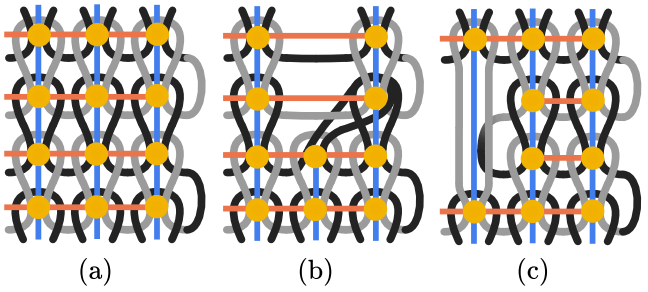

Fig. 5. The structure of a knitted object is represented by a quad-dominant mesh (nodes shown in yellow, course-wise edges in red and wale-wise edges in blue). (a) Regular quad mesh, representing a uniform Single Jersey knit. (b) Horizontal T-junction, representing internal shaping. (c) Vertical T-junction, representing a short row.

- It should be possible to classify the mesh edges E into two groups: *course* edges and *wale* edges.
- The edges of the mesh must be attributed a consistent orientation.
- The distance between two adjacent vertices must reflect the uniformity and geometric proportions of the knit structure.

The first requirement ensures that the stitches can be grouped into rows, while the second guarantees that the connections between the rows will result in a directed acyclic graph, and the third preserves the desired physical dimensions of the knitted object.

Given a unit tangent vector field $\vec{\mathbf{v}}_w$ defined at each vertex of the input triangular mesh, we start by computing $\vec{\mathbf{v}}_c$ as the rotation of $\vec{\mathbf{v}}_w$ by $\pi/2$ in the respective local tangent plane. From this cross-field $(\vec{\mathbf{v}}_w; \vec{\mathbf{v}}_c)$ we compute an orthogonal, equally spaced stripe pattern using Knoppel *et al.*'s [2015] method. This orthogonal stripe pattern describes the knitted object's *wales* and *courses*. The spacing d between two consecutive stripes is controlled by setting the stripe frequency to $\omega = 2\pi/d$. We set the spacing between course stripes, generated from $\vec{\mathbf{v}}_c$, and wale stripes, generated from $\vec{\mathbf{v}}_w$, to the horizontal and vertical physical stitch dimensions, respectively. It is also possible to locally modify the spacing between consecutive stripes, which allows us to account for variable stitch sizes when dealing with an input containing multiple stitch patterns.

Having computed the stripe pattern, we now generate the vertices S of M_Q by computing the intersection between two orthogonal stripes. This is efficiently performed by evaluating those intersections within each triangle of the input mesh. At this stage, we can easily augment the vertices with any additional information given by the input texture maps, such as yarn information or stitch patterns. In practice, the texture resolution is rarely equivalent to the resolution of M_Q. In order to deal with this mismatch, in our current implementation we compute the Voronoi cell for each vertex of M_Q and project it onto the input triangular mesh. We then assign to each vertex the information corresponding to the most common pixel value within the projected Voronoi cell.

Finally, similar to [Dong et al. 2005], we build an ordered list of the intersection points by tracing along each stripe on the input mesh, in the direction of its corresponding vector field. This connects the vertices and creates the edges of M_Q. Each edge is therefore classified as a wale or course by verifying whether it connects two

Algorithm 1: Building the `KN nodes`

input : $M_Q = (S, E)$
output: N, a list of `KN Nodes`

initialize a *visited* flag array to *False*
for *all stitches* s_i *in* S **do**
  if s_i ***not*** *visited* **then**
    $s_c \leftarrow$ go to leftmost stitch of s_i in the course direction
    $T_c \leftarrow$ get template of s_c
    $n \leftarrow$ createNode
    while s_c *is valid* **do**
      $T \leftarrow$ get template of s_c **if** T *and* T_c *equal* **then**
        add s_c stitch to node n
      else
        add node n to list N
        $n \leftarrow$ createNode
        $T_c \leftarrow T$
      end
      flag s_c as visited.
      $s_c \leftarrow$ adjacent stitch of s_c in the course direction
    end
    add node n to list N
  end
end
return N

Algorithm 2: Building the `KN Edges`

input : $M_Q = (S, E)$,
  N, the list of `KN Node` extracted from M_Q
output: C, list of connections between nodes of the *KnitNet*

for *all nodes* n_i *in* N **do**
  $s_0 \leftarrow$ first stitch of node n_i
  $s_0^w \leftarrow$ adjacent stitch of s_0 in the wale direction
  $n_{s_0^w} \leftarrow$ node of s_0^w
  for *all stitches* s_k *in node* n_i **do**
    $s_k^w \leftarrow$ adjacent stitch of s_k in the wale direction
    $n_{s_k^w} \leftarrow$ node of s_k^w
    if $n_{s_0^w}$ *and* $n_{s_k^w}$ *are different* **then**
      add connection from n_i to $n_{s_0^w}$ in the list C
      $n_{s_0^w} \leftarrow n_{s_k^w}$
    end
  end
end
return C

vertices along the wale or course stripe, respectively. Moreover, it is attributed an orientation that matches with the corresponding input vector field. It is important to note that Knoppel *et al.* [2015]'s stripe parameterization methods can in some cases lead to helices, which would break the orientation consistency of the edges of M_Q. To resolve this issue, we apply the grid-preserving operators of Bommes *et al.* [2011] at the end of our remeshing step.

In the case where the input geometry is three-dimensional, each vertex of M_Q must be assigned to either the front or back bed of a V-bed knitting machine. Concretely, this means that the vertices in each row of M_Q should be evenly divided into two groups, such that the boundary between them is continuous across adjacent rows. This is equivalent to computing a surface that cuts through the input geometry along the knitting direction, dividing it into half. In practice, this can be achieved by computing a series of consistently oriented planes for every row of M_Q and then assigning each vertex to the front or back bed based on its position with respect to that plane. The vertices in front and behind the plane are thus assigned to the front and back beds, respectively, see Fig. 4d.

5.2 Constructing the *KnitNet*

Once we have constructed the quad-dominant mesh M_Q and assigned the vertices to their needle beds, generating the *KnitNet* data structure is quite straightforward. First, we cluster the vertices into distinct rows in order to build the `KN Nodes`, see Algorithm 1. We start from an arbitrary vertex on the mesh. Then, we navigate in the course direction towards the leftmost boundary vertex. At this point, we create a new node and navigate the mesh along the course direction, while adding each vertex we encounter to that node. In cases where multiple templates have been assigned to the vertices of a row, we create a new node every time the current vertex has a different template attributed to it than the previous one. We stop when we reach the rightmost boundary. We repeat this process until all the vertices of M_Q have been visited. Second, we compute the `KN Edges`, i.e., the connection between the nodes, see Algorithm 2. Starting from an arbitrary node, we traverse its assigned stitches in an ordered manner. For each vertex we look up the node corresponding to its adjacent vertex in the course direction. We create a new edge every time a new adjacent node is referenced. We repeat this process for all nodes in the *KnitNet*.

6 GENERATING MACHINE INSTRUCTIONS

The major challenge of automating the generation of machine knitting instructions is dealing with the highly interlinked nature of low-level operations. To illustrate this, we consider the task of knitting a single course over a certain *span* of needles using some *stitch pattern*. The sequence of low-level machine operations needed to realize this task depends on:

(1) The *stitch pattern*: Producing a course with Single Jersey (SJ) stitches on the front bed requires performing successive `front-knit` operations. Producing a Double Jersey (DJ) course would require performing two passes over the same needle span, first with the cycled sequence {`front-knit`, `back-knit`} and then with {`back-knit`, `front-knit`}.
(2) The poses of the carriage and yarn carriers: The knitting mechanism most commonly involves a laterally moving carriage which simultaneously feeds the yarn via carriers and actuates needles. The operations needed to knit a course depend on whether the carriage and yarn carriers are on the left or right of the affected needles. In some cases, *carriage return* and *kick-back* operations are needed to ensure carriers do not interfere.
(3) The configuration of loops on the needle bed: New loops on unoccupied regions of the needle bed may require additional

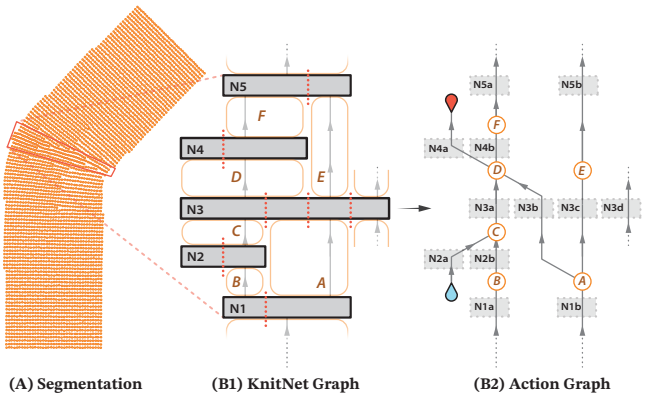

Fig. 6. The action graph G (shown in C) is generated from an input *KnitNet* (shown in B). Each node of G corresponds to an edge in the *KnitNet* and two nodes of G are connected if the corresponding edges in the *KnitNet* share a nonempty span of stitches in their mutual node. Note that there is no edge between nodes C and E in G, even though edges C and E in the *KnitNet* share a common node $N3$, because they do not share an adjunct partition of stitches in $N3$.

Algorithm 3: Generating the sequence of actions

input : K, the *KnitNet* data structure, P, a set of rules
output: A, a sequence of actions

$G \leftarrow$ build action graph from K
while $G \ncong \overline{G}$ **do**
  matchFound $\leftarrow$ false
  for $p_i \sim (L_i, R_i) \in P$ **do**
    $g \leftarrow$ find a match for subgraph L_i in G
    if $g \neq \emptyset$ **then**
      substitute g with R_i in G
      matchFound $\leftarrow$ true
      break
    end
  end
  if *matchFound* = *false* **then**
    error: unable to collapse graph
end
$A \leftarrow$ get action sequence in $\overline{G}$
return A

set-up operations. Existing loops which are held for an extended period may require operations to release tension.

In order to handle this complexity, we divide the machine instruction generation process into two stages. First, we use the input *KnitNet* to build a sequence of abstract knitting *actions*, which are generally needed to realize the object at hand. These *actions* correspond to conceptual tasks to be carried out by the machine and are not concerned with the dependencies mentioned above. Second, we translate those *actions* into machine-specific low-level operations that map directly into the machine's native data format.

6.1 Generating the sequence of actions

A knit object is fundamentally a series of connected rows of stitches that is fabricated in a sequential manner. We capture this row configuration by our *KnitNet* data structure. Conceptually, the knitting process can be seen as the act of realizing the *KnitNet* one `KN Node` at a time. In each step of the process, a knitting action is invoked in order to create a `KN Node`, i.e., a row of stitches.

In order to represent the sequential nature of the knitting process, we construct an *action graph* G that describes the steps required to knit the input *KnitNet*. Each node of G corresponds to a `KN Edge` in the *KnitNet*. Two nodes of G are connected if the corresponding `KN Edges` share a nonempty span of stitches in their mutual `KN Node`. In other words, the edges of G correspond to mutually disjoint partitions of the `KN Nodes`, see Fig. 6 for illustration. In this context, the transition from one node of G to another via an edge represents an *action* that would knit the partition of the `KN Node` embedded in the edge. When the partitions of the `KN Nodes` are not connected to any other, their corresponding edges in G would be connected to specially tagged *start* or *end* nodes. These specially tagged *start* or *end* nodes indicate the boundaries of the knit object and their incoming/outgoing edges represent additional *cast-on* or *bind-off* actions.

In order to generate the sequence of *actions* described by G, we proceed with a double pushout algebraic graph rewriting approach [Corradini et al. 1997]. Each rewriting operation corresponds to combining multiple knitting steps according to a suitable rule. In more technical terms, each rule is defined as $P_i = \{L_i \rightarrow R_i\}$, where L_i and R_i refer to a template subgraph and a replacement subgraph respectively. Rewriting the graph according to a rule P_i consists of searching for an occurrence of the subgraph L_i in G that satisfies certain constraints and replacing it with an instance of R_i. The goal of this approach is to transform G to its canonical form $\overline{G}$, consisting only of two nodes connected by a single edge, according to the given set of graph transformation rules. It represents the macro-transition between the start and the end of the knitting process using the sequence of actions embedded in its edge. This rewriting approach is detailed in Algorithm 3.

Figure 7 illustrates this process for the double torus example. In this case, we have the rule library $P = \{P_{1a}, P_{1b}, P_{2a}, P_{2b}, P_3, P_4\}$, with rules being ordered according to their priority. Using this particular set of rules, the algorithm is able to handle the class of torus-like topologies without short rows. P_1 and P_2 handle geometric increases and decreases, respectively. P_3 is a simple concatenation rule that removes a node and chains the corresponding actions, and P_4 handles the scheduling of hole topologies. Generating the sequence of actions that would produce the double torus starts by generating the graph G from the respective input *KnitNet*. At each iteration, the algorithm identifies a subgraph of G that is isomorphic to a template graph L_i of a rule P_i. If multiple valid subgraphs are found, the one corresponding to the highest priority rule is selected. In our implementation, Ullman's algorithm [Ullmann 1976] is used for identifying matching subgraphs. Using the double-pushout rewriting algorithm, the subgraph L_i is substituted with its corresponding template R_i. By iterating this procedure, G is gradually reduced in size, while the embedded action sequences in its edges grow larger.

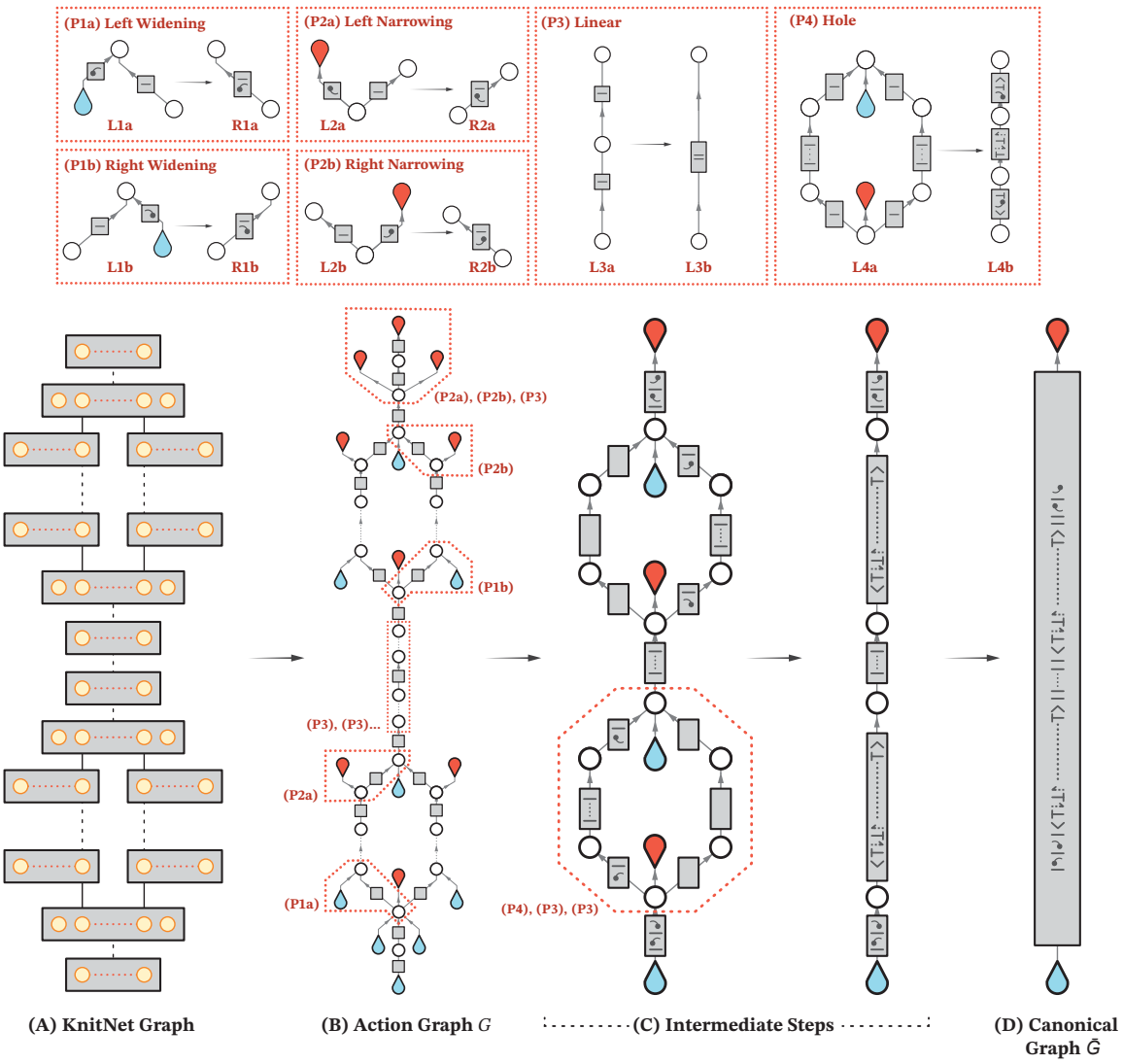

Fig. 7. The input *KnitNet* is converted into an action graph G, and iteratively transformed using a double pushout graph transformation approach, according to a user-defined library of rules P. At the end of the transformation, we obtain the canonical action graph $\overline{G}$, and extract the sequence of actions A needed to produce the knit structure from the single edge of $\overline{G}$.

Eventually, G consists of only two nodes connected by a single edge, the canonical form $\overline{G}$. The final sequence of actions can thus be extracted from the edge of this graph. If the algorithm fails to transform G into $\overline{G}$, this indicates that the provided rule library is incapable of handling the object's topology. In this case, an expert user can extend the rule library to allow the system to handle more complex topologies.

Building the set of graph transformation rules is not a trivial task. Defining the functions attached to the edges of the replacement graphs requires some level of expertise in machine knitting and can be used for fine-grained control over the scheduling (see Sect. 7.2). However, once this set is built, it can be reused to generate the sequence of actions for any input graph that conforms to certain geometric shapes and knitting templates. Allowing expert users to define custom rules at this level ensures that the system is highly extensible and can adapt to different requirements.

6.2 Generating the low-level instructions

Having generated a sequence of abstract actions A, which are still independent from the machine context but reflect the sequential nature of the knitting process, from the input *KnitNet* via G and $\overline{G}$, the goal now is to compile it into a series of low-level machine operations. The principal challenge here is handling the highly interlinked nature of machine knitting, which we overcome by keeping track of the physical *state* of the machine while translating the abstract actions into *operations*. Each action is associated with a set

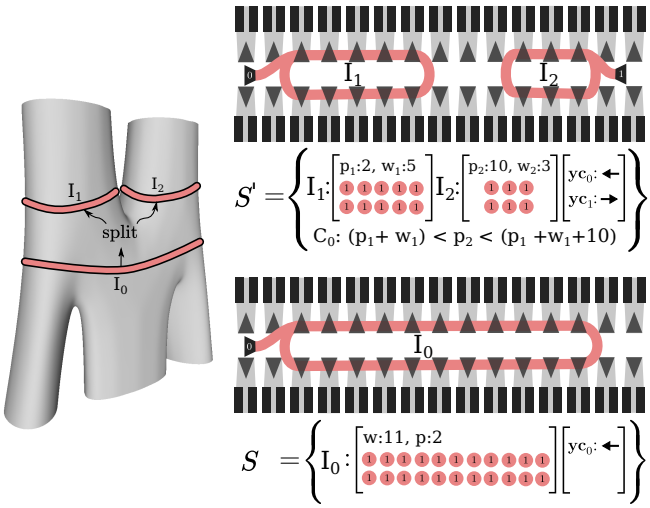

Fig. 8. A state S encodes the layout of carriers and loops on the needle bed at a certain point in the knitting process. An action executed in the context of the state S changes the layout of the yarn on the needle bed. For example, a `split` action would split one island into two and attribute a new yarn carrier to the newly created island.

of user-defined *routines* which correspond to different implementations of the abstract knitting task. These routines depend on the current state of the machine and modify it within each execution step.

In our implementation, the machine's *state* S is encoded as a set of i *islands* and a set of j *yarn carrier* poses:

$$S = \Big(\{I_0, \ \ldots, \ I_i\}; \{\mathrm{yc}_0, \ \ldots, \ \mathrm{yc}_j\}\Big)$$

Additionally, we include relative spatial constraints between the *islands*, which ensure that they are within a certain distance of each other and do not overlap. An *island* is a contiguous region of the needle bed occupied by knitted loops. In general, each *island* corresponds to a separate branch of the geometry, spans a certain number of stitches w, and can occupy either one or both sides of the needle bed. It is represented by an array of *slots*, each containing information about the loops occupying a physical needle, e.g., the `template` and `type` of the last stitch operation performed on the needle. We encode the location of an island by its leftmost reference point with respect to the start of the needle bed. For example, in Fig. 8, in the updated state S', there are two islands I_1 and I_2 positioned at needles 2 and 10, with widths 5 and 3 respectively. Here, the constraint C_0 specifies that I_1 must be located on the right of I_0, and separated by a maximum distance of 10 needles without overlapping. This ensures that subsequent processing steps do not violate these constraints, rendering the program unknittable. Each *yarn carrier*, is encoded by its absolute *position* on the needle bed, along with the *direction* of movement, i.e., leftwards or rightwards, of the most recent operation involving it. Yarn carriers can be temporarily disabled by sending them *out* of operation on either side of the needle bed.

In general, realizing a knitting *action* leads to a change of the machine's state, such as moving of yarn carriers and altering the

Algorithm 4: Transforming actions into operations

input : S_0, the initial state, A, the sequence of actions
output: O, a list of operations

initialize $O \leftarrow \emptyset$
$S \leftarrow S_0$
for $a_i \in A$ **do**
  $p \leftarrow$ get parameters of action a_i
  $R \leftarrow$ get the corresponding routine a_i
  $o_i \leftarrow$ get operations by executing R with parameters p and state S. update the state S according the operations
  o_i append o_i to the list of operations O
end
return O

composition and layout of loops on the slots of the needle bed. Figure 8 shows how a *split* action changes the state of the machine by splitting one island into two, requiring the involvement of a second yarn carrier and changing both yarn carrier poses. Thus, the translation from abstract knitting *actions* to low-level *operations* is achieved by a linear fold over the input sequence of actions. Given the initial state S_0, the algorithm iterates through the input sequence of actions A and performs the following: (1) inspect the current *state* and *action* components, (2) look up the corresponding routine(s), and (3) append the resulting operations to the list of operations O, and update the *state* accordingly. Algorithm 4 summarizes this procedure.

For illustration, Fig. 9 shows successive `knit`, `widen`, and `split` actions being executed in the context of the double torus (see Fig. 7), with the corresponding state updates and generated operations visualized in KnitPaint. In the case of the `split` action, the associated routine splits the island into two and associates a yarn carrier with each new island. The conceptual task of knitting a single course is encapsulated by a `knit-course` action. This action is implemented by the routines `KnitSJ` or `KnitDJ` (among others), for knitting in Single Jersey or Double Jersey stitch patterns respectively. When the operation generation algorithm encounters a `knit-course` action, it dispatches either of these routines depending on the stitch *types* encoded in the state at hand. The routines may additionally call other subroutines if the state's island does not conform to specific constraints. For instance, the `KnitSJ` routine may call one which transfers all stitches from the back bed to the front, converting a double-bed island to a single-bed island in preparation for knitting a Single Jersey course.

Also, this library of action types and routines is entirely extensible, allowing one to define new actions or routines that adapt an existing action to new contexts or purposes. Since action types are determined largely by the graph transformation rules, there is a certain degree of interdependence between the two rule sets. For example, a `split` graph transformation rule would require the definition of a corresponding `split` action and its routines in the routine library. However, it is important to note that these libraries are completely decoupled from the input geometry and can be reused across many situations.

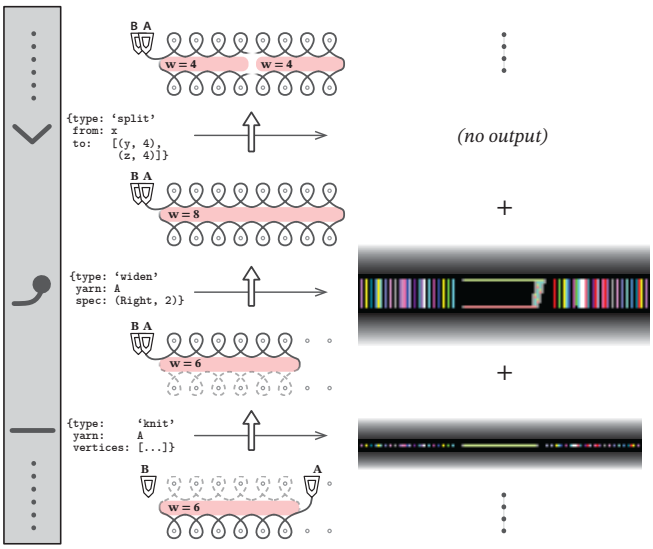

Fig. 9. The sequence of actions A is threaded through successive states to produce low-level instructions (represented here as fragments in KnitPaint format). Some actions may produce no output and only modify the state representation. For instance, the split action here splits the island of the incoming state into two.

6.3 Building the transformation rules and routine library

In this section we give some insights into building both the set of graph transformation rules and the routine library. As mentioned previously, this is expected to be carried out by specialists who are familiar with the inner workings of the target knitting machine.

We start by analyzing the shape at hand and identifying high-level tasks, i.e., *actions*, that must be carried out by the knitting machine. Considering the double torus of Fig. 2, its production can be broken down into the following tasks: `knit-course` adds a series of stitches on top of existing ones, `knit-hole` creates a hole by splitting and then merging, and `widen` and `narrow` increases and decreases the width of the course, respectively.

For generating the sequence of actions (see Algorithm 3) each transformation rule P_i is comprised of two graphs L_i and R_i. L_i is a topological representation of a particular knitting task and R_i its topologically simplified version. To build the set of transformation rules, we start by expressing each knitting task by its topological representation. We then define the replacement graph R_i and the derivation of its edges in terms of transformations of the edges in L_i. Each knitting task may have multiple topological representations. For instance, widening on the left side of the fabric is different from a widening on the right. Therefore, in order to handle the double torus example, we define a left and right variant for each widening and narrowing rule, see Fig. 7.

Each *action* then maps to multiple low-level implementations, or *routines*, see Fig. 7. A routine is a self-contained function that takes as input a machine state and outputs the instructions along with the updated state. To create the various routines of a particular action, we first list its parameters and enumerate the various initial states and scenarios in which it can be expressed. We then write a routine for each of those scenarios. For example, executing a `knit-course` action depends on the width of the course, the relative pose of the yarn carrier, and the stitch pattern (Single Jersey, Full Cardigan, ...). In our implementation, each of the `knit-course` routines handles a specific stitch pattern and generates the instructions according to the relative pose of the yarn carriers, as well as inserting kickback moves. When a `knit-course` action is invoked, the system looks up and executes the corresponding routine, see Algorithm 4.

In the supplementary material, we have included examples of several graph transformation rules and routines along with a detailed commentary.

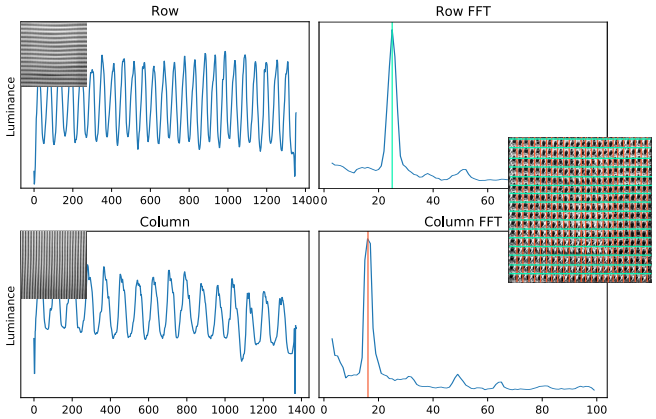

Fig. 10. To geometrically calibrate our knitting system, stitch pattern dimensions are determined from image analysis of knitted samples.

7 RESULTS

In this section, we demonstrate the capability of the *KnitKit* machine knitting system. All of the presented examples were knitted using a 15-gauge Shima Seiki MACH2XS knitting machine [2XS].

7.1 Geometric calibration

As stated in Sect. 5.1, it is possible to control the spacing between two consecutive stripes. By setting the spacing parameters to the physical stitch height and width for the course and wale stripes, respectively, we can calibrate our knitting framework to knit objects of desired dimensions. However, the distance between two consecutive courses and wales varies for different stitch patterns (and for different stitch value parameters that control loop length). For instance, the dimensions of a Single Jersey stitch pattern are different from the dimensions of a Rib stitch pattern. Moreover, those dimensions change when another material is used, e.g., glass fiber yarn would yield different stitch dimensions compared to cotton-acrylic yarn. To account for the varying spacing between two consecutive stripes, we locally modify the size parameters of the node `template` and stitch `type` according to a lookup table that contains the relative dimensions.

Here, we present a systematic method to easily obtain these dimensions, see Fig. 10. First, we knit a small rectangular sample

Stitch pattern	**Width** (*mm*)	**Height** (*mm*)
Single Jersey	1.471	1.256
Double Jersey	1.376	1.437
2x2 Rib	0.992	1.445
Full Cardigan	1.679	1.825
Linen	1.544	1.572
Pique Lacoste	2.019	0.874

Table 1. Dimensions for various stitch patterns knitted from a 30/2 cotton-acrylic yarn.

(~ 180 mm wide) with a certain stitch pattern. Then, we fix this sample inside a jig and take a photograph showing the details of the stitch pattern. In order to measure the height and width of a single stitch, we first enhance the contrast of courses and wales by applying horizontal and vertical blur filters respectively. Then, we build two 1D profiles of the image by computing the median pixel value for each row and column. Finally, we run a fast Fourier transform on the profiles and calculate the width and height of the stitch pattern from the frequency of each profile with the highest amplitude. Table 1 presents the measured dimensions of various stitch patterns for fabrics knitted with a 30/2 cotton-acrylic yarn.

Figure 11a shows the knitted output of a 75 × 300 mm rectangle where we attribute a different stitch pattern to each section of the rectangle. Despite the different dimensions of the input stitch patterns, the length of our knitted output is approximately 300 mm and its width remains close to 75 mm, even when the stitch pattern changes. This is due to the parameterization operation that locally adjusts the stripe frequency to account for the changing stitch dimensions. Moreover, using these measured stitch dimensions, our system can closely reproduce the geometric features of the input geometries. For instance, in Fig. 11b the angles formed by the knitted outputs are close to those specified in the input geometries.

7.2 Customizability and flexibility of the *KnitKit* system

The separation of our machine knitting system into two independent stages allows non-expert users to design textiles without requiring in-depth knowledge about the inner workings of machine knitting. In this section, we demonstrate the flexibility and customizability of the *KnitKit* in various scenarios.

Customizing global knitting directions. Here, we demonstrate the potential of tweaking the input vector field. This allows the user to influence the global scheduling of the knitting process, independently of the overall shape of the knitted textile. For instance, Fig. 12 shows a rectangular textile comprising three differently colored regions, i.e., different yarns. When processed with a uniform, horizontally oriented vector field, the resulting boundaries between different yarns occur in the middle of knitted courses. On the other hand, when the input knitting direction is adapted to be orthogonal to the color boundaries, the resulting transitions between different yarns occur only between successive courses. In both cases, the resulting knitted textiles have the same overall shape and color pattern, since they are generated from the same input geometry and texture, only differing in terms of their vector fields. However,

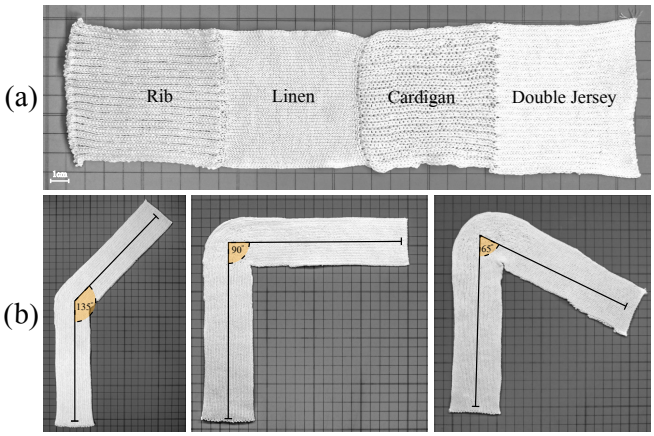

Fig. 11. The geometric calibration allows us to locally configure the parameterization process such that the resulting shape and size of the knitted output is in accordance with the geometric dimensions of the input design.

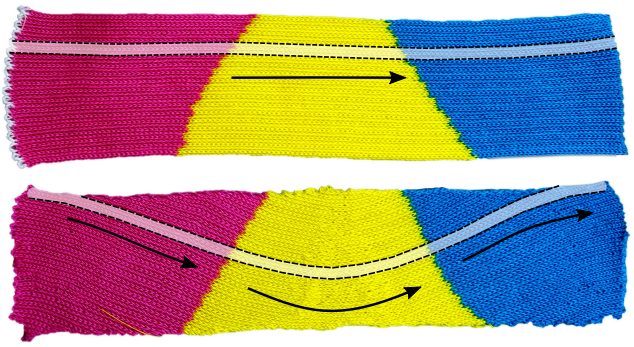

Fig. 12. The same rectangular shape with a 3-color texture was knitted with our system using an horizontally aligned (top) and an adaptive (bottom) vector field. In the latter, the color change only happens across courses and not within a course (Size: 200 x 50 mm).

the generated operations for knitting both textiles differ greatly. The top version requires the use of *intarsia* techniques that may involve extra carriage movements and kickbacks in order to deal with mid-course yarn changes. The bottom version involves short rows and shaping operations in order for yarn changes to occur between successive courses.

Customizing yarns and stitch patterns. Within the *KnitKit* system, the generation of machine instructions is not bound to any particular yarn type or stitch pattern.

In Figure 13, we map regions of an input rectangular geometry to Pique Lacoste and 2x2 Rib stitch patterns with different yarn colors. These two stitch patterns vary greatly in their dimensions and stiffness. Having calibrated these stitch pattern dimensions in Sect. 7.1, our system is able to maintain the desired proportions of the input geometry by assigning suitable numbers of wales and courses during the parameterization stage, and by dealing with internal shaping and short rows at the region boundaries during the instruction generation stage. By simply changing the input

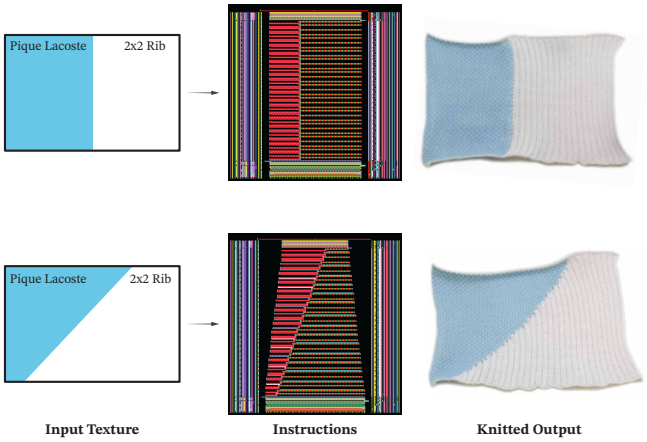

Fig. 13. A rectangular patch is knitted with two different configurations of yarn colors and stitch patterns (blue Pique Lacoste and white 2x2 Rib), which are simply controlled by regions on the input texture that map to the corresponding `template`.

texture, we can alternate between vertical and diagonal boundaries without modifying the underlying rules, even though the low-level instructions required at the boundary are quite different.

In Figure 14, we have used our system to generate textured knitted textiles from an input image, in two different styles. The first one consists of a simple 2-yarn `template` that we configure with a single flag to produce a Fair Isle colorwork. To reproduce the input image in grayscale, we set the `type` field of vertices in the `KN Node` to the dithered luminance of the image. For the second style, we have defined a complex `template` comprised of 5 yarns of different colors with the goal of reproducing the input color image using inlay stitch patterns. A white yarn is knitted in a Double Jersey stitch type and cyan (C), magenta (M), yellow (Y), and black (K) yarns are inlaid between the stitches of the white yarn. In order to configure this complex template, we define the `type` field of each vertex as a 4-flag array. In this configuration, each flag corresponds to the C, M, Y, and K yarns respectively, and setting the flag to `true` indicates that the corresponding colored yarn should be displayed on the fabric surface. In both of these cases, the expert user defines corresponding routines for the `knit-course` action, which are dispatched when the `template` is encountered. In this context, the designer only needs to provide an input image, and the low-level complexity of dealing with multiple colored yarns is hidden from the user.

Customizing action scheduling. Every topological branch in the geometry along the knitting direction requires the execution of a `split` action, which divides a contiguous island into two. Due to the inherent nature of knitting, a single yarn carrier cannot simultaneously knit across two or more islands without bridging them by floating strands of yarn. If the branches are sufficiently short, this can be resolved by a *serial* scheduling strategy. This entails entirely knitting one branch before proceeding with the other, which results in a single floating-yarn connection that is easily post-processed, i.e., cut. However, this strategy poses a problem when dealing with larger textiles or more delicate materials, as the loops of yarn on

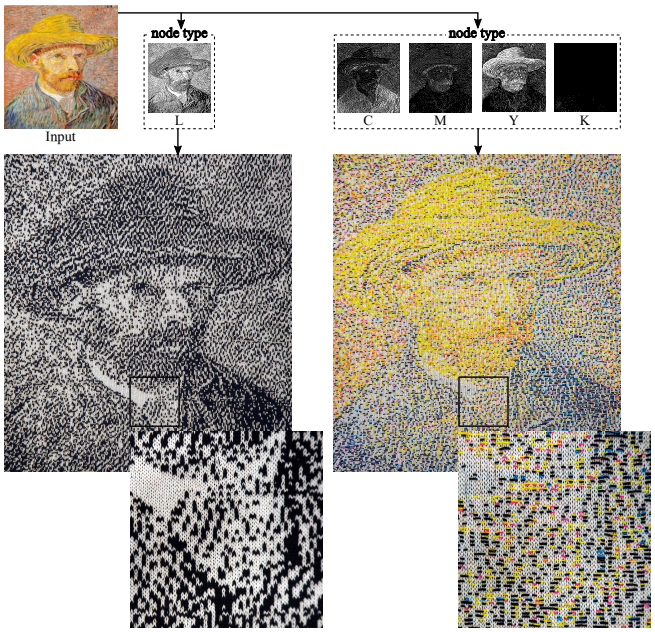

Fig. 14. Knitting a picture as a textured textile with two different templates. On the left, the dithered luminance of the input image is mapped onto a rectangular geometry to guide a 2-color Fair Isle colorwork. On the right, the CMYK channels of the same image are used to configure a complex 5-yarn custom `template` (Size: 375 x 380 mm).

both islands are subject to inconsistent amounts of tension, which might cause defects or knitting failures.

To address these potential problems, several scheduling strategies can be considered. A more conventional strategy is to introduce additional yarn carriers threaded with the same yarn type so that the islands can be knitted in parallel. However, due to the limited number of yarn carriers that are physically present on a given machine, this strategy limits the possible topologies of a knittable object. With this strategy, a machine with 4 available yarn carriers can only knit geometries with a maximum of 4 branches at any given point in the knitting process. Alternatively, we can handle splits in a more sophisticated manner by alternating between the split islands with a *tuck-and-bridge* action. Like the serial strategy, this requires only one set of carriers but with the advantage of a more uniform tension. Implementing this more complex scheduling strategy in the conventional Shima Seiki KnitPaint macro language is a highly impractical task. However, with our system this amounts to only modifying the corresponding split rule.

Figure 15 shows a Y-shaped object knitted with two different splitting strategies along with their respective generated instructions visualized in the KnitPaint format. The top version is achieved with the serial strategy of knitting the left branch entirely before switching to the right branch. In contrast, the bottom version was produced with the more complex tuck-and-bridge strategy where a single carrier knits both branches by alternating between them at intervals to minimize tension differences. This produces floating *bridges* of yarns linking the branches at each interval, which are

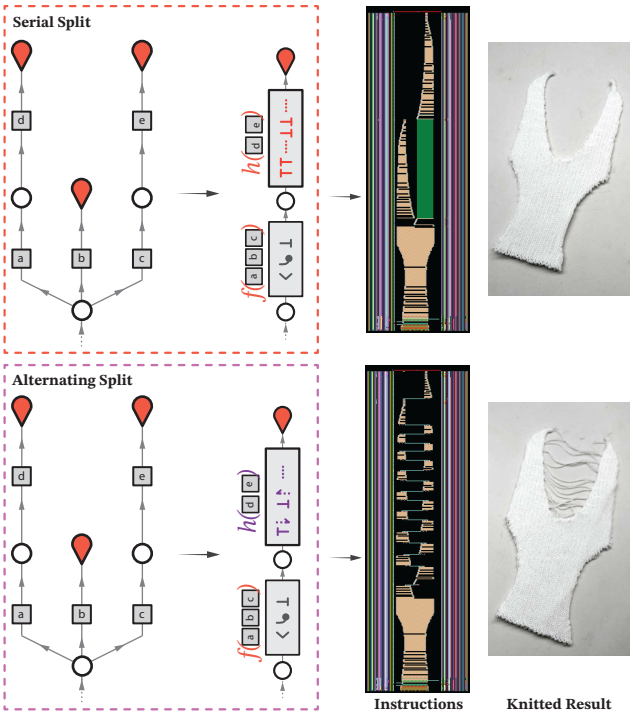

Fig. 15. Just by changing the split action rule, it is possible to generate instructions that use different scheduling strategies for dealing with split branches from the same *KnitNet*. The bottom strategy achieves more uniform tension by means of a single yarn carrier knitting both branches in an alternating manner with floating yarns as bridges.

simply cut after knitting. It is important to note here that while the two sets of instructions are quite different, they were generated from the same input *KnitNet*. The only difference is the graph transformation rules, where we substituted the serial split rule with the alternating split rule. This causes the graph collapse stage of our system to generate two different sequences of actions, each corresponding to the respective split strategy. The graph transformation is also independent of the instruction generation stage, as seen in Fig. 16 where the same `split` rule was reused with a more complex geometry and stitch `template`.

7.3 Knit design and functional applications

Our *KnitKit* system aims to make the design process independent of the low-level operations. This means that given a certain set of graph transformation rules and a routine library for instruction generation, a non-technical user can design knitted objects using typical 3D modeling and image editing tools.

To generate the examples here, we have defined a collection of 12 rules that can handle objects containing branching paths, internal shaping, and certain configurations of short rows. These rules generate about 6 classes of actions, each of which is implemented by several routines. Altogether, 30 different routines were implemented, supporting various complex stitch patterns, variations on *cast-on* and *bind-off* operations, and state manipulation operations.

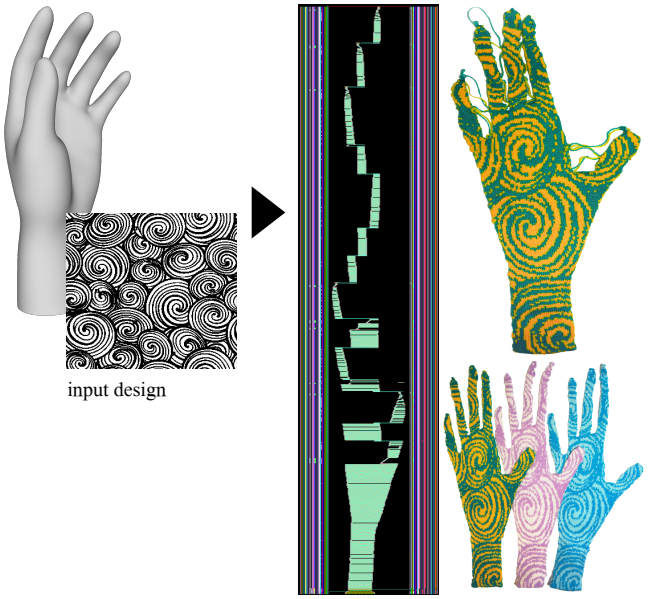

Fig. 16. A 3D hand model with a 2-colored texture is knitted using the same alternating split rule as Fig. 15b, requiring only 2 yarn carriers for branching the textured fingers and minor post processing (Size: 150 x 320 mm).

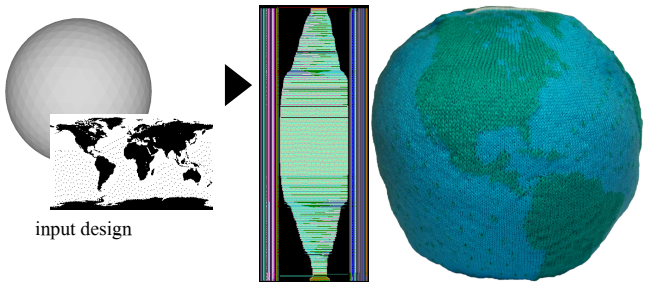

Fig. 17. An earth replica obtained by guiding the yarn template with a dithered globe texture attached to a sphere. The resulting knitted object is stuffed with polyester fiberfill to attain the 3D globe (Size: diam 180 mm).

For example, to produce the double torus of Fig. 2, the user first loads the geometry into the system. Then, a knitting direction is computed based on the Fiedler vector of the mesh Laplacian [Lévy and Zhang 2010]. The user then loads a texture, here the SIGGRAPH logo, that configures the stitch type and selects a 2-colored cotton yarn template. In this situation, the user selects the corresponding set of rules from the routine library from our collection. Now, the *KnitNet* is generated from the design inputs and subsequently processed to generate the low-level instructions. Finally, the output instructions are loaded into the machine and the SIGGRAPH-style double torus is knitted. A similar fabrication procedure was followed when producing each of the other objects such as the glove (Fig. 16, capitalizing also on the tuck-and-bridge split scheduling strategy for branches) or the globe (Fig. 17). Another example that showcases multiple high-level capabilities of the *KnitKit* system is displayed

	Gen. *KnitNet* *(Sect. 5)*	**Gen. instr.** *(Sect. 6)*	**Knitting time**
Double torus	11.1s	24s	12min 30s
Color picture	8.2s	<1s	21min
Hand	7.8s	17s	13min 30s
Globe	6.5s	9s	9min 25s

Table 2. Execution times required to generate the *KnitNet* and machine instructions for the presented applications with our implementation of the *KnitKit* system, as well as knitting times. Computation time was measured on a system with an Intel Core i7 CPU and 16 GB DDR4 RAM.

Fig. 18. A complex knitted object showcasing high-level capabilities of the *KnitKit* system. Custom 5-color stitch `templates`, action definitions for an integrated drawstring channel and shaping techniques are used to generate the object from a geometric and image texture input. Inset photos show the close-up details of the complex stitch pattern and drawstring channel.

in Fig. 18. A complex 5-color stitch `template` configured to an input image texture is used for the main body of the fabric, which is shaped according to a 3D-curved elliptical input geometry. Using customized action specifications, an integrated drawstring channel is knitted along the outer boundary to tighten the fabric around a rigid cardboard frame, making it a functional textile decoration element.

These examples demonstrate that our proposed *KnitKit* system makes machine knitting more accessible. The prototyping of knitted textiles can be greatly simplified, since altering the designs only requires a change in the corresponding textures. This is particularly important for emerging applications of technical textiles with added functionality, where readily available, commercial templates do not exist. Table 2 shows the detailed time required in our examples to generate the machine instructions from the input designs. In its current implementation, our system can generate instructions for complex objects within minutes.

8 LIMITATIONS AND FUTURE WORK

Input knitting direction. The proposed *KnitKit* system requires a vector field to be given as an input that represents the intended knitting direction. In our implementation, we have computed it by evaluating either the gradient of a linear function defined on input geometry, e.g., the *time function* from [Narayanan et al. 2018], or the gradient of low order eigenvectors of the mesh Laplacian [Lévy and Zhang 2010], or by computing a globally optimal vector field [Knöppel et al. 2015]. This vector field has direct influence on the remeshing output and thus directly affects the *KnitNet* structure. In particular, its curl and divergence influence the occurrence of T-junctions that translate into short rows and internal shaping of the knit structure. Furthermore, the physical properties of the yarn, as well as the stitch pattern, place restrictions on the mechanical stress that the yarn can withstand during the knitting procedure. These restrictions limit the machine's capacity to knit a large number of successive short rows or perform an arbitrary number of transfers onto the same needle, since these operations might cause the fabric to tear or have loop formation errors. Thus, an input vector field with a large change in curl or divergence, which would cause successive short rows or many transfers, should be avoided. Here, we have smoothed the input vector field in order to avoid these issues. However, in future work, it would be interesting to explore the relationship between the input vector field and the generated knit structure in more detail and develop tools to generate input vector fields that take the yarns' physical constraints into consideration. This would enable the user to have even more control over the knitting process and allow the system to perform knitting-aware optimization of the input vector field in order to ensure a stable knitting process.

Impossible geometries. In case the geometry contains multiple concurrent non-planar splits, it is impossible to compute a valid segmentation of the vertices in each row of the quad mesh. Thus, a proper allocation of the vertices to either the front or back bed cannot be found and consequently the system will fail to generate a valid *KnitNet*.

Knittability of the KnitNet. The instruction generation part of our system assumes that the *KnitNet* is a directed acyclic graph. In case the input design is a 3D geometry, the vertices of the corresponding *KnitNet*'s `KN Nodes` should form two contiguous groups, each corresponding to either the front or back bed. These constraints on the structure of the *KnitNet* are weaker than the knittablity conditions laid out in [Narayanan et al. 2018], which require the input 3D geometry to have a global planar embedding, such as branches with braided crossings. Our system can successfully build a *KnitNet* for such geometries and is able to generate their corresponding knitting instructions if the set of graph transformation rules includes the ones that match to non-planar graphs. Therefore, the knittability of the *KnitNet* is not only determined by constraints on its topology, but also by the set of user defined graph transformation rules. In the future, we aim to further investigate and formalize the knittability properties of the *KnitNet*.

Graph transformation rules and routine library. Building or extending the set of rules and routines is expected to be carried out by technical users. It requires a detailed knowledge of the knitting process and some experience with our system in order to be able to identify the abstract actions that can be performed by the machines, determine their various parameters and translate that information to rules and routines. Readers can refer to the supplementary material for examples of graph transformation rules and routines. In the future, formalizing the notion of *KnitNet* knittablity would allow us to propose a more systematic approach for the development of graph transformation rules and routine libraries for our system.

Refinement of state model. The current representation of the machine state could be further enhanced, for instance, by taking into account the front-to-back positioning of the yarn carriers. If two carriers share the same rail, it is physically impossible for them to cross each other or come closer than a specified distance. However, the current state model treats them as being able to move completely independently of each other, and thus illegal instructions that result in collisions of carriers on the same rail could be generated. Rail allocations can also interact with the spatial layout of islands on the needle bed to produce undesired phenomena known as "yarn tagging". In the future, it should be possible to extend the action processing stage by taking the yarn-to-rail allocation as input and augmenting the logic to handle these cases. This would enable the detection of carrier collisions or yarn tags, and the system could potentially suggest an optimal rail allocation or insert additional operations to resolve such issues.

9 CONCLUSION

We have presented the *KnitKit* as a flexible system for the computational design and manufacturing of customizable textiles using CNC knitting machines. The aim of this system is to provide a geometry- and machine-independent workflow for the machine knitting of textiles. It manages to decouple the high-level design aspect of producing knitted textiles from the complexities and low-level specificities of knitting machines. This offers non-expert users the possibility to design knitted textiles with customized, complex, 3-dimensional geometries and intricate design patterns. At the same time, it provides knitting experts with the flexibility to implement additional stitch patterns and output machine knitting instructions independently from the input design. For example, Figure 1 shows a textured hand grabbing a globe model, which were both produced with our knitting system from 3D meshes with attached 2D textures.

This has been realized using the *KnitNet*, a row-based directed graph data structure, as a central component. It provides an abstract interface between the high-level specification of geometry, design pattern, and knitting direction and the low-level knitting instructions. At the high-level design stage, an algorithm has been developed that generates the *KnitNet* representation from an input mesh geometry and texture-defined specifications.

Furthermore, a two-stage algorithm for the low-level instruction generation has also been developed, which translates the *KnitNet* graph to machine-specific knitting instructions using a set of graph transformation rules and a routine library that can be customized by knitting experts.

The capabilities and potential of the *KnitKit* system have been demonstrated by applying the complete workflow from high-level knitting design, to low-level knitting instructions, and to fabrication for several examples with varying complexities in terms of geometries and patterns. We show that it is possible to design and fabricate textiles with precise geometric dimensions, optimize results by varying the knitting direction at the high-level stage or instruction generation rules at the low-level stage, and define complex stitch configurations that enable users to easily create textiles with highly-customized designs using multiple yarns and customized stitch patterns.

ACKNOWLEDGMENTS

The authors acknowledge support from the SUTD Digital Manufacturing and Design (DManD) Centre, supported by the Singapore National Research Foundation. Sai-Kit Yeung was partially supported by an internal grant from HKUST (R9429). We would also like to thank Tan Ying Yi for his support in improving the quality of the figures and the anonymous reviewers for their insightful comments. The hand model is from the aim@shape repository courtesy of Inria.

REFERENCES

Julianna Abel, Jonathan Luntz, and Diann Brei. 2012. A two-dimensional analytical model and experimental validation of garter stitch knitted shape memory alloy actuator architecture. *Smart Materials and Structures* 21, 8 (Aug. 2012), 085011.

Carlos Aliaga, Carlos Castillo, Diego Gutierrez, Miguel A Otaduy, Jorge Lopez-Moreno, and Adrian Jarabo. 2017. An appearance model for textile fibers. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 35–45.

David Bommes, Timm Lempfer, and Leif Kobbelt. 2011. Global structure optimization of quadrilateral meshes. *Computer Graphics Forum* 30 (2011), 375–384.

Xiaogang Chen (Ed.). 2015. *Advances in 3D textiles*. Woodhead Publishing.

Gabriel Cirio, Jorge Lopez-Moreno, and Miguel A Otaduy. 2016. Yarn-level cloth simulation with sliding persistent contacts. *IEEE Transactions on Visualization and Computer Graphics* 23, 2 (2016), 1152–1162.

Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. 1997. Algebraic approaches to graph transformation–part I: Basic concepts and double pushout approach. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*. World Scientific, 163–245.

Shen Dong, Scott Kircher, and Michael Garland. 2005. Harmonic functions for quadrilateral remeshing of arbitrary manifolds. *Computer Aided Geometric Design* 22, 5 (2005), 392 – 423.

Yuki Igarashi, Takeo Igarashi, and Hiromasa Suzuki. 2008. Knitting a 3D model. *Computer Graphics Forum* 27 (2008), 1737–1743.

Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. 2019. Knitting skeletons: a computer-aided design tool for shaping and patterning of knitted garments. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 53–65.

Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2015. Stripe patterns on surfaces. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 39:1–39:11.

Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug James, and Steve Marschner. 2018. Interactive design of periodic yarn-level cloth patterns. *ACM Transactions on Graphics (TOG)* 37, 6 (2018).

Kok Hoonh Leong, Seeram Ramakrishna, and Zhengming Huang. 2000. The potential of knitting for engineering composites—a review. *Composites Part A: Applied Science and Manufacturing* 31, 3 (2000), 197–220. https://doi.org/10.1016/S1359-835X(99)00067-6

Bruno Lévy and Hao (Richard) Zhang. 2010. Spectral mesh processing. In *ACM SIGGRAPH 2010 Courses* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 312 pages.

Minchen Li. 2018. *FoldSketch: enriching garments with physically reproducible folds*. Ph.D. Dissertation. University of British Columbia.

James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A compiler for 3D machine knitting. *ACM Transactions on Graphics (TOG)* 35, 4 (2016).

Michael Meißner and Bernd Eberhardt. 1998. The art of knitted fabrics, realistic & physically based modelling of knitted patterns. In *Computer Graphics Forum*, Vol. 17. Wiley Online Library, 355–362.

Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. 2018. Automatic machine knitting of 3D meshes. *ACM Transactions on Graphics (TOG)* 37, 3 (2018).

Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual knitting machine programming. *ACM Transactions on Graphics* 38, 4, Article 63 (2019), 13 pages.

Mariana Popescu, Matthias Rippmann, Tom Van Mele, and Philippe Block. 2018. *Automated generation of knit patterns for non-developable surfaces*. Springer Singapore, Singapore, 271–284.

E.J. Power. 2015. Chapter 12 - Yarn to fabric: knitting. In *Textiles and Fashion*, Rose Sinclair (Ed.). Woodhead Publishing, 289 – 305. https://doi.org/10.1016/B978-1-84569-931-4.00012-X

Shima Seiki. 2020. *SDS-One Apex*. https://www.shimaseiki.com/product/design/ Accessed: 2020-05-22.

Shima Seiki. 2XS. Shima Seiki MACH2XS product page. https://www.shimaseiki.com/product/knit/mach2xs/. Accessed: 2021-01-28.

David J Spencer. 2001. *Knitting technology: a comprehensive handbook and practical guide*. Woodhead Publishing.

Stoll. 2020. *M1PLUS*. https://www.stoll.com/en/software/m1plusr/ Accessed: 2020-05-22.

Julian Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42.

Jenny Underwood. 2009. The design of 3D shape knitted preforms. (2009).

Katja Wolff and Olga Sorkine-Hornung. 2019. Wallpaper pattern alignment along garment seams. *ACM Transactions on Graphics (TOG)* 38, 4 (2019).

Kui Wu, Xifeng Gao, Zachary Ferguson, Daniele Panozzo, and Cem Yuksel. 2018. Stitch meshing. *ACM Transactions on Graphics (TOG)* 37, 4 (2018).

Kui Wu, Hannah Swan, and Cem Yuksel. 2019. Knittable stitch meshes. *ACM Transactions on Graphics* 38, 1 (2019).

Kui Wu, Marco Tarini, Cem Yuksel, James Mccann, and Xifeng Gao. 2021. Wearable 3D machine knitting: automatic generation of shaped knit Sheets to cover real-world objects. *IEEE Transactions on Visualization Computer Graphics* 01 (2021), 1–1.

Kui Wu and Cem Yuksel. 2017. Real-time cloth rendering with fiber-level detail. *IEEE Transactions on Visualization and Computer Graphics* 25, 2 (2017), 1297–1308.

Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2012. Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Transactions on Graphics (TOG)* 31, 4 (2012).