Synthesis Lectures on Digital Circuits & Systems

Series Editor

Mitchell A. Thornton, Southern Methodist University, Dallas, USA

This series includes titles of interest to students, professionals, and researchers in the area of design and analysis of digital circuits and systems. Each Lecture is self-contained and focuses on the background information required to understand the subject matter and practical case studies that illustrate applications. The format of a Lecture is structured such that each will be devoted to a specific topic in digital circuits and systems rather than a larger overview of several topics such as that found in a comprehensive handbook. The Lectures cover both well-established areas as well as newly developed or emerging material in digital circuits and systems design and analysis.

Ahmadreza Farsaei

Introduction to Layout Design and Automation of Photonic Integrated Circuits

Springer

Ahmadreza Farsaei
LuDALog LLC
San Jose, CA, USA

ISSN 1932-3166 ISSN 1932-3174 (electronic)
Synthesis Lectures on Digital Circuits & Systems
ISBN 978-3-031-25287-7 ISBN 978-3-031-25288-4 (eBook)
https://doi.org/10.1007/978-3-031-25288-4

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my parents who are my roots

For my wife Niloofar and children Nick and Rumi
who are the source of my hope

Preface

Silicon photonics is a promising platform and technology whose application covers not only high-speed communication links used in data centers, but also novel applications in biological sensing, Light Detection and Ranging (LiDAR), and co-packed optics [1, 2]. The design complexity of Photonic Integrated Circuits (PICs) supporting these applications has been significantly increased both in terms of architecture and the number of device instances used to create the final mask layout. PIC designs' evolving complexity requires a unified and salable design automation solution to address their unique challenges. Although there have been many textbooks [3] on fundamental concepts and basic physics of silicon photonics, there is not a body of literature providing proper education on PIC design automation.

There are three main challenges that an Electronic Photonic Design Automation (EPDA) solution should address [4–6]: (1) Implementation of photonic components' complex curvilinear shapes, (2) Enabling a unified electro-optical co-simulation flow, (3) Native support for physical verification and extraction of complex curvilinear shapes. In this book, I am mainly focused on the implementation of complex photonic components' curvilinear shapes. Traditional Electronic Design Automation (EDA) solutions are based on supporting rectilinear/octilinear concepts. This implies that these solutions are not capable of reliably supporting complex curvilinear shapes required to implement a photonic component's mask layout. Proper implementation of an EPDA solution should be derived from an EDA solution with an additional mathematical engine to: (1) Capture curvilinear shapes using parametric mathematical equations, (2) Manipulate curves and extract curves' characteristics, (3) provide required software constructs to represent a photonic device using parametric mathematical equations, and (4) establish a link between mathematical domain and discretized polygon domain for the final generation of mask layout.

The book comprises six chapters that are structured to be very concise, practical, and example-driven. The content of each chapter is chosen carefully and is based on the training I have had for professional and industry experts. Each chapter starts with a main concept and continues with code examples and use cases to support that concept. Each chapter is written in such a way that you should be able to read and comprehend the main

concept, and write and test code examples in a day. I have included some exercises inside each chapter so that you can evaluate your progress and understanding of the concepts.

Chapters 1–3 are mainly focused on photonic devices' physical implementation and introduction to Virtuoso® CurvyCore™ Application Programming Interfaces (APIs). Chapter 4 is focused on the Parameterized Cell (PCell) concept and implementation as well as some fundamental concepts behind a hierarchical design methodology such as terminals, pins, instance terminals, etc. Chapter 5 is focused on how to solve some design problems by SKILL®™ scripting. I have included Chap. 6 to introduce proper scalable software development methodology using SKILL++.

Virtuoso is the tool used for the implementation of my designs and the Virtuoso CurvyCore mathematical engine is used to implement photonic devices' physical design (layout). SKILL is the main scripting language used for code implementation. All the examples provided in the book are based on Generic Optical Process Design Kit (GoPDK), which is a generic optical PDK developed by Cadence Design Systems, Inc. to demonstrate their photonic solution capabilities.

In this book, I have assumed you have basic knowledge of algorithm, software development, data structures, and SKILL programming. Although deep knowledge of using Virtuoso design suites such as Virtuoso Layout and Schematic Suites is not needed, you should be familiar with how to perform some basic tasks such as create a new library and cell views. Although you can use any editor to write your SKILL programs, I recommend you become familiar with SKILL Integrated Development Environment (IDE) to enhance the productivity and efficiency of your software development cycle.

Who would benefit from this book? I have written this book with two target audiences, academia and industry, in mind. Academia: any senior undergraduate/graduate student pursuing a career in integrated photonics. Industry: PIC circuit and mask designers, PIC component developers, Process Design Kit (PDK) developers, and PIC Computer-Aided Design (CAD) engineers.

San Jose, USA Ahmadreza Farsaei
November 2022

References

1. S. Tanaka, T. Akiyama, S. Sekiguchi, and K. Morito, "Silicon photonics optical transmitter technology for tb/s-class i/o co-packaged with cpu," *Fujitsu Sci. Tech. J*, vol. 50, no. 1, pp. 123–131, 2014.
2. C. V. Poulton, A. Yaacobi, D. B. Cole, M. J. Byrd, M. Raval, D. Vermeulen, and M. R. Watts, "Coherent solid-state lidar with silicon photonic optical phased arrays," *Optics letters*, vol. 42, no. 20, pp. 4091–4094, 2017.
3. A. Yariv and P. Yeh, *Photonics: optical electronics in modern communications.* 1em plus 0.5em minus 0.4em Oxford university press, 2007.

4. G. Lamant, J. Flueckiger, F. Villa, A. Farsaei, B. Wang, R. Stoffer, X. Wang, J. Pond, and T. Korthorst, "Schematic driven simulation and layout of complex photonic ic's," in *2016 IEEE 13th International Conference on Group IV Photonics (GFP)*. 1em plus 0.5em minus 0.4em IEEE, 2016, pp. 92–93.
5. J. Pond, X. Wang, Z. Lu, E. Schelew, G. Lamant, and A. Farsaei, "Latest advancements to the industry-leading epda design flow for silicon photonics," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1em plus 0.5em minus 0.4em IEEE, 2019, pp. 1–6.
6. J. Pond, G. S. Lamant, and R. Goldman, "Software tools for integrated photonics," in *Optical Fiber Telecommunications VII*. 1em plus 0.5em minus 0.4em Elsevier, 2020, pp. 195–231.

Acknowledgments

I have been teaching in academia and educating industry experts on the domain of Electronic Photonic Design Automation (EPDA) for many years through different venues such as workshops and conferences. Their feedback established the need for a more practical and educational solution for this new domain in my mind. I am very thankful to all of them for their feedback and motivation. Among them are Shahriar Mirabbasi (University of British Columbia), Lukas Chrostowski (University of British Columbia), Sudip Shekhar (University of British Columbia), Ashkan Seyedi (Nvidia), James Pond (Ansys), Mahdi Nikdast (Colorado State University), Ali Asghar Eftekhar (Amazon Web Services), Saeed Fathololoumi (Intel), and Chenyang Wu (Intel).

I am very thankful for the support of Cadence Design System experts and business leaders in this process. In particular, Gilles Lamant, who read and provided very helpful suggestions as well as coordinated and supported the feasibility of publishing the book.

Although this effort has been driven by my passion and responsibility to contribute to the integrated photonics ecosystem, I am sincerely thankful to Intel's management/leadership for recognizing this need and supporting me. Among them are Sreevatsa Sreekantham, Yuliya Akulova, Todd Swanson, and Hong Hou.

The timely production of the book was made possible by the hard work of the staff at Springer. I also thank them for their patience during this process.

Disclaimer

The code examples provided in this book are just for educational purposes and need proper modification before being used in production. Any CurvyCore APIs, SKILL/SKILL++ functions and constructs, database-level APIs, and application-level APIs used in this book are subject to change without notice by Cadence Design Systems, Inc.

Virtuoso®, SKILL®™, and CurvyCore™ are trademarks of Cadence Design Systems, Inc. Cadence grants permission to print examples in this book written in SKILL and using licensed and copyrighted Cadence APIs. Cadence disclaims all warranties and Cadence reserves the right to make changes at any time to the published APIs.

Contents

List of Figures

List of Codes

Curves Implementation

1

1.1 Curves

Curves are basic building blocks in the implementation of photonic devices in the physical(Layout) domain. From passive devices such as photonic waveguides to active devices such as ring modulators and photodetectors. Access to curve equations and basic curve characteristic through properly defined APIs are essential for verification and simulation purpose of a PIC design. Having a good understanding of these APIs, which will be introduced in this chapter, is essential to properly implement a photonic component's physical design. This become much more important when you want to decompose an arbitrary waveguide into its constituents by interpreting its parametric equations. Without these APIs, you need to reconstruct a waveguide's centerline curve using its discretized polygon representation which is not a trivial task.

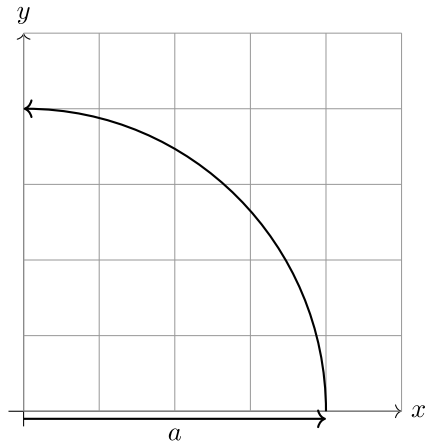
In this chapter, we will discuss how a curve is defined and represented in Virtuoso® CurvyCore™ and learn how to implement that using CurvyCore Application Programming Interfaces (APIs).

1.2 Parametric Curves

Curves are represented as parametric equations. This means that instead of $y = f(x)$, they should be represented as $x(t)$, $y(t)$ where both x and y depend on a third variable called a parameter. In Virtuoso this variable is defined as t.

Curves, which are represented as ccCurve objects, are the basic building blocks of photonic components. In most cases, they specify the path along with the optical mode propagates. So the first step to implement a physical representation of a photonic component is to be able to describe the main curves associated with the physical shapes in a parametric form.

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_1

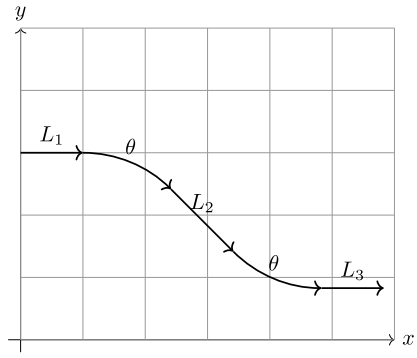


$$x(t) = a \cdot \cos(t); y(t) = b \cdot \sin(t)$$
$$0 \leq t \leq \frac{\pi}{2}$$

Fig. 1.1 Parametric representation of an arc bend

In simple cases such as an arc bend shown in Fig. 1.1, the curve can be described as a single curve. There are cases such as a generalized S-Bend shown in Fig. 1.2 where its representative curve is created by stitching multiple curves. The stitched curve is defined as a new object called a polycurve. When a polycurve is created, user would be able to pass some certain conditions such as the continuity of the polycurve. For the most of polycurves used in photonic devices, the following conditions should be met.

1. The end point of the fist curve should coincide with the begin point of the next curve. This means your final curve should be continuous.
2. The curve should have a continuous first order derivative.



$$x(t) = t; y(t) = 0$$
$$0 \leq t \leq L_1$$
$$x(t) = R_1 \cdot \sin(t); y(t) = R_1 \cdot \cos(t)$$
$$0 \leq t \leq \theta$$
$$x(t) = t; y(t) = t \cdot tan(\theta)$$
$$0 \leq t \leq L_3$$
$$x(t) = R_2 \cdot \cos(t); y(t) = R_2 \cdot \sin(t)$$
$$\frac{3\pi}{2} - \theta \leq t \leq \frac{3\pi}{2}$$
$$x(t) = t; y(t) = 0$$
$$0 \leq t \leq L_3$$

Fig. 1.2 Parametric representation of an S-bend

3. It is also recommended that the curve should have continuous second order derivative. This means that the radius of the curvature is changing continuously along the polycurve.

Code 1.1 is an example of how a 90° arc is created. An arc curve with different spanning angle, θ, can be used to generate different arc bend waveguides which can be used as a building block of more complex waveguides such as an S-bend as shown in Fig. 1.2. The S-bend shown in Fig. 1.2 is a special case of a generalized S-bend shown in Fig. 1.3 where $\theta_5 = 0$ and $\theta_4 = 0$.

Code 1.1 SKILL®^{TM} code to implement an arc bend

```
1 defMathConstants('m)
2 arcRadius = 4.0
3 arcX = lsprintf("%0.3f*cos(t)" arcRadius)
4 arcY = lsprintf("%0.3f*sin(t)" arcRadius)
5 arcCurve = ccCreateCurve(arcX arcY 0.0:m.PI)
```

As shown in Figs. 1.1, 1.2 and 1.3, we are trying to find common pattern so that more complex curves can be generated from some basic and simple building blocks. Another example is a directional coupler shown in Fig. 1.4. As is shown, a typical directional coupler's curves can be created based on four Sbend structures.

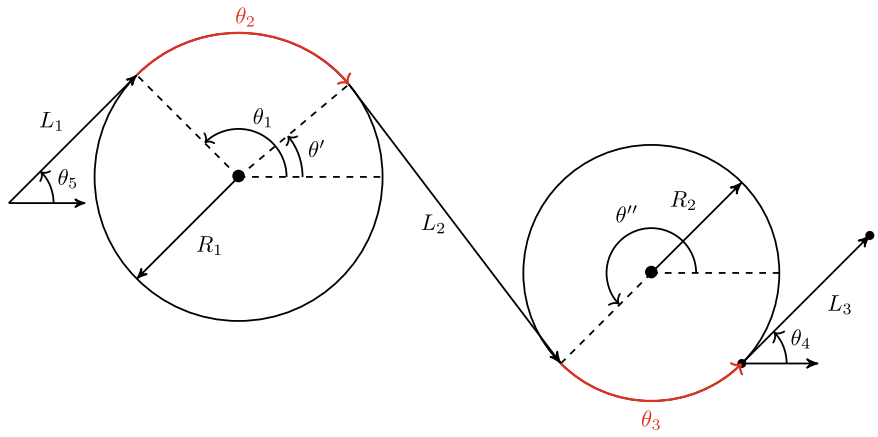


Fig. 1.3 Generalized S-bend; $\theta_1 = \theta_5 + \frac{\pi}{2}, \theta'' = \pi + \theta', \theta_4 = \theta'' + \theta_3 - \frac{3\pi}{2}$

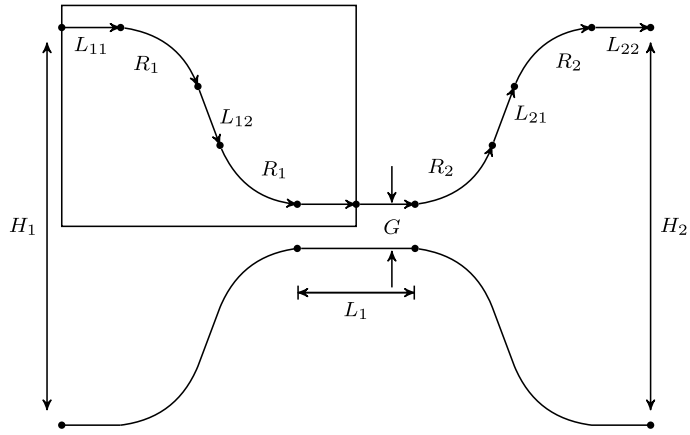


Fig. 1.4 A typical directional coupler curves based on four S-bend curves

1.3 Case Study

The example code shown in Code 1.1 can be converted to a general "procedure" to be used in implementing complex curves. The procedure gets radius, range of angle variation, and convexity of an arc; and generates its associated curve object. `convexity` argument defines whether the arc curve would be convex or concave as is shown in Fig. 1.5. Another object type in CurvyCore is a polycurve. A polycurve is a sequence of curve objects that are connected in such a way that the the end point of a curve is the start point of the next curve. A user can impose certain conditions on the smoothness of the connection points between the curves. The minimum requirement to create a polycurve from multiple curves is that the end point of a curve is the start point of the next curve; i.e., the final polycurve would be continuous. A user can specify first-order and/or second-order continuity making sure that

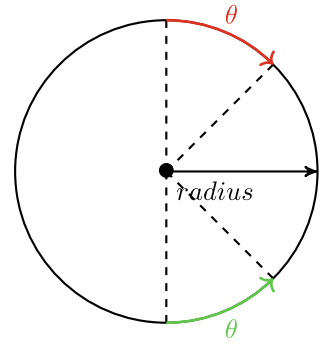


Fig. 1.5 Red and green arcs define concave and convex arc respectively

the connected curves have a continuous first-order and/or second-order derivatives at the connection points respectively.

Code 1.2 SKILL code to implement a general procedure creating an arc bend

```
1  procedure(LUDACreateArcCurve(radius thetaRange convexity)
2      prog((mathConstants PI clockwise arcX arcY arcCurve)
3          defMathConstants('mathConstants)
4          PI = mathConstants.PI
5          cond(
6              (convexity == "Convex"
7                  clockwise = -1.0
8              )
9              (convexity == "Concave"
10                 clockwise = 1.0
11             )
12             (t
13                 warn("Convexity hasn't been set properly.")
14                 return(nil)
15             )
16         )
17         arcX = lsprintf("%.12f*sin(t)" radius)
18         arcY = lsprintf("%.3f*%.12f*cos(t)" clockwise radius)
19         arcCurve = ccCreateCurve(arcX arcY car(thetaRange)/180.0*PI
               :cadr(thetaRange)/180.0*PI)
20         return(arcCurve)
21     )
22 )
23 ;Example
24 cv = geGetEditCellView()
25 arcCurveConcave = LUDACreateArcCurve(10.0 0:45.0 "Concave")
26 arcCurveConvex = LUDACreateArcCurve(10.0 -45.0:45.0 "Convex")
27 ccGenFigs(ccCreateLine(cv "waveguide" ccCreatePolyCurve(list(
       arcCurveConcave))))
28 ccGenFigs(ccCreateLine(cv "waveguide" ccCreatePolyCurve(list(
       arcCurveConvex))))
```

The easiest way to get a graphical representation of a curve in Virtuoso Layout tools is to create a ccLine object from a curve and then use `ccGenFigs` function to generate the figure associated with the ccLine object representing the curve. Considering that `ccCreateLine` function accepts a ccPolyCurve as an input argument, the single arc curve needs to be converted to a polycurve object using `ccCreatePolyCurve` function as shown in lines 27 and 27 of Code 1.2. The result of loading Code 1.2 is shown in Fig. 1.6.

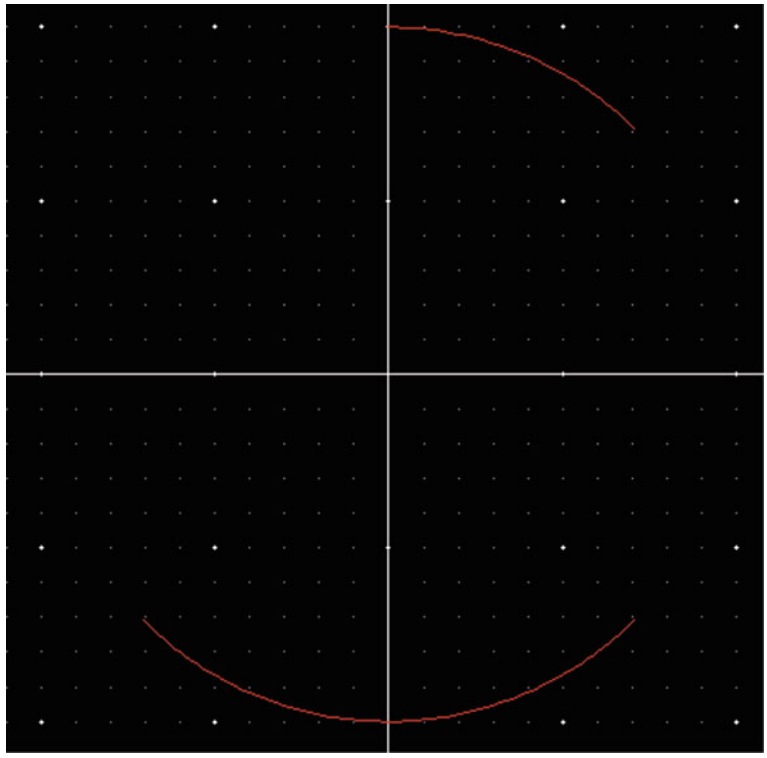


Fig. 1.6 Result of loading Code 1.2 to create sample concave and convex arc curves

1.4 Curve Stitching

Most of photonic components need more than a single curve to be represented properly. For example, an S-Bend curve consist of at least two arc bend curves. A user can implement those curves independently and manipulate them for their final shape;however, this is not an efficient method and would create a complicated and non-reliable code. A better solution would be to combine as many single curves as possible into a polycurve. For example, an S-Bend shown in Fig. 1.2 consist of 5 curves that can be combined into a single polycurve. creating a polycurve from multiple curves is called curve stitching. Curve stitching is based on finding end point of a curve and then shift the next curve in such a way that its start point coincides with the end point of previous curve. This can be implemented in different ways, one implementation is shown in Code 1.3. `LUDAStitchCurves` gets a list of curves to be stitched and two other parameters: (1) offset: since the default reference point for the stitched curve is origin, offset value can be used to offset the default origin; (2) rotation: specifies the rotation value for the stitched polycurve.

Code 1.3 SKILL procedure to implement a curve stitching solution

```
procedure(LUDAStitchCurves(curves @key
                        (offset 0:0)
                        (rotation 0.0)
                        )
    prog((shiftVectorX shiftVectorY stitchedCurve tempCurve endPt
        refPolyCurve)
        unless(forall(curve curves curve~>objType == "ccCurve")
            warn("Input arguments should be of \"ccCurve\" type")
            return(nil)
        )
        endPt = 0.0:0.0
        stitchedCurve = foreach(mapcar curve curves
                            shiftVectorX = xCoord(endPt) - xCoord(
                                ccGetCurvePoint(curve car(curve~>
                                range)))
                            shiftVectorY = yCoord(endPt) - yCoord(
                                ccGetCurvePoint(curve car(curve~>
                                range)))
                            tempCurve = ccTransformPolyCurve(
                                ccCreatePolyCurve(list(curve)) list(
                                shiftVectorX:shiftVectorY 0.0))
                            endPt = tempCurve~>endPt
                            car(tempCurve~>segments)
                        )
        refPolyCurve = ccCreatePolyCurve(stitchedCurve)
        return(ccTransformPolyCurve(refPolyCurve list(offset
            rotation)))
    )
)
```

Using Code 1.3, an S-Bend curve can be implemented efficiently using Code 1.4. In this example, we have used previously defined `LUDACreateArcCurve` to create arc curves. We also implemented a new function, `LUDACreateStraightLine`, to implement a straight line curve with an arbitrary angle which is used between two arc curves in the S-Bend.

Code 1.4 Create an S-Bend by stitching multiple arc and straight curves

```
procedure(LUDACreateStraightLine(lineLength angle)
    prog((mathConstants)
        defMathConstants('mathConstants)
        PI = mathConstants.PI
        return(ccCreateCurve(lsprintf("%.12f*t" cos(angle/180.0*PI)
            ) lsprintf("%.12f*t" sin(angle/180.0*PI)) 0:lineLength)
            )
    )
)
;Example
```

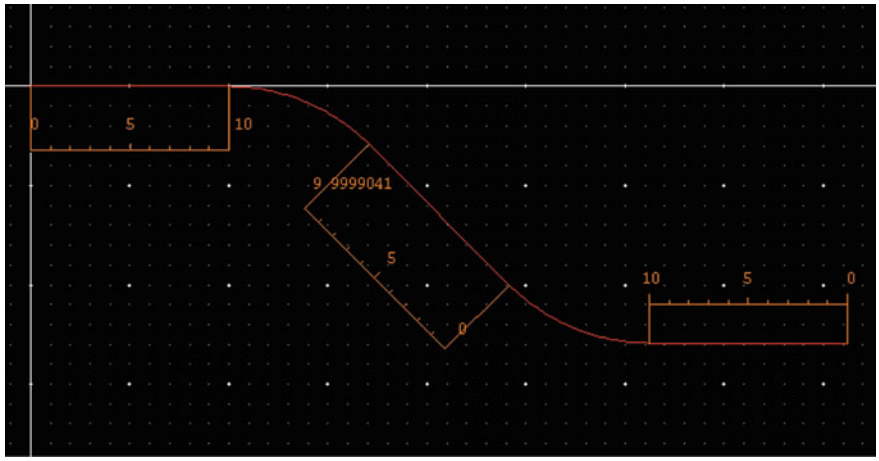


Fig. 1.7 Result of loading Code 1.4 to create a typical S-Bend based on stitching multiple curves

```
9  sBendCurve1 = ccCreateCurve("t" "0" 0:10)
10 sBendCurve2 = LUDACreateArcCurve(10.0 0:45 "Concave")
11 sBendCurve3 = LUDACreateStraightLine(10 -45)
12 sBendCurve4 = LUDACreateArcCurve(10.0 -45:0 "Convex")
13 sBendCurve5 = ccCreateCurve("t" "0" 0:10)
14 sBendPolyCurve = LUDAStitchCurves(list(sBendCurve1 sBendCurve2
       sBendCurve3 sBendCurve4 sBendCurve5))
15 ccGenFigs(ccCreateLine(cv "waveguide" sBendPolyCurve))
```

The result of loading Code 1.4 is shown in Fig. 1.7. The SBend consists of three straight lines with $length = 10.0$ and two arc bends with $radius = 10.0$ and $angle = 45.0°$.

Exercise: Create a procedure that gets a typical S-Bend parameters shown in the previous section and return a polycurve representing the S-Bend.

1.5 Case Study

In this section we are going to create the top-branch of a directional coupler shown in Fig. 1.4. As is shown the top branch of the directional coupler consist of two S-Bend curves. Code 1.5 shows a typical S-Bend implementation; the only new parameters which is introduced is `direction` specifying whether S-Bend's end point is above or below the start point. This parameter can be useful because the top-branch of directional coupler consists of two S-Bends each have different direction. There are two ways that these two S-Bends can be joined: (1) create two S-Bends, SBendLeft and SBendRight, find the end point of SBendLeft and when creating SBendRight, use that as the offset value; this solution would have two polycurves; (2) create two S-Bends, SBendLeft and SBendRight,find extract constituent

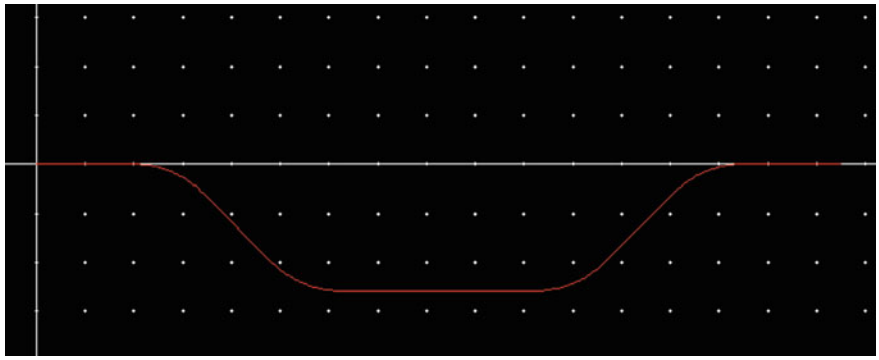


Fig. 1.8 Result of loading Code 1.5 to create the top branch of directional coupler shown in Fig. 1.4

curves of each polycurve and use curve stitching to create a single polycurve. Line 30 and 31 in Code 1.5 shows how the curve elements of a polycurve object can be extracted using its `segments` property. The result of loading Code 1.4 is shown in Fig. 1.8.

Code 1.5 A typical S-Bend implementation

```
1  procedure(LUDACreateSBend(straight1L straight2L straight3L radius
       thetaRange @key
2                        (direction "Up")
3                        (offset 0:0)
4                        )
5      prog((mathConstants PI clockwise arcX arcY arcCurve)
6          defMathConstants('mathConstants)
7          PI = mathConstants.PI
8          sBendCurve1 = ccCreateCurve("t" "0" 0:straight1L)
9          cond(
10             (direction == "Down"
11                 sBendCurve2 = LUDACreateArcCurve(radius 0:
                       thetaRange "Concave")
12                 sBendCurve3 = LUDACreateStraightLine(straight2L -
                       thetaRange)
13                 sBendCurve4 = LUDACreateArcCurve(radius -thetaRange
                       :0 "Convex")
14             )
15             (direction == "Up"
16                 sBendCurve2 = LUDACreateArcCurve(radius 0:
                       thetaRange "Convex")
17                 sBendCurve3 = LUDACreateStraightLine(straight2L
                       thetaRange)
18                 sBendCurve4 = LUDACreateArcCurve(radius -thetaRange
                       :0 "Concave")
19             )
20         )
```

```
21          sBendCurve5 = ccCreateCurve("t" "0" 0:straight3L)
22          sBendPolyCurve = LUDAStitchCurves(list(sBendCurve1
                sBendCurve2 sBendCurve3 sBendCurve4 sBendCurve5))
23          return(ccTransformPolyCurve(sBendPolyCurve list(offset 0.0)
                ))
24      )
25  )
26  ;Create top branch of directional coupler
27  SBendLeft = LUDACreateSBend(10.0 10.0 10.0 10.0 45?direction "Down
        ")
28  SBendRight = LUDACreateSBend(10.0 10.0 10.0 10.0 45?direction "Up"
        )
29
30  SBendLeftCurves = foreach(mapcar seg SBendLeft~>segments car(seg))
31  SBendRightCurves = foreach(mapcar seg SBendRight~>segments car(seg
        ))
32
33  DCTopBranch = LUDAStitchCurves(append(SBendLeftCurves
        SBendRightCurves))
34  ccGenFigs(ccCreateLine(cv "waveguide" DCTopBranch))
```

Exercise: Create a procedure that generates a typical directional coupler polycurves.

Exercise: Create a procedure that gets height and length of a S-Bend instead of its arc radius and straight sections' length.

Path Implementation

2

2.1 Paths

ccPath objects are essential building blocks of CurvyCore^{TM} engine to represent photonic waveguides. They are created by offsetting a waveguide's center curve by a positive or negative values. For example, to create a photonic waveguide with $width = 300nm$, you need a curve representing the center of the waveguide and two offset curvers with $offset = 150nm$ and $offset = -150nm$ for the left and right curves respectively as shown in Fig. 2.3. Figures 2.3 and 2.2 show some of the shapes than can be implemented using ccPath objects. In this chapter, we will use the curve generators from the last chapter to create the center of ccPath objects and then creates different photonic waveguides such as SBend and arcBend; In addition, we will discuss how the offset curves' distance be defined parametric so that taper waveguides can be created. Considering that in most of Photonic Integrated Circuits (PICs) fabrication processes, there are multiple mask layers for creating a specific waveguide profile, we will introduce how an offset shape can be created based on a reference ccPath representing the waveguide.

2.2 Implement a Uniform ccPath

Figure 2.4 shows the graphical representation of a typical uniform ccPath object. The ccPath object is uniform because the offset curves have a constant offset distance from the center curve. Most of waveguide routings used in PICs can be represented using a uniform ccPath object. Code 2.1 shows SBend waveguide implementation; as is shown in line 8, `ccCreatePath` accepts a list of curve segments in the format `(curve transform width)`; the width parameter is the width of each curve segment. A user can specify

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_2

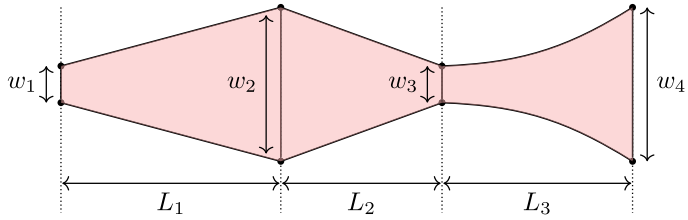


Fig. 2.1 Sample segmented taper shape which can be created by a ccPath object

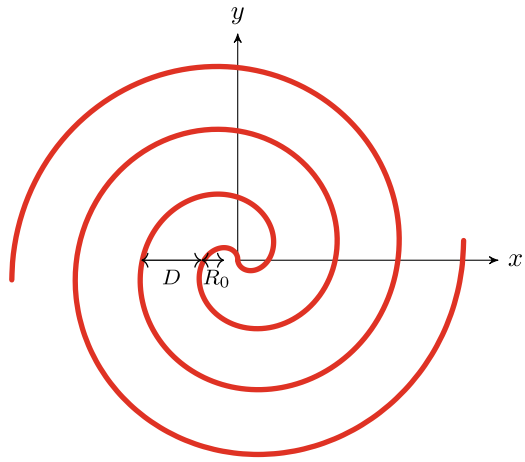


Fig. 2.2 Archimedean spiral used to create optical delay can be implemented as a ccPath object

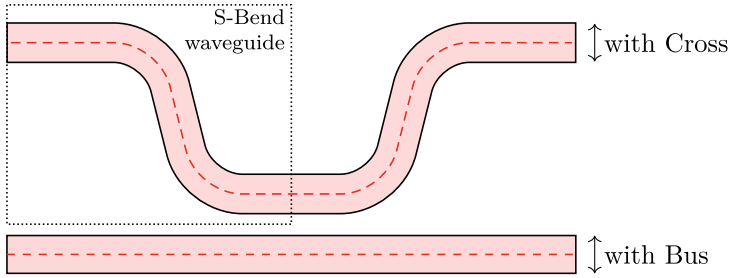


Fig. 2.3 A typical directional coupler implemented using S-Bend and straight ccPath objects

the width for all curve segments using `?defaultWidth` argument. The result of loading Code 2.1 is shown in Fig. 2.5 representing a S-Bend waveguide having 1000nm for its width. To provide a better understanding of ccPath object's implementation, the center of ccPath object is drawn as a ccLine on a separate layer.

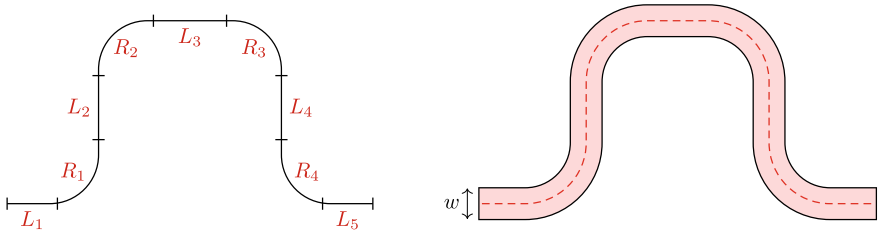


Fig. 2.4 Sample trombone waveguide created using a ccPath object by offsetting its center curve

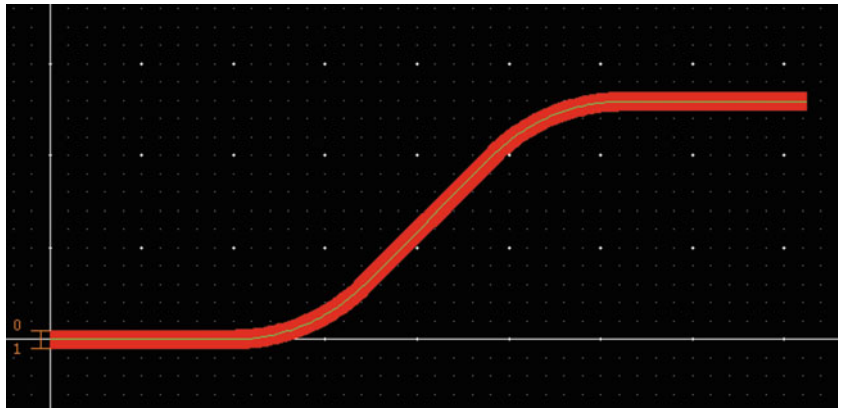


Fig. 2.5 Result of loading Code 2.5 to create an S-Bend waveguide with $width = 1000nm$

Code 2.1 SBend waveguide implementation using ccPath object

```
procedure(LUDACreateSBendWG(cellView lpp width straight1L
    straight2L straight3L radius thetaRange @key
                        (direction "Up")
                        (genFigs t)
                        )
    let((sBendPolyCurve sBendPath)
        sBendPolyCurve = LUDACreateSBend(straight1L straight2L
            straight3L radius thetaRange ?direction direction)
        sBendPath = ccCreatePath(cellView lpp sBendPolyCurve~>
            segments ?defaultWidth width)
        when(genFigs
            ccGenFigs(ccCreateLine(cv "SiNWaveguide" sBendPolyCurve
                ))
            ccGenFigs(sBendPath)
        )
        sBendPath
    )
)

```

```
;Example
cv = geGetEditCellView()
LUDACreateSBendWG(cv list("waveguide" "drawing") 1.0 10.0 10.0 10.
    0 10.0 45.0)
```

Exercise: Using Curve stitching create a trombone shape showed in Fig. 2.4
Exercise: Using curve stitching create a chain of arcBends as shown in Fig. 2.9

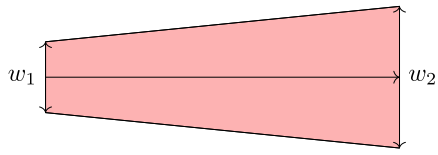
2.3 Implementation of a Non-uniform Path

A waveguide taper is an example of a non-uniform ccPath object; i.e., the offset curve value is not constant along the center curve. As mentioned in the previous section, a path segment can have its own width whose value can be a constant string representing a uniform width or it can be a parametric equation. As an example a linear taper can be specified by a straight section curve whose width increases from w_1 to w_2 along the center curve linearly as shown in Fig. 2.6.

The taper implementation using ccPath object is shown in Code 2.2; and the result of loading the code is shown in Fig. 2.7.

Code 2.2 Linear taper implemented using ccPath object

```
procedure(LUDACreateLinearTaper(cellView lpp widthS widthE taperL
    @key
                              (genFigs t))
    let((taperCenter taperWidth taperPath)
        taperCenter = ccCreateCurve("t" "0" 0:taperL)
        taperWidth = lsprintf("%.12f + t/%.12f*(%.12f-%.12f)"
            widthS taperL widthE widthS)
        taperPath = ccCreatePath(cellView lpp list(list(taperCenter
             list(0:0 0) taperWidth)))
        when(genFigs
            ccGenFigs(taperPath)
        )
        taperPath
    )
)
cv = geGetEditCellView()
LUDACreateLinearTaper(cv list("waveguide" "drawing") 1.0 3.0 10.0)
```



$$x(t) = t; y(t) = 0$$
$$w(t) = w_1 + \frac{t}{L} * (w_2 - w_1)$$
$$0 \leq t \leq L$$

Fig. 2.6 Parametric representation of an arc bend

Fig. 2.7 Result of loading Code 2.2 implementing a linear taper

To expand the features of the taper generation procedure, an exponential taper style is added to Code 2.2 to implement a procedure capable of generating linear and exponential tapers as shown in Code 2.3.

Code 2.3 Taper waveguide supporting linear and exponential taper style

```
1  procedure(LUDACreateTaper(cellView lpp widthS widthE taperL @key (
       genFigs t) (taperType 'linear))
2      let((taperCenter taperWidth taperPath expFactor)
3          taperCenter = ccCreateCurve("t" "0" 0:taperL)
4          caseq(taperType
5              ('linear
6                  taperWidth = lsprintf("%.12f + t/%.12f*(%.12f-%.12f
                       )" widthS taperL widthE widthS)
7              )
8              ('exponential
9                  expFactor = log(widthE-widthS+1)
10                 taperWidth = lsprintf("%.12f + exp(t*%.12f/%.12f)-1
                       " widthS expFactor taperL)
11             )
12         )
13         taperPath = ccCreatePath(cellView lpp list(list(taperCenter
               list(0:0 0) taperWidth)))
14         when(genFigs
15             ccGenFigs(taperPath)
16         )
17         taperPath
18     )
19 )
```

Exercise: Implement a procedure that can generate a series of abutted taper waveguides, as shown in Fig. 2.1, where each taper section is provided as `(widthStart Length widthEnd taperStyle)`. The assumption is that the end's width of each section is the same as the start's width of next section.

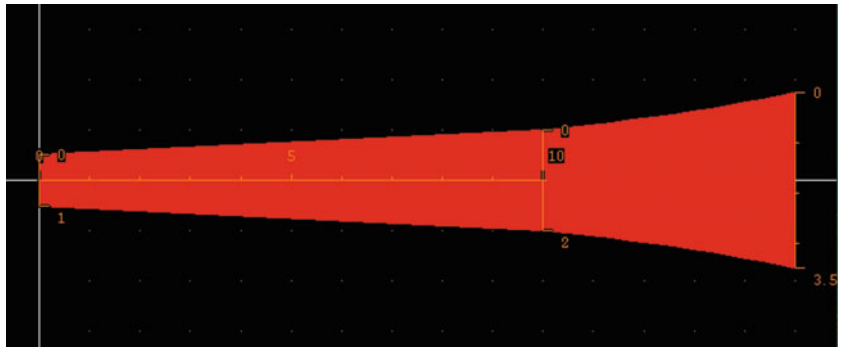
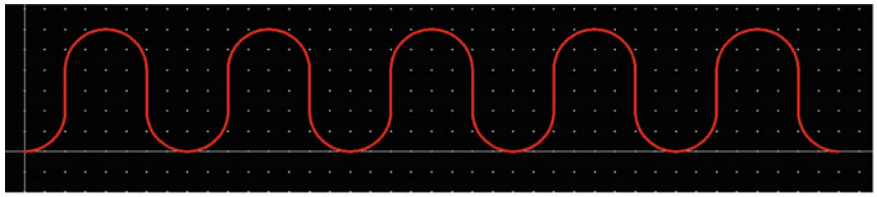


Fig. 2.8 Result of loading Code 2.4 generating multi-segmented taper

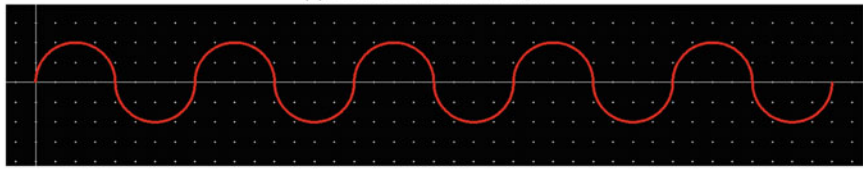
Solution: A basic solution is shown in Code 2.4 and the result of loading the code is shown in Fig. 2.8 generating a two-segment taper whose first taper is linear and the second one is exponential.

Code 2.4 An example of implementing a multi-segmented taper using ccPath

```
procedure(LUDACreateMultiSegTapers(cellView lpp taperList @key (
    genFigs t))
    let((offset taperWidth expFactor widthS widthE taperL
        taperSegment taperSegs taperType taperPath)
        offset = 0.0:0.0
        taperSegs = foreach(mapcar taper taperList
                        widthS = nthelem(1 taper)
                        taperL = nthelem(2 taper)
                        widthE = nthelem(3 taper)
                        taperType = nthelem(4 taper)
                        caseq(taperType
                            ('linear
                                taperWidth = lsprintf("%.12f + t/%.12
                                    f*(%.12f-%.12f)" widthS taperL
                                    widthE widthS)
                            )
                            ('exponential
                                expFactor = log(widthE-widthS+1)
                                taperWidth = lsprintf("%.12f + exp(t
                                    *%.12f/%.12f)-1" widthS expFactor
                                     taperL)
                            )
                        )
                        taperSegment = list(ccCreateCurve("t" "0" 0:
                            taperL) list(offset 0.0) taperWidth)
                        offset = xCoord(offset)+taperL:yCoord(offset
                            )
```



(a) Chain of Trombone shapes



(b) Chain of arcBend shapes

Fig. 2.9 Chain a reference ccPath object multiple times

```
                    taperSegment
                )
        taperPath = ccCreatePath(cellView lpp taperSegs)
        when(genFigs
            ccGenFigs(taperPath)
        )
        taperPath
    )
)
cv = geGetEditCellView()
taperPath = LUDACreateMultiSegTapers(cv list("waveguide" "drawing"
    ) list(list(1.0 10.0 2.0 'linear) list(2.0 5.0 3.5 '
    exponential)))
```

Exercise: Implement a procedure that gets a ccPath object and generate a chain of representing n ccPath objects as is shown in Fig. 2.9 for Trombone and arcBend shapes.

2.4 ccFacet Objects

At the first inspection of generated ccPath shapes, they look like a polygon objects; however, a ccPath shape is not simply a polygon; the start and end edges of a ccPath object are constructed using a special CurvyCore object named ccFacet. ccFacet objects are key elements in seamless connection of PIC components making sure that their edges are seamlessly aligned especially when they are rotated. Figure 2.10 shows what would potentially happen for the edges of two rotated polygons. The figure shows that the upper-left and upper-right vertices of the polygons; the upper-left vertex of red polygon is closer to the upper-right grid point and the upper-right vertex of blue polygon is closer to the lower-left grid point;

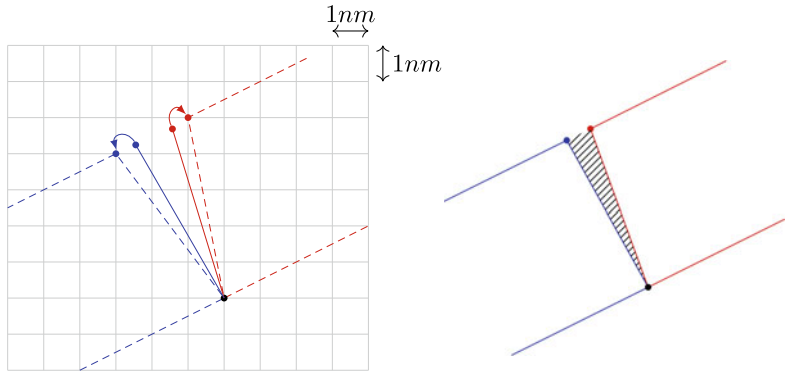


Fig. 2.10 When floating point calculation and snap to rectilinear grid can cause slivers; the figures are exaggerated to convey the underlying problem

this means, when the polygon vertices are adjusted to be on grid, there will be a gap (sliver) between two polygons. This is why we need a special object type, ccFacet, carrying more information so that these edges are created in a way that slivers get eliminated.

As is shown in the following database query, a ccPath object has a `beginFacet` and `endFacet`; and a ccFacet is defined by its center point, left/right offsets, and angle.

```
taperPath~>beginFacet~>?
(objType angle corners name offsets
    radius width xy
)
taperPath~>endFacet~>?
(objType angle corners name offsets
    radius width xy
)
```

Figure 2.11 shows graphical representation of a ccFacet object. It is important to consider the followings while you define a ccFacet object:

- The angle of a facet is measured between the negative y-axis and the perpendicular line to the straight line between right vertex (R) and left vertex (L). Since every straight line has two perpendicular lines, the perpendicular line used for angle definition is the one located on the left side when you are traversing from R to L
- the offset value for left and right edges can be positive and/or negative. Positive means the edge vertex will be located at the half plane $y \geq 0$; and negative means the vertex will be located at at the half plane $y \leq 0$. These half planes are based on the origin of the facet

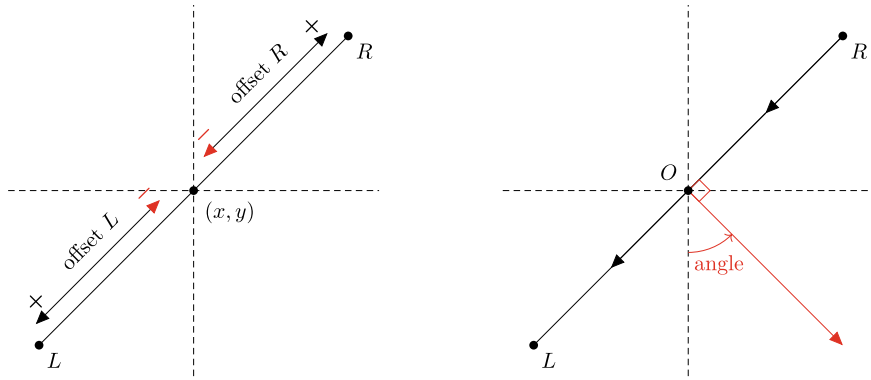


Fig. 2.11 How ccFacet object is defined by its origin O, left and right edges' offset, and angle. The left shape shows how the sign of offset (R/L) can change the location of its associated vertex for a facet with $angle = 45°$

Consider the following guidelines when implementing a facet object:

- Facets are used in the interface between two PIC components; e.g., a waveguide has an input where the optical mode is injected and an output where the optical mode leaves the waveguide; or a MMI1x2 has an input facet and two output facets.
- If two facets needs to be aligned, the origin of their facet should be coincident and the facets should be define consistently with respect to left/right offset values.

To get a graphical representation of a facet, you can create a ccLine object from it and then use `ccGenFigs` to generate it in layout as shown in Code 2.5 whose result is shown in Fig. 2.12.

Exercise: Can you specify facet1 and facet2 in Fig. 2.12; also specify the direction of potential waveguide attached to each facet.

Code 2.5 Examples of creating a ccFacet object

```
cv = geGetEditCellView()
facet1 = ccCreateFacet(5:0 ?angle 90.0 ?offsets -5:10)
facet2 = ccCreateFacet(5:0 ?angle 270.0 ?offsets -5:10)
ccGenFigs(ccCreateLine(cv "waveguide" list(facet1)))
ccGenFigs(ccCreateLine(cv "waveguide" list(facet2)))
```

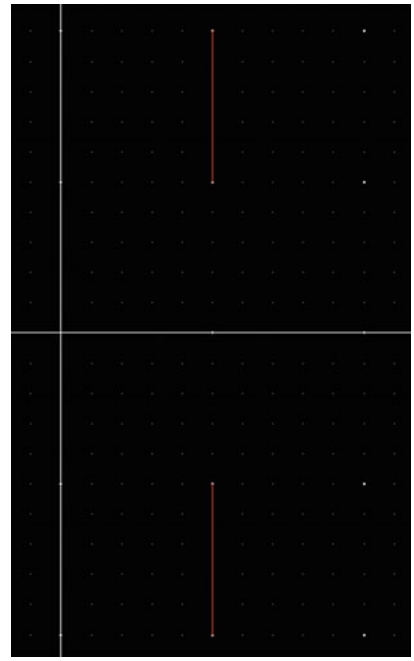


Fig. 2.12 Result of loading Code 2.5 generating examples of ccFacet object

2.5 Preserve Facet Objects

In general you define facets at the periphery of your component to make sure that it would align seamlessly with another component. Figure 2.13 shows an example of a taper having three facets at the start and three facets are the end. The shape similar to this taper can be generated in two ways: (1) generated the red and gray shapes separately, red shape can be creates using ccPath object; however, grays shapes cannot be created as a ccPath objects because of being asymmetrical; we will show how gray shapes can be created using ccSurface object in next chapter. (2) we can create a blanket gray shape and then use boolean operation to create two gray wings shapes.

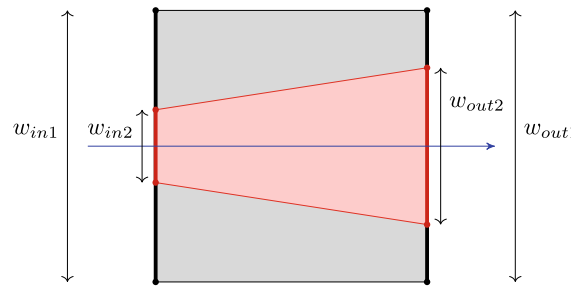


Fig. 2.13 A linear taper whose start and end edges are represented by three ccFacet objects drawn thicker

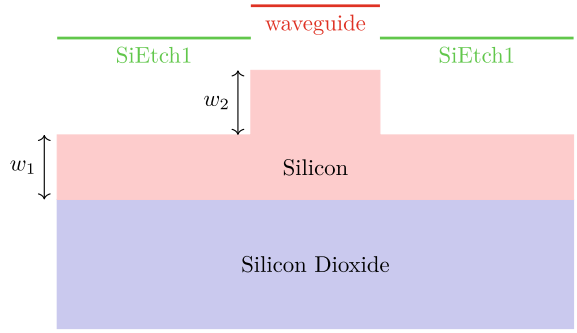


Fig. 2.14 Typical representation of an rib waveguide. SiEtch1 layer specifies where the silicon needs to be etched to create rib structure

There are use cases where you need to perform some boolean operations to reach your final shape and it is important to make sure that the boolean operation doesn't change the nature of cell shapes' facets to an ordinary ccCurve object. Assume that you want to create a rib waveguide whose cross section is shown in Fig. 2.14. You can create the "waveguide" layer first and then apply `AndNot` operation between a blanket layer representing the boundary of the waveguide and "waveguide" layer to generate "SiEtch1" layer.

Code 2.6 shows the flow of generating a taper based on rib waveguide shown in Fig. 2.14. `taperPath` is a segmented taper which was discussed and implemented in the previous section. It has a linear and exponential taper sections. `box` is the blanket layer representing the waveguide envelope. In line 4, the boolean operation is applied to generate the "SiEtch1" layer; because the result of boolean operations is a list, `foreach` is used to generate final shapes shown in Fig. 2.15. Since `box` is a ccPath object it has two facets at its start and end, the `?facetsToCurves nil` insures that int the final shapes the input and output facets are preserved; if you set it as `t`, then the facets will be converted to ordinary ccCurve objects for the boolean operation.

Guideline: If you are working with waveguides including tapers, always preserves the facets so that the connection to the next element be seamless and there is not gap.

Fig. 2.15 Result of creating a rib taper based on waveguide cross section shown in Fig. 2.14 and preserving facets

Code 2.6 Examples of creating a ccFacet object

```
cv = geGetEditCellView()
taperPath = LUDACreateMultiSegTapers(cv list("waveguide" "drawing"
    ) list(list(1.0 10.0 2.0 'linear) list(2.0 5.0 3.5 '
    exponential)))
box = ccCreatePath(cv list("SiEtch1" "drawing") list(ccCreateCurve
    ("t" "0" 0:15)) ?defaultWidth 6.0)
ribSection = ccLayerAndNot(box taperPath ?facetsToCurves nil)
foreach(shape ribSection ccGenFigs(shape))
```

2.6 Generate General Waveguide Shapes Using `ccGenOffsetFig`

As shown in Fig. 2.14, a typical waveguide needs more than a layer for its implementations. In most of the cases there is a main layer representing the waveguide which can be a non-mask layer, and the other mask layers are created based on that. This means the derived mask layer shapes should be generated from the master reference shape with some offset value. Figure 2.16 shows how a typical mask layer can be derived from its associated reference layer. Since in most of the cases the reference shape for a specific waveguide is based on ccPath; CurvyCore APIs provide a `ccGenOffsetFig` that gets a ccPath object and creates a derived shape based on its left and/or right edges. The derived shape will creates an relevant facets for the start/end of the derived shape based on the start/end facets of reference ccPath shape.

Code 2.7 shows how an SBend waveguide, based on the cross-section in Fig. 2.14, can be implemented. Figure 2.17 show the result of the implementation.

Code 2.7 Creating an SBend with its derived shapes based on cross-section shown in Fig. 2.14

```
cv = geGetEditCellView()
SBendPath = LUDACreateSBendWG(cv list("waveguide" "drawing") 1.0 1
    0.0 10.0 10.0 10.0 45.0)
```

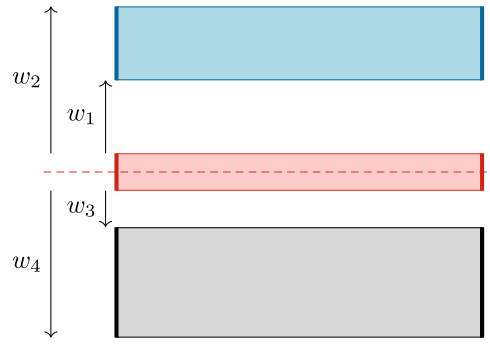


Fig. 2.16 How derived shapes can be defined based on the main waveguide layer. The reference waveguide layer is used to generate derived layers and their associated facets drawn by a thicker line

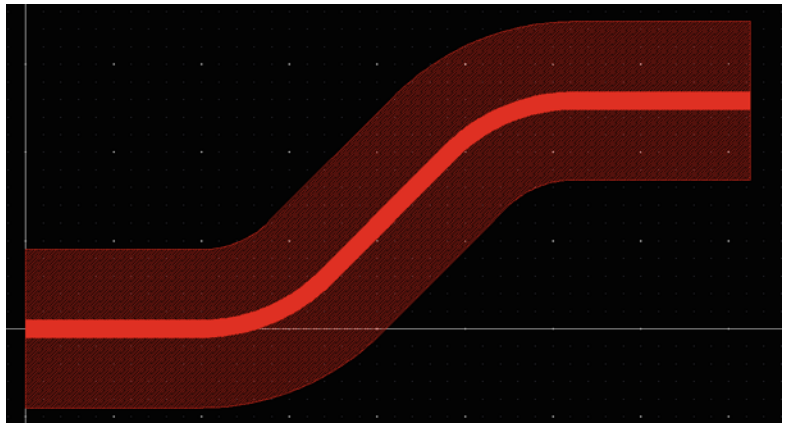


Fig. 2.17 Result of implementing an SBend waveguide based on Fig. 2.14 cross-section

```
3 topSBendWing = ccGenOffsetFig(SBendPath list("SiEtch1" "drawing")
      ?side "left" ?inner 0.0 ?outer 4.0)
4 bottomSBendWing = ccGenOffsetFig(SBendPath list("SiEtch1" "drawing
      ") ?side "right" ?inner 0.0 ?outer 4.0)
5 foreach(shape list(topSBendWing bottomSBendWing) ccGenFigs(shape))
```

2.6.1 Waveguide Template

In most of photonic processes, there are limited number of cross-sections used for waveguides. Since each of the cross-sections can be associated with a specific waveguide type. This means a waveguide type can be represented by a template defined by a data structure in the code. A data structure that can be used to represent a waveguide template is a hash table. Code 2.8 shows how Fig. 2.16 can be represented as a hash table that can be passed as an input argument to waveguide generation functions. As is shown in line 3, each derived shape is represented as a disembodied property list (DPL) where `leftShapeOffset` defines the left derived shape, based on the left edge of reference ccPath object; and `rightShapeOffset` defines the right derived shape, based on the right edge of reference ccPath object. In both cases the offset value is defined `a:b` where `a` is the inner offset and `b` is the outer offset.

Code 2.8 Waveguide template implemented using hash table data structure for Fig. 2.16 with $w_1 = 0$ $w_3 = 0$ $w_2 = 4$ and $w_4 = 2$

```
1 cv = geGetEditCellView()
2 WGTemplateTable = makeTable('WGTemplateTable nil)
3 WGTemplateTable["WGTemplate1"] = list(
4                                 list(nil 'lpp "SiEtch1" 'shapeType "offset
                                      " 'leftShapeOffset 0.0:4.0 '
                                      rightShapeOffset 0.0:2.0)
```

```
                                    )

procedure(LUDACreateSBendWG(cellView lpp width straight1L straight2L straight3L radius thetaRange @key
                        (wgTemplate nil)
                        (direction "Up")
                        (genFigs t)
                        )
    let((sBendPolyCurve sBendPath derivedShapes currentDerivedShape)
        sBendPolyCurve = LUDACreateSBend(straight1L straight2L straight3L radius thetaRange
                                    ?direction direction)
        sBendPath = ccCreatePath(cellView lpp sBendPolyCurve~>segments ? defaultWidth width)
        when(wgTemplate
            derivedShapes = foreach(mapcon derivedShape WGTemplateTable[wgTemplate]
                                currentDerivedShape = car(derivedShape)
                                case(currentDerivedShape->shapeType
                                    ("offset"
                                        derivedShapeL = ccGenOffsetFig(sBendPath currentDerivedShape->lpp ?side "left"
                                                                ?inner car(currentDerivedShape->leftShapeOffset)
                                                                ?outer cadr(currentDerivedShape->leftShapeOffset))
                                        derivedShapeR = ccGenOffsetFig(sBendPath currentDerivedShape->lpp ?side "right"
                                                                ?inner car(currentDerivedShape->rightShapeOffset)
                                                                ?outer cadr(currentDerivedShape->rightShapeOffset)
                                                                )
                                        list(derivedShapeL derivedShapeR)
                                    )
                                )
                            )
        )
        when(genFigs
            ccGenFigs(ccCreateLine(cv "y0" sBendPolyCurve))
            ccGenFigs(sBendPath)
            foreach(shape derivedShapes ccGenFigs(shape))
        )
        sBendPath
    )
)

```

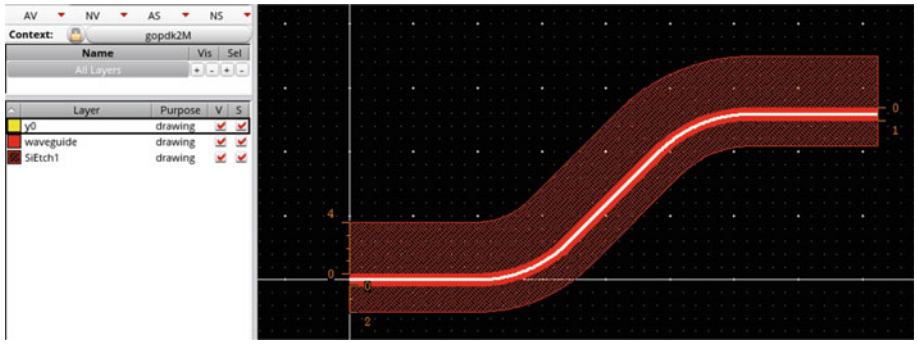


Fig. 2.18 SBend waveguide generated by Code 2.8 based on Fig. 2.16

```
41 SBendPath = LUDACreateSBendWG(cv list("waveguide" "drawing") 1.0 10.0 10.0
       10.0 10.0 45.0 ?wgTemplate "WGTemplate1")
```

Line 41 calls the procedure to generate a SBend waveguide based on "WGTemplate1", and the result is shown in Fig. 2.18.

Exercise: Add a new derived shape type, named "enclosure" where the derived layer will be drawn as a blanket layer based on the centerline of reference ccPath object. Use `ccGetPathCenterLine` function to get the center line and have a property `'enclosureVal` as part of the DPL defining the width of the blanket layer.

Exercise: Update Code 2.8 so that it can generate the newly added "enclosure" derived layer.

`Solution:` The solution is provided in Code 2.9.

Code 2.9 Waveguide template implemented using hash table data structure for Fig. 2.16 with additional "enclosure" derived layer (new added lines are shown with a red arrow)

```
1  cv = geGetEditCellView()
2  WGTemplateTable = makeTable('WGTemplateTable nil)
3  WGTemplateTable["WGTemplate1"] = list(
4                                  list(nil 'lpp "SiEtch1" 'shapeType "offset
                                       " 'leftShapeOffset 0.0:4.0 '
                                       rightShapeOffset 0.0:2.0)
5                                ⟹list(nil 'lpp "SiEtch0" 'shapeType "
                                    enclosure" 'enclosureVal 9.0)
6                                  )
7
8  procedure(LUDACreateSBendWG(cellView lpp width straight1L straight2L straight
       3L radius thetaRange @key
9                  (wgTemplate nil)
10                 (direction "Up")
11                 (genFigs t)
12                 )
13     let((sBendPolyCurve sBendPath derivedShapes currentDerivedShape)
14         sBendPolyCurve = LUDACreateSBend(straight1L straight2L straight3L
               radius thetaRange
```

```
15                          ?direction direction)
16          sBendPath = ccCreatePath(cellView lpp sBendPolyCurve~>segments ?
                defaultWidth width)
17          when(wgTemplate
18            derivedShapes = foreach(mapcon derivedShape WGTemplateTable[
                  wgTemplate]
19                  currentDerivedShape = car(derivedShape)
20                  case(currentDerivedShape->shapeType
21                    ("offset"
22                      derivedShapeL = ccGenOffsetFig(sBendPath
                            currentDerivedShape->lpp ?side "left"
23                              ?inner car(currentDerivedShape->leftShapeOffset
                                    )
24                              ?outer cadr(currentDerivedShape->
                                    leftShapeOffset))
25                      derivedShapeR = ccGenOffsetFig(sBendPath
                            currentDerivedShape->lpp ?side "right"
26                              ?inner car(currentDerivedShape->
                                    rightShapeOffset)
27                              ?outer cadr(currentDerivedShape->
                                    rightShapeOffset))
28                      list(derivedShapeL derivedShapeR)
29                  )
30                ⟹("enclosure"
31                ⟹ sBendPathCenter = ccGetPathCenterLine(sBendPath)
32                ⟹ derivedShapeEnc= ccCreatePath(cellView currentDerivedShape
                      ->lpp sBendPathCenter~>segments
33                              ?defaultWidth currentDerivedShape->
                                    enclosureVal)
34                ⟹ list(derivedShapeEnc)
35                 )
36               )
37             )
38          )
39          when(genFigs
40            ccGenFigs(ccCreateLine(cv "y0" sBendPolyCurve))
41            ccGenFigs(sBendPath)
42            foreach(shape derivedShapes ccGenFigs(shape))
43        )
44        sBendPath
45      )
46  )
47
48  SBendPath = LUDACreateSBendWG(cv list("waveguide" "drawing") 1.0 10.0 10.0
        10.0 10.0 45.0 ?wgTemplate "WGTemplate1")
```

Surface Implementation

3

3.1 Surfaces

In CurvyCore® implementation, ccSurface object is defined as a closed boundary; i.e., before creating a ccSurface object, its associated boundary needs to be implemented using a ccPolyCurve object. The boundary needs to be continuous but its first-order derivative doesn't necessarily need to be continuous. When a surface is created, one of its database properties would be its area which will be positive if the direction of the ccPolycurve representing ccSurface boundary is counter-clockwise (positively) oriented; and negatively oriented otherwise. Positively oriented surfaces have their interior on the left while traveling along their boundary; and negatively oriented surfaces have their exterior on the left while traveling their boundary. Negatively oriented surfaces are considered holes in CurvyCore®. Figure 3.1 shows how a ring shape can be created using ccSurface object defined by C_1 with a hole represented by C_2.

3.2 ccSurface Use Cases

3.2.1 Donut/Ring Shape

In this section, some use cases of ccSurface object will be discussed. Code 3.1 shows how a donut surface shape can be created based on its outer and inner boundaries which are closed ccPolyCurve objects representing the edges of a surface. As is shown in line 20, the `donutSurface` is generated based on `surfaceBoundary` as its outer boundary and `surfaceHoleCW` as its inner boundary also known as a hole. `ccCreateSurface` accepts a surface's holes as an input argument of type list. Notice how `ccReversePolyCurve` is used in line 19 to reverse the polycurve representing the

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_3

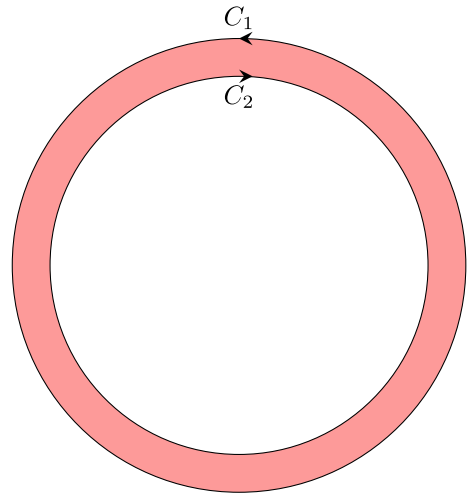


Fig. 3.1 A ring shape can be implemented as a surface C_1 with a hole represented by C_2

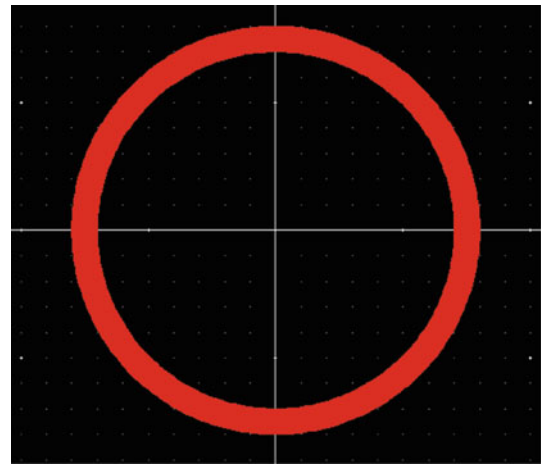


Fig. 3.2 Result of loading Code 3.1 to create donut surface

inner boundary from a positively oriented one to a negatively oriented one so that it can be used as a hole. The result of calling `LUDACreateDonut` is shown in Fig. 3.2.

Code 3.1 Implement a procedure to generate a donut shape

```
1 procedure(LUDACreateDonut(cv lpp radiusIn radiusOut @key
2                           (genFigs t)
3                           )
4     prog((mathConstants PI outerCircleX outerCircleY outerCircle
          innerCircleX innerCircleY innerCircle surfaceHoleCW surfaceHoleCCW
           surfaceBoundary donutSurface)
5         unless(radiusIn < radiusOut
6             warn("Inner radius should be less than the outer radius")
7             return(nil)
```

```
        )
        defMathConstants('mathConstants)
        PI = mathConstants.PI
        outerCircleX = lsprintf("%.12f*cos(t)" radiusOut)
        outerCircleY = lsprintf("%.12f*sin(t)" radiusOut)
        outerCircle = ccCreateCurve(outerCircleX outerCircleY 0:2*PI)
        surfaceBoundary = ccCreatePolyCurve(list(outerCircle) ?close t)
        innerCircleX = lsprintf("%.12f*cos(t)" radiusIn)
        innerCircleY = lsprintf("%.12f*sin(t)" radiusIn)
        innerCircle = ccCreateCurve(innerCircleX innerCircleY 0:2*PI)
        surfaceHoleCCW = ccCreatePolyCurve(list(innerCircle) ?close t)
        surfaceHoleCW = ccReversePolyCurve(surfaceHoleCCW)
        donutSurface = ccCreateSurface(cv lpp surfaceBoundary list(
            surfaceHoleCW))
        when(genFigs
            ccGenFigs(donutSurface)
        )
        return(donutSurface)
    )
)
cv = geGetEditCellView()
LUDACreateDonut(cv list("waveguide" "drawing") 7.0 8.0)
```

3.2.2 Pie Shape

Code 3.2 shows implementation of a pie generation procedure. Two instances of similar pie shape with `rotation=0.0` and `rotation=135.0` is shown in Fig. 3.3.

General Guideline: If a shape needs to be rotated, in most cases, it would be much easier and cleaner to implement the reference shape with `offset = 0 and rotation = 0` and then use `ccMoveFig` to apply the offset and rotation. It should be mentioned that when both `rotation and offset` are specified, the rotation gets applied before the offset.

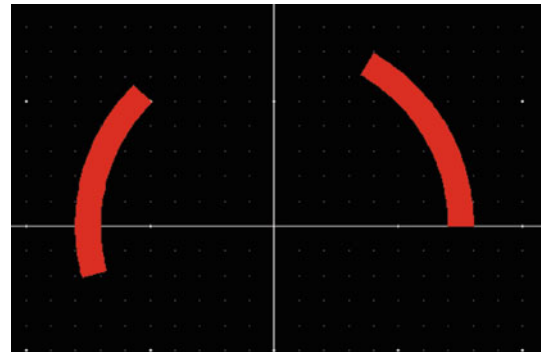


Fig. 3.3 Tow similar instances of a pie shape with `rotation = 0.0 and 135.0`

Exercise: Try to evaluate the order of rotation and offset when both are specified in `LUDACreatePie` procedure.

Code 3.2 Implement a procedure to generate a pie shape

```
procedure(LUDACreatePie(cv lpp rin rout angle @key
                        (genFigs t)
                        (offset 0:0)
                        (rotation 0.0)
                        )
    prog((mathConstants PI line1 line2 ccOuterArc ccInnerArc arcBoundary
         arcSurface)
        defMathConstants('mathConstants)
        PI = mathConstants.PI
        line1 = ccCreateCurve("t" "0.0" rin:rout)
        ccOuterArc = ccCreateCurve(lsprintf("%.12f*cos(t)" rout) lsprintf("
            %.12f*sin(t)" rout) 0:angle/180.0*PI)
        ccInnerArc = ccCreateCurve(lsprintf("%.12f*cos(t)" rin) lsprintf("
            %.12f*sin(t)" rin) angle/180.0*PI:0)
        line2 = ccCreateLineSegment(ccGetCurvePoint(ccOuterArc cadr(
            ccOuterArc~>range))
                    ccGetCurvePoint(ccInnerArc car(ccInnerArc~>range)))
        arcBoundary = ccCreatePolyCurve(list(line1 ccOuterArc line2
            ccInnerArc) ?close t ?smoothness 'G0)
        arcSurface = ccCreateSurface(cv lpp arcBoundary)
        ccMoveFig(arcSurface ?transform list(offset rotation))
        when(genFigs
            ccGenFigs(arcSurface)
        )
        return(arcSurface)
    )
)

cv = geGetEditCellView()
LUDACreatePie(cv list("waveguide" "drawing") 7.0 8.0 60.0)
LUDACreatePie(cv list("waveguide" "drawing") 7.0 8.0 60.0 ?rotation 135.0)
```

3.3 Boolean Operation

`ccLayerOr, ccLayerAnd, ccLayerAndNot` are the main APIs used to perform boolean operation in the mathematical domain; the result of these boolean operations would be a list of ccSurface objects or nil. Assume you want to create a ring modulator having heater on top of the ring waveguide and also there is a design rule that a specific layer needs to enclose the heater layer by a certain value; the sketch of this example is shown in Fig. 3.4. Code 3.3 shows how ccBoolean operations can be used in each stage to generated different sections of the heater. `ccCreateEllipse` is used to create a circle surface representing the inner circle of the ring; `ccLayeAndNot` operation of this shape and extended straight waveguides, define by `heaterPathTop` and `heaterPathBot` gener-

(a) A simplified ring modulator with a heater

(b) Apply `ccLayerAnd` as shown in Code 3.3

(c) Apply `ccLayerAndNot` as shown in Code 3.3

Fig. 3.4 How boolean operation used to generate ring modulator shapes

ates the wings of the heater as shown in Fig. 3.4c. `ccLayerAnd` of heater's bounding box with the ring shape generates the arc sections of the heater as shown in Fig. 3.4b. Eventually `ccLayerOr` between the heater wings and arc sections will generate the final heater shapes.

Code 3.3 Boolean operation used to generated shapes in Fig. 3.4

```
1 donutSurface = LUDACreateDonut(cv list("waveguide" "drawing") 7.0 8.0)
2 heaterPathTop = ccCreatePath(cv "waveguide" list(ccCreateCurve("t" "3.0" -
      12:12)) ?defaultWidth 1.0)
3 heaterPathBot = ccCreatePath(cv "waveguide" list(ccCreateCurve("t" "-3.0"
      -12:12)) ?defaultWidth 1.0)
4 innerCircle= ccCreateEllipse(cv "waveguide" -7:-7 7:7 ?genFigs nil)
5 heaterShapesA = ccLayerAndNot(list(path1 path2) innerCircle ?lpp "Metal0")
6 heaterShapesB = ccLayerAnd(donutSurface ccCreateRect(cv "waveguide" -12:-3
       12:3 ?genFigs nil) ?lpp "Metal0")
7 heaterShapes = ccLayerOr(shapes1 shapes2)
8 foreach(shape heaterShapes ccGenFigs(shape))
9 foreach(shape ccLayerOffset(heaterShapes 1.0 ?lpp "Metal1") ccGenFigs(
      shape))
```

As mentioned before, hypothetically, a specific layer needs to enclose the heater layer; this means that the heater layer needs to be offsetted to generated this layer. There is already an API for this purpose, `ccLayerOffset`. This API gets a list of shapes, merges them, and then applies the offset to the merged shape. This API also removes cusps that can be created in the offset process as is shown in Fig. 3.5. It is important to notice that the left and right edges of the heater shape haven't been offsetted, this is because the heater's straight sections are created by ccPath objects causing those edges to be ccFacet objects. In general, an offset operation will not be applied to ccFacet objects. The final results of Code 3.3 is shown in Fig. 3.6. In Code 3.3 the result of `ccLayerOr` is a list and that is reason of using `foreach`; `ccLayerOffset` can be replaces with sequence of operations as described in the following.

- Use `ccLayerOr(heaterShapes nil)` to remove potential cusps and clean the boundary of the surface
- Use `ccOffsetPolyCurve` to offset the boundary of the surface, this function return a `ccPolyCurve` object
- Use the offsetted polycurve to create a new `ccSurface` representing the offsetted version of the original surface

Exercise: Try to follow the alternative solution proposed above to offset a surface. `ccOffsetPolyCurve` provides a mechanism where you can debug the offsetted surface to see cusps, try to use that to see the cusps caused by the offset process as shown in Fig. 3.5.

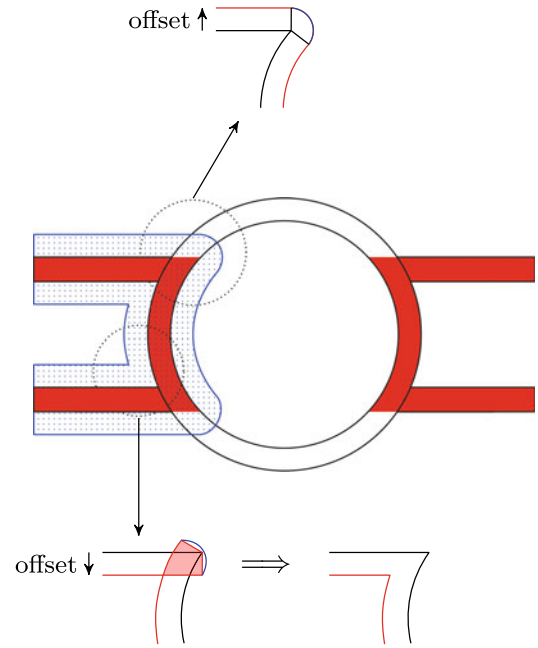


Fig. 3.5 `ccLayerOffset` is used to remove the cusps shown in the bottom shape and fill the gap between the offsetted curves with a circular arc

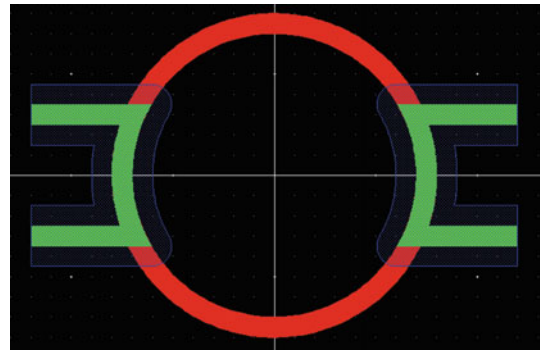


Fig. 3.6 Final result of Code 3.3 to generate a simplified ring with a heater

Exercise: Try to remove the ccFacet object in the boolean operation by passing proper value to `facetToCurves` and evaluate what would be the final heater shape. Justify which approach would be the right one if the heater shape needs to be enclosed inside another layer.

3.4 Asymmetric Taper

As mentioned in Chap. 2, ccPath object can be used to generate symmetrical taper structures; however, creating an asymmetric taper using a ccPath needs boolean operation and would create a code which is difficult to read an maintain. A preferable solution to generate an assymetric taper shown in Fig. 3.7 is to use a ccSurface object.

Code 3.4 shows how an asymmetric taper can be implemented using a ccSurface object. Make sure that you understand how start and end facets are created in lines 9 and 11 respectively. The result of calling `LUDACreateAssymTaper` is shown in Fig. 3.8.

Exercise: Annotate right and left corners of the start and end facets in Fig. 3.8; also specify the $t \geq 0$ and facet's perpendicular line.

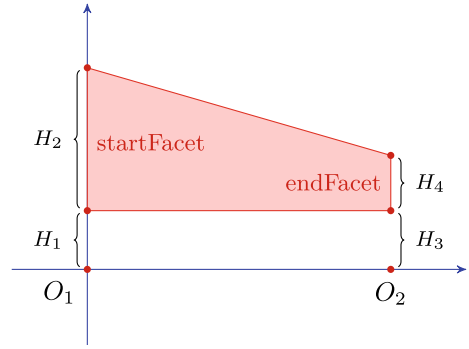


Fig. 3.7 Asymmetric tapers can be preferably created using ccSurface objects

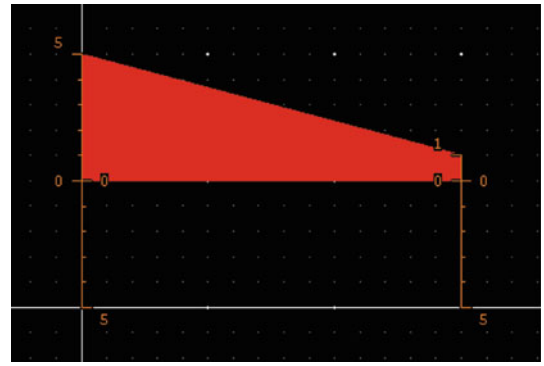


Fig. 3.8 Calling `LUDACreateAssymTaper`, Code 3.4, to generate an assymetric aper using a ccSurface object

Code 3.4 Implement a procedure to generate an asymmetric taper using a ccSurface object

```
procedure(LUDACreateAssymTaper(cv lpp H1 H2 H3 H4 L @key
                      (genFigs t)
                      (offset 0:0)
                      (rotation 0.0)
                      )
    prog((mathConstants PI line1 line2 startFacet endFacet taperBoundary
        taperSurface)
        defMathConstants('mathConstants)
        PI = mathConstants.PI
        startFacet = ccCreateFacet(0:0 ?angle 90.0 ?offsets H1+H2:-H1 ?name
            "startFacet")
        line1 = ccCreateCurve("t" lsprintf("%.12f" H1) 0:L)
        endFacet = ccCreateFacet(L:0 ?angle 270.0 ?offsets -H3:H3+H4 ?name
            "endFacet")
        line2 = ccCreateLineSegment(L:H3+H4 0:H1+H2)
        taperBoundary = ccCreatePolyCurve(list(startFacet line1 endFacet
            line2) ?close t)
        taperSurface = ccCreateSurface(cv lpp taperBoundary)
        when(genFigs
            ccGenFigs(taperSurface)
        )
        return(taperSurface)
    )
)
cv = geGetEditCellView()
LUDACreateAssymTaper(cv "waveguide" 5.0 5.0 5.0 1.0 15.0)
```

Parameterized Cells (PCells)

4

4.1 Introduction

Parameterized cells (PCell), which also known as programmable cells, are groups of cells which have significant commonality in their implementations. There may be some differences such as number of inputs, width/length of shapes used for their construction. There groups of cells can be defined and implemented by their master, set of parameters describing the difference in behavior between variants, and the code that implemented the master. As an example an S-Bend can be defined as a PCell with parameters describing the bend radius, height, waveguide width, etc; and with the code relating the parameters to actual implementation. Although it is very common to have a PCell based on physical design of a component; the logical design, schematic and/or symbol, representation of a component can be parameterized. In this chapter, we use an S-Bend to examine a PCell implementation and also describe some related topics such as public APIs used for waveguide generation and component description format (CDF).

4.2 PCell Implementation Structure

Code 4.1 shows the implementation of the a typical SBend. As is shown, a PCell implementation is divided into three sections:

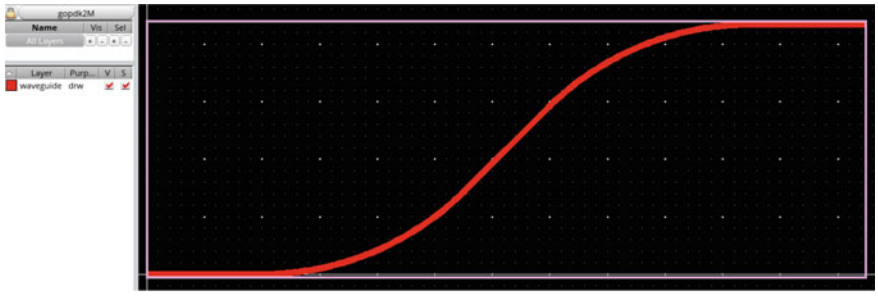
- Cell identifier representing library ID, cell name, view name, and view type. View type is optional and the default would be "maskLayout"
- PCell parameter names, types, and their default value. It would be a good practice to define PCell parameters' type as a string type

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_4

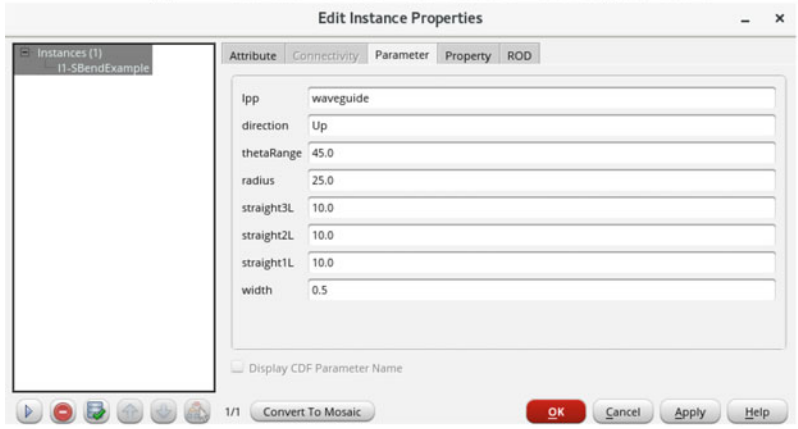
- Implementation code that is responsible to generate the required geometries and other PCell features

In Chap. 2, a utility function, `LUDACreateSBendWG` was implemented to generate a S-Bend waveguide, this utility function is used for the implementation of the PCell. In `pcDefinePCell` procedure context, `pcCellView` refers to the PCell cellview. Using string as a parameter type has many advantages; for example, a user would be able to input a parameter value using scientific notation and `cdfParseFloatString` would convert it to the right floating point value; or when a parameter needs to accept multiple values, they can be passed as a single string and then the string can be parsed to extract parameters.

Figure 4.1 shows an instance of SBend PCell based on its default parameters and the instance property editor form.



(a) An instance of S-Bend PCell based on default parameters



(b) Instance propery editor form

Fig. 4.1 S-Bend Parameterized Cell

Code 4.1 S-Bend PCell implementation

```
pcDefinePCell(
    list(ddGetObj("Training") "SBendExample" "layout")
    list(
        ;Here you define PCell parameter names, types, and default values
        (lpp "string" "waveguide" )
        (width "string" "0.5" )
        (straight1L "string" "10.0" )
        (straight2L "string" "10.0" )
        (straight3L "string" "10.0" )
        (radius "string" "25.0" )
        (thetaRange "string" "45.0" )
        (direction "string" "Up" )
    )
    ;Here is the implementation section where the goemtries will be
        generated
    let(()
        LUDACreateSBendWG(
            pcCellView
            cdfParseFloatString(lpp)
            cdfParseFloatString(width)
            cdfParseFloatString(straight1L)
            cdfParseFloatString(straight2L)
            cdfParseFloatString(straight3L)
            cdfParseFloatString(radius)
            cdfParseFloatString(thetaRange)
            ?direction cdfParseFloatString(direction)
        )
    )
)
```

Most of PCells are implemented for physical design of a cell also known as layout; however, the concept can be applied for other views such as schematic and symbol. Figure 4.2 shows an example of a parameterized symbol view; the PCell parameter is the magnification factor of the symbol's artwork. This example also demonstrates that CurvyCore™ APIs can be used to create a smoother symbol artwork.

Code 4.2 shows how a parameterized S-Bend symbol is implemented. `sBendArtWork` is the `ccPath` representation of SBend artwork. It should be mentioned that when a `ccPath`

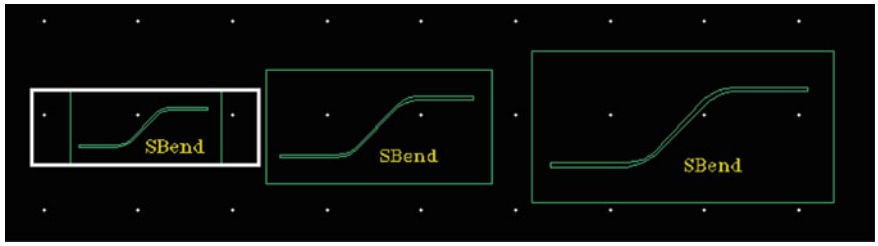


Fig. 4.2 SBend symbol instantiation with different magnification factor

object is generated by `ccGenFigs`, its shapes can be accessed using its `figs` database attribute. `mapcar` is used to scale polygon points representing the SBend and recreate the scaled version of polygon using `dbCreatePolygon`. At the end the original shape needs to be removed using `ccRemoveFigs`. There are some common implementation patterns that needs to be considered while implementing a cell's symbol view:

- Typically the artwork of a symbol is drawn in a box on "device" layer.
- The "instance drawing" layer is used to specify the selection box of an instance when it is instantiated in schematic, as shown in Fig. 4.2. Although it is not visible in a symbol's instance, it can be seen in the master cell view.
- As you can see in Fig. 4.2, the "instance" layer encloses "device" layer. This enclosure needs to be there because typically optical ports will be added between the "device" and "instance" layers. We will show how this would be done later in this chapter.
- It is very important to define the origin of the symbol properly. When a symbol is being instantieted, the origin of a symbol, $(x, y) = (0, 0)$, defines where the mouse pointer is pointing to; so if it is not set properly, it would cause inefficiencies in schematic capture flow.
- In a PIC flow, it would be beneficial, most of the time, to define the origin of a symbol to be at the cell's optical input port.

Code 4.2 Parameterized S-Bend symbol implementation

```
pcDefinePCell(
    list(ddGetObj("Training") "SBendExample" "symbol" "schematicSymbol")
    list(
        (magnification 1.0)
    )
    let((sBendArtwork sBendArtWorkFig sBendArtWorkMag)
        dbCreateRect(pcCellView "device" list(0.2*magnification:-0.2*
            magnification 1.0*magnification:0.2*magnification))
        dbCreateRect(pcCellView "instance" list(0:-0.2*magnification 1
            .2*magnification:0.2*magnification))
        sBendArtwork = LUDACreateSBendWG(pcCellView "device" 0.01 0.2
            0.2 0.2 0.1 45.0 ?offset 0.25:-0.1)
        sBendArtWorkFig = car(sBendArtwork~>figs)
        sBendArtWorkMag = foreach(mapcar pt sBendArtWorkFig~>points
            xCoord(pt)*magnification:yCoord(pt)*magnification
        )
        dbCreatePolygon(pcCellView "device" sBendArtWorkMag)
        ccRemoveFigs(sBendArtwork)
        dbCreateLabel(pcCellView list("instance" "label") 0.6*
            magnification:-0.1*magnification "SBend" "centerLeft" "R0" "
            roman" 0.0625)
    )
)
```

4.3 Enable Connectivity and Hierarchical Design

Connectivity-aware and hierarchical designs are based on a few concepts: Net, Terminal, Instance Terminals, and Pins. In this section, these concepts are explained in their simple forms without going into their details; however, this would be enough for most of the end users of design automation tools. Figure 4.3 demonstrates how different connectivity objects are related.

Nets are used to connect instances in the current level of design hierarchy to which they belong. Although nets are represented as metal wires in the physical domain, there is no concept of optical wires. In a PIC design, optical wires are represented as photonic waveguides modelled as devices. If you want to compare it with electronics, photonic waveguides are equivalent to on-chip transmission lines. Terminals, also know as ports, are logical objects enabling connection to upper level of hierarchy where an instance of the design master gets instantiated. It is important to recognize that a terminal belongs to the master of a design not its instance. Instance terminal are connection points on the instance; these objects provide a connection point from an upper level of hierarchy where the design master is instantiated to their associated terminals in lower level of hierarchy. Essentially terminals and instance terminals are main elements enabling a hierarchical design.

As mentioned both terminals and instance terminals are logical object, pins are physical representation of terminals. Pins are represented in the physical domain by their pin figures, which are shapes associated with the pin. In electrical domain, a terminal can have multiple pins and each pin can have multiple figures; however, in optical domain, an optical terminal

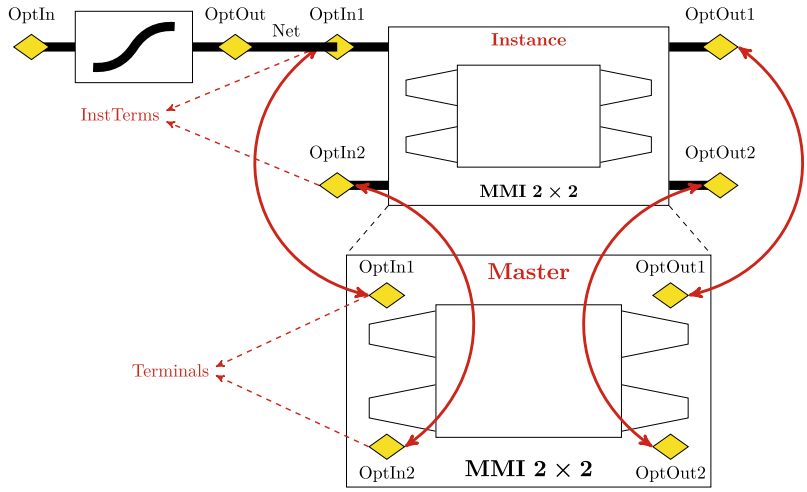


Fig. 4.3 Relationship model between nets, instance terminals, terminals, and pins

has only one pin and each optical pin has only one pin figure. It should be mentioned that a pin figure can be also be an instance; this happens, mostly, when one is implementing a symbol representation of a device to be used in a schematic.

4.4 Create Optical Port in Layout

The main APIs used to enable connectivity are: `dbCreateNet`, `dbCreateTerm`, `dbCreatePin`, and `dbAddFigToPin`; having said that, there are higher-level APIs calling the database level APIs to create the optical port such as: `phoCreatePort` and `phoAddWaveguidePorts`. Code 4.3 shows how `phoCreatePort` is used to create the "optIn" optical port and also how database level APIs can be used to create the opticl "optOut" port. If the SBend PCell is updated based on the new generator, the optical ports will be generated as shown in Fig. 4.4.

Consider the following guidelines when using optical ports in a PCell and/or design.

- Optical pin shape is a circle carrying the following attributes: angle, radius, and width. These attributes needs to be set based on the waveguide's facet (begin or end) associated with the optical port. These information are essential for auto-abutment of optical components. Similar to ccFacets, angle is defined with respect to $y \leq 0.0$
- Optical pin shape should be on a layer associated with a waveguide definition in the technology file referenced or attached to your design library. This information can be extracted by the following code:

```
techId = techGetTechFile(geGetEditCellView())
techId~>waveguideDefs~>layer~>name
```

- Auto-abutment triggers when two optical ports overlapping each other whose widths are equal and angles are compatible $angle_1 = angle_2 \pm 180.0\,°$. preferably the radius of the optical ports needs to be equal but it is not necessary for the abutment engine; the reason behind that is if two optical ports are matched with respect to all of their parameters, this means that there is no back-reflection in the waveguide junction between the optical components whose ports are abutted.

Fig. 4.4 Input port of an SBend created by `phoCreatePort`

Exercise: Use SBend ccPath object and phoAddWaveguidePorts APIs to generate "optIn" and "optOut" ports. This API would be much more efficient for optical port implementation of a waveguide.

Code 4.3 Lines 16–38 shows how optical ports can be added to SBend generator function

```
1  procedure(LUDACreateSBendWG(cellView lpp width straight1L straight2L
       straight3L radius thetaRange @key
2                           (direction "Up")
3                           (offset 0:0)
4                           (genFigs t)
5                           (genPorts t)
6                           )
7      let((sBendPolyCurve sBendPath netOptOut portX portY termOptOut
           pinShapeOptOut pinOptOut portWidth)
8          sBendPolyCurve = LUDACreateSBend(straight1L straight2L
               straight3L radius thetaRange
9                                           ?direction direction ?offset offset)
10         sBendPath = ccCreatePath(cellView lpp sBendPolyCurve~>segments ?
               defaultWidth width)
11         when(genFigs
12             ccGenFigs(sBendPath)
13         )
14         when(genPorts
15             ;This creates the input port of the SBend using phoCreatePort
16             portX = xCoord(sBendPath~>beginPt)
17             portY = yCoord(sBendPath~>beginPt)
18             portWidth = width-0.002
19             phoCreatePort(cellView list(lpp "drawing")
20                  portX portY
21                 "inputOutput" "optIn" "optIn" "optIn"
22                 sBendPath~>beginFacet~>width sBendPath~>beginFacet~>angle
                       sBendPath~>beginFacet~>radius
23                  portWidth portWidth)
24             ;This creates the output port of the SBend using low-level
                   database APIs
25             portX = xCoord(sBendPath~>endPt)
26             portY = yCoord(sBendPath~>endPt)
27             netOptOut = dbCreateNet(cellView "optOut")
28             netOptOut~>isOptical = t
29             netOptOut~>sigType = "optical"
30             termOptOut = dbCreateTerm(netOptOut "optOut" "inputOutput")
31             pinShapeOptOut = dbCreateEllipse(cellView list(lpp "drawing")
32                                 list(portX-portWidth/2.0:portY-portWidth/2.0
33                                    portX+portWidth/2.0:portY+portWidth/2.0))
34             pinOptOut = dbCreatePin(netOptOut pinShapeOptOut "optOut"
                   termOptOut)
35             phoSetPortWidth(pinShapeOptOut sBendPath~>endFacet~>width)
36             phoSetPortAngle(pinShapeOptOut sBendPath~>endFacet~>angle)
37             phoSetPortRadius(pinShapeOptOut sBendPath~>endFacet~>radius)
38         )
39         sBendPath
40     )
41 )
```

4.5 Create Optical Port for a Symbol

Similar to a photonic component's physical view that needs optical ports to be used in a hierarchical design and carry connectivity; connectivity information also needs to be enabled for its logical view (symbol). The connectivity concepts and implementation is similar to the physical view, the only difference is that the pin figure will not be circle and there is no high-level APIs such as phoCreatePort that can be used to facilitate optical port creation. As mentioned before, although pin figure is a shape object, it would be possible to be an instance especially when it is in a symbol. Code 4.4 shows how a pin can be created. This procedure can be called inside the SBend symbol implementation code as shown in lines 20 and 21 of Code 4.5. The result of recompiling SBend symbol view is shown in Fig. 4.5.

Code 4.4 Pin generation for a symbol

```
1  procedure(LUDACreateOpticalPin(cv xy netName termName pinName tailL
       pinLocation
2          @key
3          (pinH 0.025)
4          (pinW 0.05)
5          )
6      let((netId termId pinFig pinId)
7          netId = dbCreateNet(cv netName)
8          termId = dbCreateTerm(netId termName "inputOutput")
9          netId~>isOptical = t
10         netId~>sigType = "optical"
11         pinFig = dbCreatePolygon(cv list("pin" "drawing") list(xCoord(xy)-
               pinW:yCoord(xy) xCoord(xy):yCoord(xy)+pinH
12                                                          xCoord(xy)+pinW:
                                                                yCoord(xy)
                                                                xCoord(xy):
                                                                yCoord(xy)-pinH)
                                                                )
13         case(pinLocation
14             ("left"
15                 dbCreateRect(cv list("device" "drawing1") list(xCoord(xy)+
                       pinW:yCoord(xy)-0.00625 xCoord(xy)+tailL:yCoord(xy)+0.00
                       625))
16             )
17             ("right"
18                 dbCreateRect(cv list("device" "drawing1") list(xCoord(xy)-
                       pinW:yCoord(xy)-0.00625 xCoord(xy)-tailL:yCoord(xy)+0.00
                       625))
19             )
20         )
21         pinId = dbCreatePin(netId pinFig pinName)
22     )
23 )
```

Fig. 4.5 SBend symbol with its optical input and output pins

Code 4.5 Lines 20 and 21 generated the optical input and output of the SBend

```
1  pcDefinePCell(
2      list(ddGetObj("Training") "SBendExample" "symbol" "schematicSymbol")
3      list(
4          ;Here you define PCell parameter names, types, and default values
5          (magnification 1.0)
6      )
7      ;Here is the implementation section where the goemtries will be
           generated
8      let((sBendArtwork sBendArtWorkFig sBendArtWorkMag)
9          dbCreateRect(pcCellView "device" list(0.2*magnification:-0.2*
               magnification 1.0*magnification:0.2*magnification))
10         dbCreateRect(pcCellView "instance" list(0:-0.2*magnification 1.2*
               magnification:0.2*magnification))
11         sBendArtwork = LUDACreateSBendWG(pcCellView "device" 0.01 0.2 0.2 0
               .2 0.1 45.0 ?offset 0.25:-0.1 ?genPorts nil)
12         sBendArtWorkFig = car(sBendArtwork~>figs)
13         sBendArtWorkMag = foreach(mapcar pt sBendArtWorkFig~>points
14             xCoord(pt)*magnification:yCoord(pt)*magnification
15         )
16         dbCreatePolygon(pcCellView "device" sBendArtWorkMag)
17         ccRemoveFigs(sBendArtwork)
18         dbCreateLabel(pcCellView list("instance" "label") 0.6*magnification
               :-0.1*magnification "SBend" "centerLeft" "R0" "roman" 0.0625)
19         ;Generate optical pin's of SBend symbol view
20         LUDACreateOpticalPin(pcCellView 0:0 "optIn" "optIn" "optIn" 0.2*
               magnification "left")
21         LUDACreateOpticalPin(pcCellView 1.2*magnification:0 "optOut" "
               optOut" "optOut" 0.2*magnification "right")
22     )
23 )
```

Exercise: Implement a general utility procedure that gets number of electrical and optical pins with their location around the periphery of instance box and generates its symbol.

4.6 Component Description Format (CDF)

In the previous section, we discussed about PCell parameters and how those parameters are used to generate different variants of a cell. PCell parameters may not be sufficient to describe all the required parameter needed to describe a cell. Component Description Format (CDF) provides a single location where a user can specify all parameters and the attributes of parameters for a cell so that other application can use them; consequently, CDF parameters are superset of PCell parameters. CDF can also provides you functionality to dynamically change how parameters and displayed and updated. Code 4.6 show how CDF is defined for a SBend.

Code 4.6 SBend Component Description Format (CDF)

```
let((cellId cdfId)
    cellId = ddGetObj("Training" "SBendExample")
    when(cdfId = cdfGetBaseCellCDF(cellId)
        cdfDeleteCDF(cdfId)
    )
    cdfId = cdfCreateBaseCellCDF(cellId)
    cdfCreateParam(cdfId ?name "lpp" ?prompt "Layer Purpose" ?type "string"
         ?storeDefault "yes"
        ?defValue "waveguide" ?display "t"
    )
    cdfCreateParam(cdfId ?name "straight1L" ?prompt "Straight Section 1
        Length" ?type "string" ?storeDefault "yes"
        ?defValue "10.0" ?display "t"
    )
    cdfCreateParam(cdfId ?name "straight2L" ?prompt "Straight Section 2
        Length" ?type "string" ?storeDefault "yes"
        ?defValue "10.0" ?display "t"
    )
    cdfCreateParam(cdfId ?name "straight3L" ?prompt "Straight Section 3
        Length" ?type "string" ?storeDefault "yes"
        ?defValue "10.0" ?display "t"
    )
    cdfCreateParam(cdfId ?name "radius" ?prompt "SBend Radius" ?type "
        string" ?storeDefault "yes"
        ?defValue "25.0" ?display "t"
    )
    cdfCreateParam(cdfId ?name "thetaRange" ?prompt "Span Angle" ?type "
        string" ?storeDefault "yes"
        ?defValue "45.0" ?display "t"
    )
    cdfCreateParam(cdfId ?name "direction" ?prompt "Layer Purpose" ?type "
        cyclic" ?storeDefault "yes"
        ?defValue "Up" ?display "t" ?choices list("Up" "Down")
    )
    cdfSaveCDF(cdfId)
)
```

4.6.1 Callbacks

As mentioned CDF can be used to dynamically validate and/or update cell parameters. This is done by callback functions. Callbacks are SKILL®^{TM} procedure that are called when a CDF parameter is modified. These callbacks are GUI based callbacks meaning they are not triggered when a CDF parameter is modified by SKILL APIs. Assume that SBend radius shouldn't go below 10.0, so we need a callback for `radius` parameter so that when its value is modified to be less than 10.0, the callback preserves the last value before this modification. As is shown Code 4.7, callback can be passed as an argument `?callback`; modify the Code 4.6 to add the callback function and then load the `checkRadius` procedure, now if you want to modify the radius below 10.0, callback triggers and revert back the value to its previous one. `cdfgData` is a data structure populated with CDF parameters. `cdfgForm` is the actual form related to "Edit Instance Property" form, as shown in Fig. 4.6. Although one can access CDF parameters and attributes through `cdfgData` and `cdfgForm`, it is a good practice to use `cdfgData` instead of `cdfgForm` wherever possible in the code.

Code 4.7 Add a callback to "radius" parameter to prevent radius from going below 10.0

```
cdfCreateParam(cdfId ?name "radius" ?prompt "SBend Radius" ?type "string"
        ?storeDefault "yes" ?defValue "25.0" ?display "t" ?callback "
            checkRadius()"
    )
procedure(checkRadius()
    let(()
        when(cdfParseFloatString(cdfgData~>radius~>value) < 10.0
            cdfgData~>radius~>value = cdfgForm~>radius~>lastValue
        )
    )
)
```

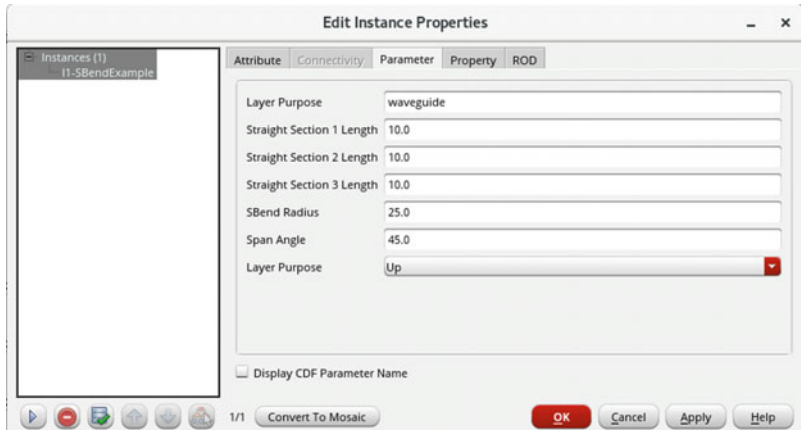


Fig. 4.6 Edit Instance Property form after CDF data is implemented and loaded

4.7 Schematic Driven Layout (SDL) Methodology

Most of custom design cycles starts with a designer capturing the logical representation of designed IP. In the electronic design automation terminology the final outcome of this stage would be a schematic view of a design. This schematic view not only drives the layout view but also drives most of simulators; as a result, it would be important that a designer spend enough time to partition the IP hierarchically and create a clean schematic.

As mentioned the schematic view is used to generate the layout view, which is IP's physical design. Because a schematic view captures a design's components and how they are connected, the generated layout also carries the connectivity information; and consequently layout can be compared with schematic to make sure the followings: (1) components are properly connected in the layout and there is no open and/or short connections, (2) Component parameters are matched between schematic and layout. If a layout engineer changes any parameter in the layout, the parameter can be back-annotated into schematic so that the schematic is synchronized with layout, and the effect of layout change can be included into the simulation. Figure 4.7 show a simple SDL flow. Make sure that you follow the following guidelines in SDL methodology:

- Port associated with a symbol and layout view of a cell needs to be matched in terms of name, direction, and type.

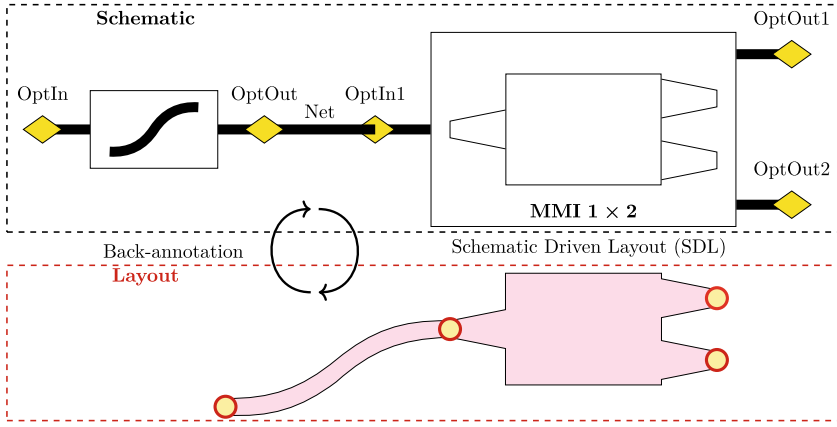


Fig. 4.7 Simple SDL flow shows how schematic drives layout and how layout changes can be back-annotated to schematic

- wires are used in schematic to represent nets connecting instance terminals. In the physical view, electrical wires are represented as metal shapes, but, we don't have a concept of optical wires in the layout. That is why the optical components are connected when their optical ports are overlapped as shown in Fig. 4.7.
- Typically optical ports' direction should be set to "inputOutput".
- Because source of connectivity is coming from schematic, any change in connectivity needs to be initiated from schematic and propagated to the layout.

Introduction to Design Implementation Using Scripting

5

Capturing a physical or logical design using tools benefiting from graphical user interface such as Virtuoso® schematic or layout editor is very common. Having said that, capturing a design by a script can provide an efficient method for certain designs where one can find repeating patterns. In this chapter, some general concepts and SKILL®^{TM} APIs to capture a design by a script is introduced.

5.1 Create an Instance

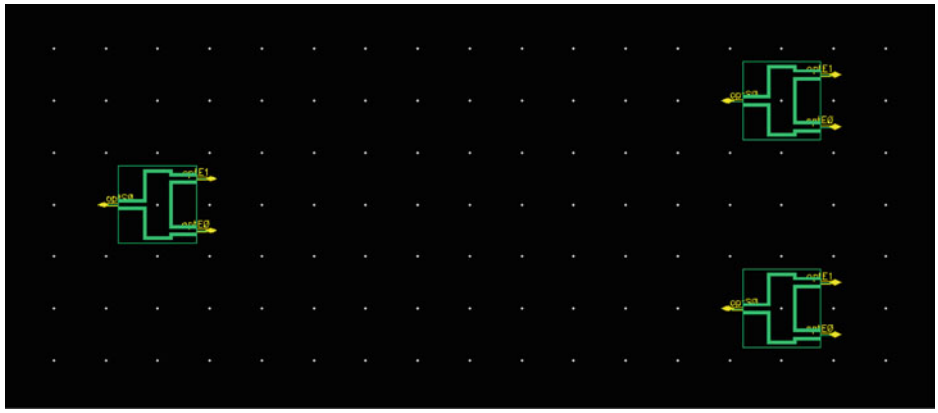
The typical APIs used to create an instance are `dbCreateInst`, `dbCreateInstByMasterName`, `dbCreateParamInst`,`dbCreateParamInstByMasterName`. Because most of the photonic components' physical views are parameterized, `dbCreateParamInst`, `dbCreateParamInstByMasterName` are used more frequently in physical design scripting. The following code shows how to create MMI1x2 instances with default parameters. The result of running Code 5.1 is shown in Fig. 5.1a.

Code 5.1 Create a parameterized cell instance using SKILL APIs

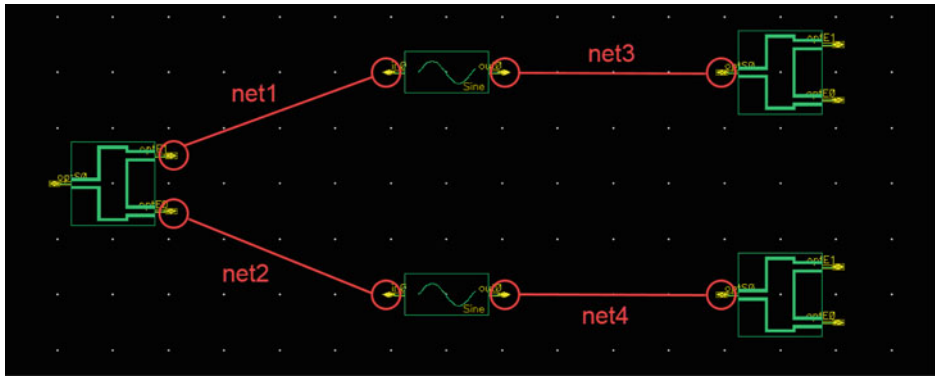
```
1 cv = geGetEditCellView()
2 dbCreateParamInstByMasterName(cv "gopdk" "MMI_wg_1x2" "symbol" "I0" 0:0 "
      R0")
3 dbCreateParamInstByMasterName(cv "gopdk" "MMI_wg_1x2" "symbol" "I1" 6:1 "
      R0")
4 dbCreateParamInstByMasterName(cv "gopdk" "MMI_wg_1x2" "symbol" "I2" 6:-1 "
      R0")
```

To connect three MMIs shown in Fig. 5.1a, three sine connector waveguides are used as shown in Fig. 5.1b. These connectors are created similar to MMIs as shown in Code 5.2. Figure 5.1b is also annotated with required optical logical connections. The circles represent

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_5



(a) Loading Code 5.1 to create three instances of MMI1x2 instance.



(b) Adding sine waveguide connectors between MMIs, Code 5.2

Fig. 5.1 Generating schematic by scripting

instance terminals and the thin red lines represents optical wires. It is important to remind that the object carries connectivity information is a `net`, so the instance terminals connected to an optical wire along with the wire object are on the same net object.

Code 5.2 Create connector waveguides between MMIs

```
1 cv = geGetEditCellView()
2 dbCreateParamInstByMasterName(cv "gopdk" "wgSineConnector" "symbol" "I3" 3
      :1 "R0")
3 dbCreateParamInstByMasterName(cv "gopdk" "wgSineConnector" "symbol" "I4" 3
      :-1 "R0")
```

5.2 Establish Connectivity

As shown by thin red lines in Fig. 5.1b four net objects need to be created to establish connectivity. A procedure is defined in Code 5.3 to create an optical wire between two instance terminals of two instances. Here are some guidelines to create a schematic using SKILL APIs:

- Create instances using proper SKILL APIs: `dbCreateInst, dbCreateInstByMasterName, dbCreateParamInst, dbCreateParamInstByMasterName`
- Create required connectivity by `dbCreateNet`
- Extract coordinates of pin figures associated with instance terminals. It is important to know that how pins are defined in the symbol. For this example, the pin figures are implemented by instances so we use `xy` representing the center of the diamond shape. In other situations where the pin figure is implemented as an OA shape object, one can use the center of shape's bounding box.
- Establish Connectivity using `dbCreateConnByName`
- Define signal type for optical nets
- Create optical wire using `dbCreateLine` and label representing wire name using `dbCreateLabel`
- Establish relationship between wire and its label by setting the label's parent to be wireId

Code 5.3 Create required connectivity for instance terminals

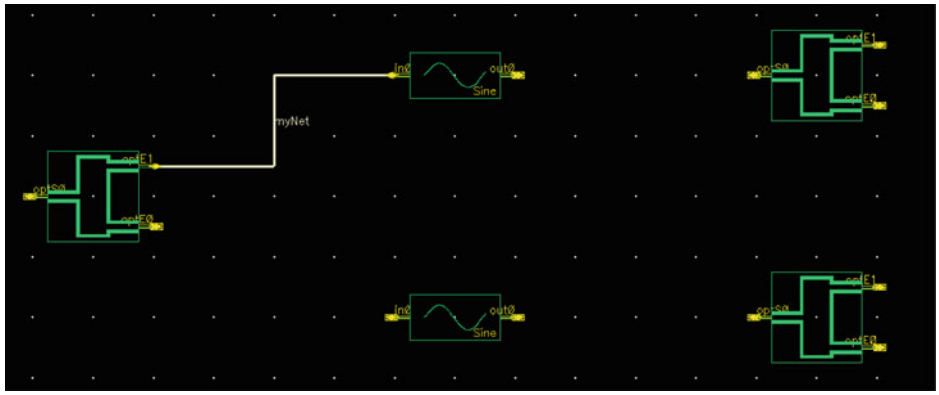
```
1  procedure(LUDACreateOpticalWire(cv instSName instTermS instEName instTermE
       netName)
2      let((netId instS instE termS pinFigS pinFigSCenter termE pinFigE
           pinFigECenter centeWire opticalWire opticalWireLabel)
3
4          instS = dbFindAnyInstByName(cv instSName)
5          instE = dbFindAnyInstByName(cv instEName)
6
7          termS = car(setof(term instS~>master~>terminals term~>name == instTermS
               ))
8          pinFigS = car(car(termS~>pins)~>figs)
9          pinFigSCenter = dbTransformPoint(pinFigS~>xy instS~>transform)
10
11         termE = car(setof(term instE~>master~>terminals term~>name == instTermE
               ))
12         pinFigE = car(car(termE~>pins)~>figs)
13         pinFigECenter = dbTransformPoint(pinFigE~>xy instE~>transform)
14
15         centeWire = centerBox(list(pinFigSCenter pinFigECenter))
16         opticalWire = dbCreateLine(cv list("wire" "drawing") list(
               pinFigSCenter xCoord(centeWire):yCoord(pinFigSCenter)
17                           xCoord(centeWire):yCoord(pinFigECenter)
                                 pinFigECenter))
18
19         opticalWireLabel = dbCreateLabel(cv list("wire" "label") centeWire
               netName "centerLeft" "R0" "stick" 0.0625)
20
```

```
21
22          netId = dbCreateNet(cv netName)
23          netId~>sigType = "optical"
24
25          dbCreateConnByName(netId instS instTermS)
26          dbCreateConnByName(netId instE instTermE)
27          opticalWireLabel~>parent = opticalWire
28      )
29  )
30
31  cv = geGetEditCellView()
32  LUDACreateOpticalWire(cv "I0" "optE1" "I3" "in0" "myNet")
33  dbSave(cv)
34  schCheck(cv)
```

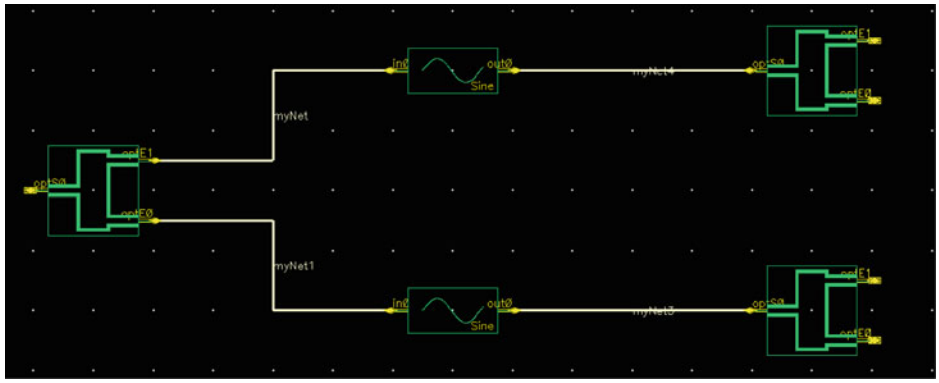
Figure 5.2a shows the result of running Code 5.3 to create an optical wire between two instance terminals. You should be able to create the remaining optical wires as an exercise to get the final schematic capture as shown in Fig. 5.2b. As mentioned in the previous chapter, schematic is used as the main database view to capture the design from which the layout view and connectivity information is extracted. Figure 5.2c shows the generated layout with the connectivity information, nets, are shown using white solid lines.

5.3 Creating Terminals and Pins for Hierarchical Design Use

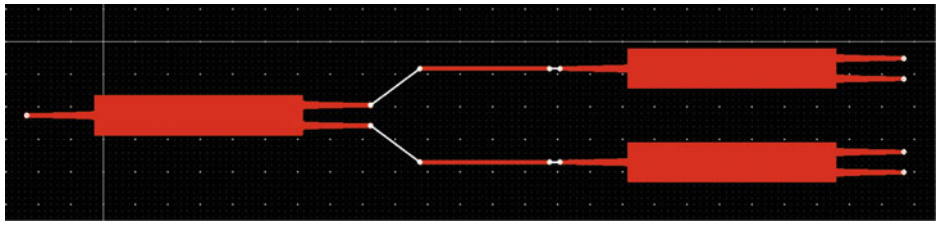
For our design in Fig. 5.2a to be complete, optical pins and terminals needs to be added. It is important that for connectivity to be established, there shouldn't be an actual wire connecting two instance terminals or an instance terminal and a terminal. As discussed in the previous section, it is possible to create a `line` and `label` objects and assign the `line` as the parent of `label`; this would force `line` representing the wire to carry the same net name as the `label`. We use this method to create wire stubs with proper label names based on optical terminals. Code 5.5 provides two utility functions: (1) `LUDACreateOpticalPin` which is used to create an optical terminal and pin, (2) `LUDACreateWireStub` which is used to create a wire stub with a label. For our design example, 5 optical terminals and pins needs to be created: `optIn, optOut1, optOut2, optOut3, optOut4`; these can be generated using implemented utility functions as shown in Code 5.4. The schematic and layout of the complete design example is shown in Fig. 5.3. Try to complete the sections "up" and "down" in Code 5.5 as an exercise.



(a) Create an optical wire between "I0:optE1" and "I3:in0" by using Code 5.3

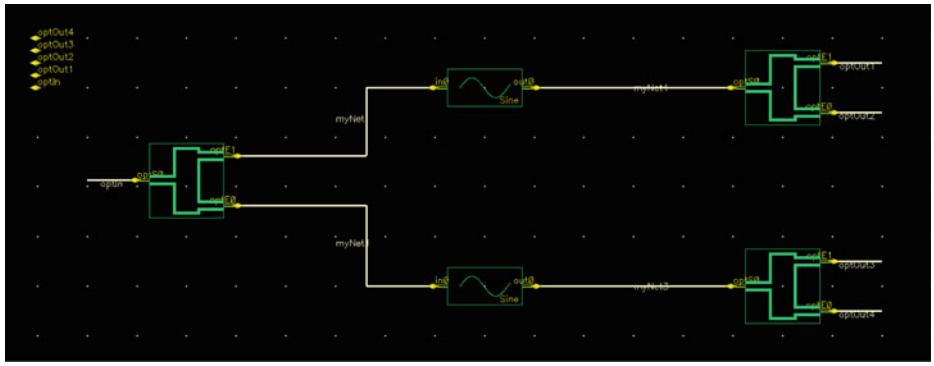


(b) Final schematic capture with optical wires and established connectivity

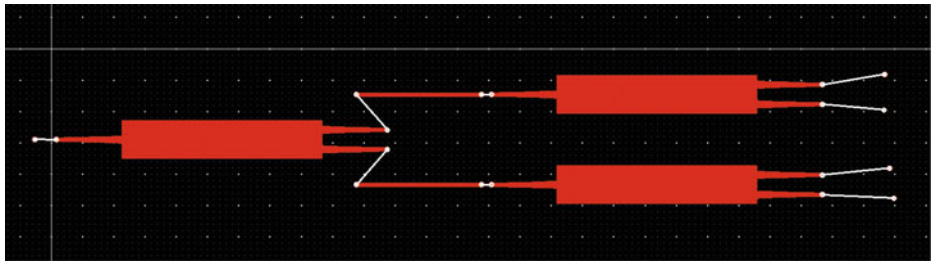


(c) Layout view generated from the final schematic showing the connectivity information established in schematic

Fig. 5.2 Creating optical wire using database-level SKILL APIs and establishing connectivity



(a) Scripted schematic



(b) Generated layout using schematic driven layout flow

Fig. 5.3 The scripted schematic and layout of the complete design example with optical terminals and pins

Code 5.4 Generating required optical terminals and pins to complete design example shown in Fig. 5.2a

```
1 LUDACreateOpticalPin(cv list("optIn" "optOut1" "optOut2" "optOut3" "
     optOut4") -1:1)
2 LUDACreateWireStub(cv "I0" "optS0" "optIn" "left")
3 LUDACreateWireStub(cv "I1" "optE1" "optOut1" "right")
4 LUDACreateWireStub(cv "I1" "optE0" "optOut2" "right")
5 LUDACreateWireStub(cv "I2" "optE1" "optOut3" "right")
6 LUDACreateWireStub(cv "I2" "optE0" "optOut4" "right")
7 dbSave(cv)
8 schCheck(cv)
```

Code 5.5 Utility procedures to (1) create optical terminals and pins; (2) create wire stubs with proper label names

```
1 procedure(LUDACreateOpticalPin(cv pinListName xy)
2     let((pinInstId instId netId termId pinId
3         (i 0)
4         )
5         pinInstId = dbOpenCellViewByType("opticalLib" "optPinInOut" "symbol")
6         foreach(pinName pinListName
```

```
            instId = dbCreateInst(cv pinInstId pinName xCoord(xy):yCoord(xy)+0.1
                25*i "R0")
            netId = dbCreateNet(cv pinName)
            netId~>sigType = "optical"
            termId = dbCreateTerm(netId pinName "inputOutput")
            pinId = dbCreatePin(netId instId pinName)
            i++
        )
    )
)

procedure(LUDACreateWireStub(cv instName instTermName netName direction)
    let((opticalWire netId opticalWireLabel instId termId pinFigId pinFigCenter)
        instId = dbFindAnyInstByName(cv instName)
        termId = car(setof(term instId~>master~>terminals term~>name ==
            instTermName))
        pinFigId = car(car(termId~>pins)~>figs)
        pinFigCenter = dbTransformPoint(pinFigId~>xy instId~>transform)
        case(direction
            ("up"
                ;Implement this section as an exercise
            )
            ("down"
                ;Implement this section as an exercise
            )
            ("left"
                opticalWire = dbCreateLine(cv list("wire" "drawing") list(
                    pinFigCenter xCoord(pinFigCenter)-0.5:yCoord(pinFigCenter)))
                opticalWireLabel = dbCreateLabel(cv list("wire" "label") xCoord(
                    pinFigCenter)-0.25:yCoord(pinFigCenter)
                        netName "centerTop" "R0" "stick" 0.0625)

                netId = dbFindNetByName(cv netName)

                dbCreateConnByName(netId instId instTermName)
                opticalWireLabel~>parent = opticalWire
            )
            ("right"
                opticalWire = dbCreateLine(cv list("wire" "drawing") list(
                    pinFigCenter xCoord(pinFigCenter)+0.5:yCoord(pinFigCenter)))
                opticalWireLabel = dbCreateLabel(cv list("wire" "label") xCoord(
                    pinFigCenter)+0.25:yCoord(pinFigCenter)
                        netName "centerTop" "R0" "stick" 0.0625)

                netId = dbFindNetByName(cv netName)

                dbCreateConnByName(netId instId instTermName)
                opticalWireLabel~>parent = opticalWire
            )
        )

    )
)
```

Introduction to Software Development and General Scripting

6

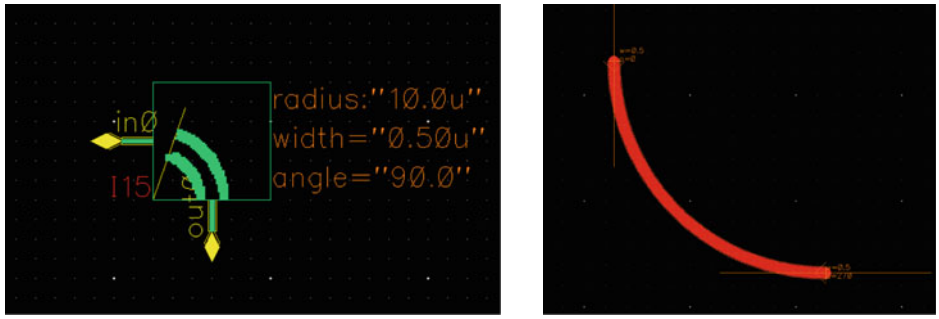
Scripting using SKILL®^{TM} language and software development using SKILL++ needs a separate book. In this chapter, some general scripting, software development methodologies, and language constructs get introduced by examples.

6.1 Orientation Matching Between Schematic and Layout

As mentioned in the previous chapter, the standard PIC design flow is based on capturing a schematic representing the logical design from which the physical design, layout, is generated. Although a schematic carries logical view of a design using symbols; a properly developed symbols's artwork should resembling the actual physical design. In most of the EDA tools supporting SDL flow, when a physical design is created it carries the same orientation as the one in the schematic; consequently, while a symbol artwork is being implemented, it is important to make sure that the artwork in combination with different orientation resembles its associated layout view having the same orientation. Figure 6.1 shows the `R0` orientation of an arc waveguide representation as symbol and layout views. As can be seen from Fig. 6.1, the `R0` orientation of the symbol doesn't resemble the `R0` orientation of the layout; so there is an initial transform to be applied to the layout instance so that they are consistent. In Fig. 6.2 layout instance's orientation has been changed to `MYR90` so that the symbol artwork and layout views are consistent with regards to shapes and input/output ports.

Figure 6.3 captures an asymmetric-MZI structure schematic and initial layout view generated from schematic. From the generated layout view, it is obvious that initial orientation of the arc waveguides and the output MMI are not set properly. As is demonstrated in Fig. 6.2, the initial orientation of an arc bend should be `MYR0` and the consequent orientation modifications should be concatenated to this orientation. Code 6.1 shows how instance of a layout

A. Farsaei, *Introduction to Layout Design and Automation of Photonic Integrated Circuits*,
Synthesis Lectures on Digital Circuits & Systems,
https://doi.org/10.1007/978-3-031-25288-4_6



(a) Symbol View (b) Layout View

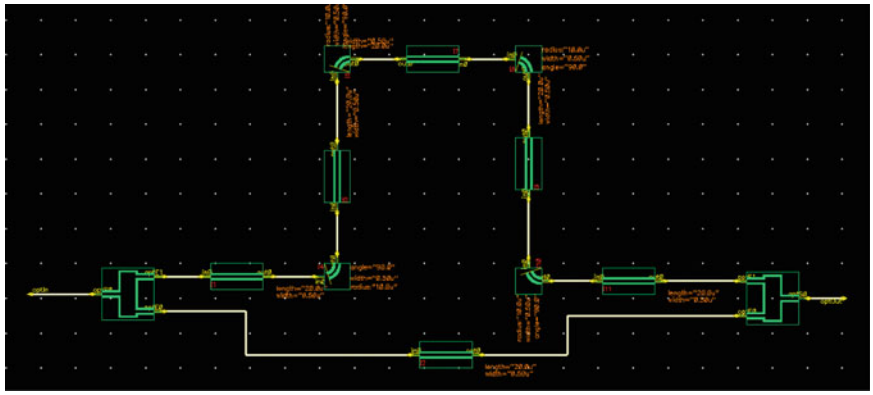
Fig. 6.1 An S-Bend symbol artwork and layout views with `R0` orientation

(a) Symbol View (b) Layout View

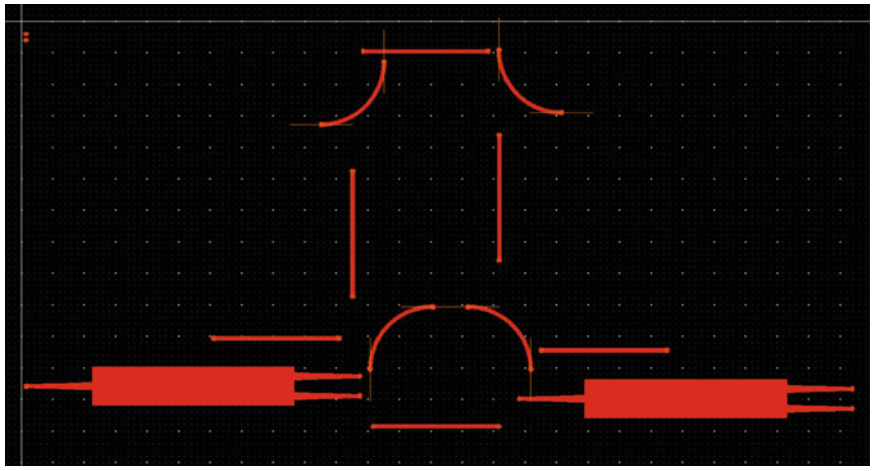
Fig. 6.2 An S-Bend symbol artwork and layout views with `MYR90` orientation

view, without any hierarchy, can be adjusted based on their bounded objects in schematic. In this example, we have used an instance's name to find the binding between layout and schematic. In Code 6.1, the only special component that needs special orientation adjustment is "wgBend" and the rest of them, can inherit the same orientation as its bounded schematic symbol. The result of calling this function is shown in Fig. 6.4.

Exercise: Although instance names have been used to find the binding between schematic and layout, this is not a robust method because instance names of binded objects can be different. Try to investigate `bndGetBoundObjects` and see how Code 6.1 can be modified to incorporate this function.



(a) Schematic View



(b) Initial Layout View

Fig. 6.3 Schematic capture of an asymmetric-MZI with its associated initial layout

Code 6.1 A procedure adjusting orientation of layout instances based on their associated bounded schematic objects

```
procedure(LUDAAdjustOrientation(layCVID schCVID)
    let((schematicOrien newLayTransform schematicBNDInst)
        foreach(inst layCVID~>instances
            schematicBNDInst = car(setof(schInst schCVID~>instances schInst
                ~>name == inst~>
                    name))
            schematicOrien = schematicBNDInst~>orient
            if(inst~>cellName == "wgBend"
                then
                    newLayTransform = dbConcatTransform(list(0:0 "MYR90" 1.
                        0) list(car(inst~>
```

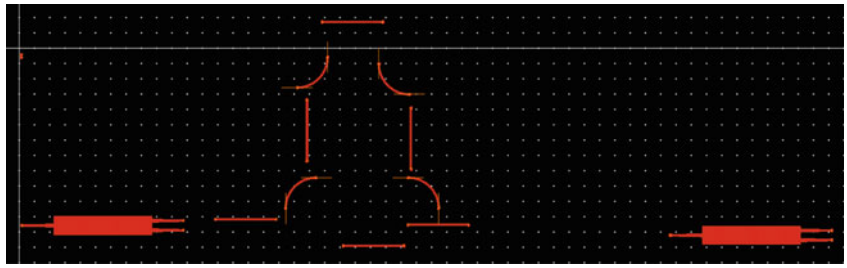
```
10                                                  transform) schematicOrien 1.0))
11                      inst~>transform = newLayTransform
12                  else
13                      newLayTransform = list(car(inst~>transform)
                            schematicOrien 1.0)
14                      inst~>transform = newLayTransform
15              )
16          )
17          t
18      )
19  )
20  LUDAAdjustOrientation(geGetEditCellView() schCVID)
```

6.2 Instance Chaining

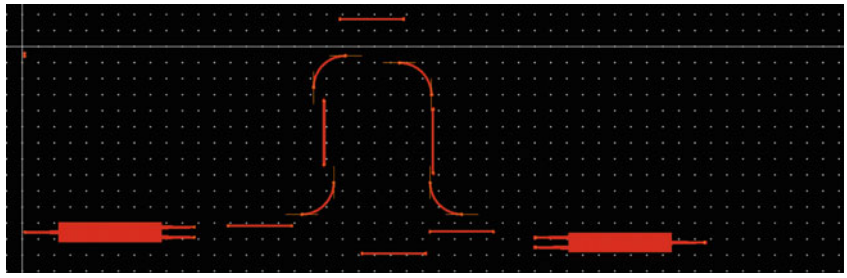
As shown in the previous section, we were able to adjust the orientation of the layout instances so that they resemble what is captured in the schematic. Although instances in Fig. 6.4 has been properly adjusted with regards to the orientation, all the optical nets are "open" and components are not connected. The chaining solution that will be discussed in this section is going to resolve this issue. The following items are assumed in development of the solution.

- The solution is based on connectivity driven environment
- Each optical net is connected to either exactly two instance terminals or an instance terminal and an optical terminal
- The chaining solution doesn't modify components' parameters including rotation
- The solution is for chaining devices having two ports and will stop chaining when it reaches a component having more than two optical ports
- As discussed in the previous chapter, an optical instance terminal is associated with an optical terminal that has a pin with an ellipse pin figure. For the chaining algorithm we need to find the center of the pin fig
- The algorithm needs an anchor instance terminal to which consecutive instance terminals are chained
- The adjacent components in the chain are compatible with respect to the rotation; i.e., the angle of optical ports supposed to be connected follows $angle_{Port1} = angle_{Port2} \pm 180.0$.

Code 6.2 shows how a basic chaining solution can be implemented for Fig. 6.4b. It should be reminded that a terminal associated with an instance terminal belongs to the master view of an instance, so to find the proper location of a pinFig associated with an instance terminal, one needs to apply `dbTransform` based on `inst~>transform` as is shown in Lines 4 and 5 of Code 6.2. The result of executing Code 6.2 while left MMI1x2 is selected is shown in Fig. 6.5; it should be obvious from the figure that why chaining solution for devices having more than two optical ports can be challenging.



(a) Initial generated layout of Asymmetric MZI

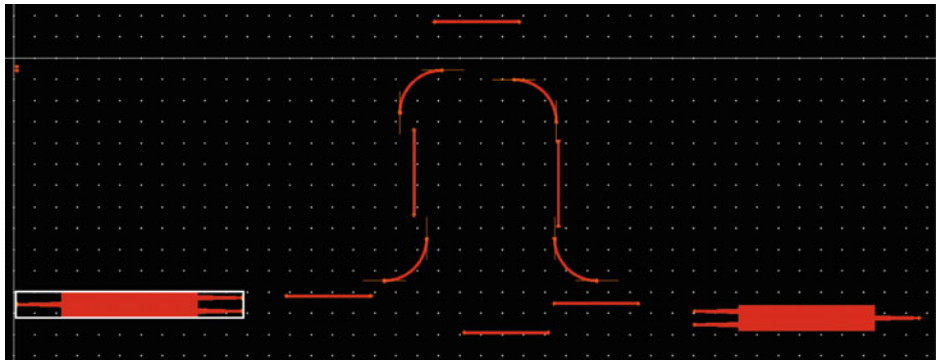


(b) Layout view of the Asymmetric MZI after instances' orientation adjustment

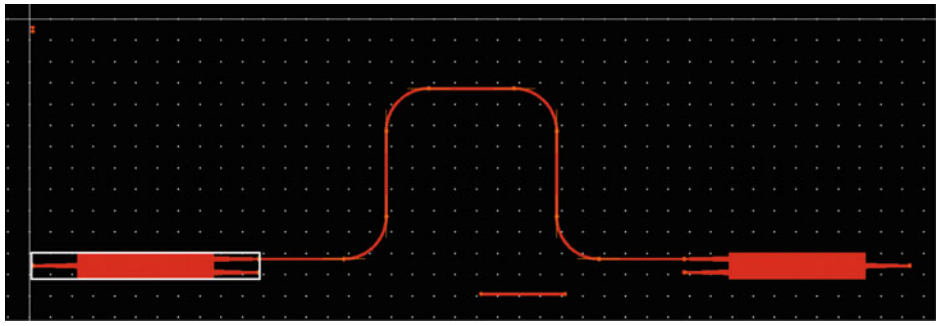
Fig. 6.4 Applying `LUDAAdjustOrientation` function on the initial generated layout of asymmetric MZI

Code 6.2 A basic chaining solution for Fig. 6.4b

```
procedure(LUDAChain(anchorInstTerm)
    let((anchorInstTerms connectedInstTerm anchorInstTermPinCenter
         connectedInstTermPinCenter requiredTransform nextInstTerm)
        when(connectedInstTerm = car(setof(instTerm anchorInstTerm~>net~>
             instTerms instTerm != anchorInstTerm))
            anchorInstTermPinCenter = centerBox(dbTransformBBox(car(car(
                anchorInstTerm~>term~>pins)~>figs)~>bBox anchorInstTerm~>
                inst~>transform))
            connectedInstTermPinCenter = centerBox(dbTransformBBox(car(car(
                connectedInstTerm~>term~>pins)~>figs)~>bBox
                connectedInstTerm~>inst~>transform))
            requiredTransform = xCoord(anchorInstTermPinCenter)-xCoord(
                connectedInstTermPinCenter):yCoord(anchorInstTermPinCenter)
                -yCoord(connectedInstTermPinCenter)
            dbMoveFig(connectedInstTerm~>inst connectedInstTerm~>cellView
                list(requiredTransform "R0" 1))
            nextInstTerm = setof(instTerm connectedInstTerm~>inst~>
                instTerms instTerm != connectedInstTerm)
            when(length(nextInstTerm) == 1
                LUDAChain(car(nextInstTerm))
            )
        )
    )
```



(a) Asymmetric MZI before its top branch is chained



(b) Asymmetric MZI after executing Code 6.2 to chain its top branch

Fig. 6.5 Chaining top branch of an asymmetric MZI using Code 6.2 with "optE1" of the left MMI1x2 as the anchor point

```
14 )
15 startInstTerm = car(setof(instTerm css()~>instTerms instTerm~>name == "
       optE1"))
16 LUDAChain(startInstTerm)
```

6.3 Extract Connectivity Based on Physical Proximity

There are cases that the layout is generated using legacy tools or flows where connectivity is not generated. In this section, we are focusing on generating connectivity for these kind of designs. Figure 6.6 shows a layout design where the components are connected but there is no connectivity information, nets, in this design. The solid circles in the figure shows where pin figures for each component is located. We are going to break this problem into two parts: the first part is to extract pin figures supposed to be on the same net based on the proximity measure, the second part would be to generate a connectivity graph describing

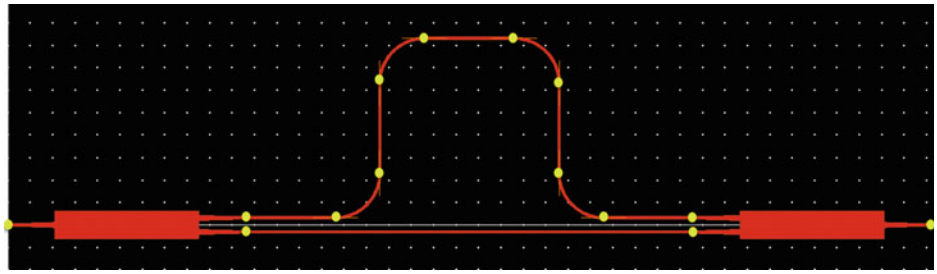


Fig. 6.6 Layout design of an asymmetric MZI without any nets, the connection points, pin figures, are shown by yellow circles

how instance terminals are related. This connectivity graph can be used in different graph traversal solutions such as Depth First Search (DFS) or Breadth First Search (BFS) to find optical path between different instance terminals.

6.3.1 Pin Figures Proximity Sort

As shown in Fig. 6.6 connectivity extraction is based on proximity, so to generate the connectivity the instances' pin figures should be sorted as shown in Algorithm 1. In the sort algorithm the un-ordered list of points representing center of components' pin figures are sorted in such a way that each successive point of the list is closest to its predecessor point. The measure used to measure distance between the points is Euclidean distance $x^2 + y^2$. The implementation of the sort algorithm is shown in Code 6.3. The `LUDASortcarList` in Code 6.3 sorts a list based on its `car`. For example if the $inputList$ = ((0 : 0 "1") (4 : 4 "4") (3 : 3 "3") (2 : 2 "2")) the sorted list would be ((0 : 0 "1") (2 : 2 "4") (3 : 3 "3") (4 : 4 "2"))

Algorithm 1 Sort algorithm to create a list of pin figures's center points which each successive element is closest to its predecessor point in the list

1: $unorderedLisr \leftarrow list$(center of components' pin figures)
2: $orderedList \leftarrow list()$
3: $refPt \leftarrow unorderdList[1]$
4: $subList \leftarrow unorderedList[2..end]$
5: **while** length($orderedList$)<length($unorderedList$) **do**
6: $newRefPt \leftarrow$ findClosestPt($refPt\ subList$)
7: $sortedList \leftarrow append(sortedList\ newRefPt)$
8: $refPt \leftarrow neeRefPt$
9: $subList \leftarrow$ remove $sortedList$ points from $undorderedList$
10: **end while**

Code 6.3 Implementation of algorithm Algorithm 1

```
procedure(LUDASortcarList(inputList)
    let(
        (subList refPt newRefPt sortedInstTerms lastPt
        (sortedList ncons(car(inputList)))
        )
        refPt = car(inputList)
        subList = cdr(inputList)
        sortedInstTerms = ncons(cadr(refPt))
        while(length(sortedList) < length(inputList)-1
            newRefPt = LUDAFindClosestPt(refPt subList)
            sortedList = append1(sortedList newRefPt)
            sortedInstTerms = cons(cadr(newRefPt) sortedInstTerms)
            refPt = newRefPt
            subList = setof(pt inputList !member(cadr(pt) sortedInstTerms))
        )
        lastPt = car(setof(pt inputList !member(cadr(pt) sortedInstTerms)))
        sortedList = append1(sortedList lastPt)
    )
)
procedure(LUDAFindClosestPt(refPt subList)
    let((closestPt minDistance newMinDistance)
        minDistance = LUDACartesianDist(car(refPt) car(car(subList)))
        closestPt = car(subList)
        foreach(pt cdr(subList)
            newMinDistance = LUDACartesianDist(car(refPt) car(pt))
            when(newMinDistance < minDistance
                closestPt = pt
            )
        )
        closestPt
    )
)
procedure(LUDACartesianDist(pt1 pt2)
    let(()
        sqrt((xCoord(pt1)-xCoord(pt2))**2 + (yCoord(pt1)-yCoord(pt2))**2)
    )
)
```

6.3.2 Connectivity Graph

SKILL++ provides object-oriented interface where software developers can create scalable software that can enhance productivity and quality while reducing the maintenance cost. The SKILL++ language constructs used for object-oriented implementation are: class objects, generic functions, and methods specialized on generic functions based on class type it is acting on.

Code 6.4 shows the implementation of the connectivity graph. The connectivity graph class is defined based on its encapsulated data as follows:

- `cv` initialized by `geGetEditCellView()`
- `nodes` initialized by `nil`and a setter function `setGraphNodes` representing a list of pin figures center and associated instance terminals
- `tolerance` initialized by $1e-3$ defining a threshold value. Two pin figures are considered connected if theirs center distance is below this value
- `adjList` initialized by an an empty association table. This adjList table captures connectivity information of the layout design(cellview) using adjacency list representation of a the connectivity graph.

As is shown in Code 6.4, a class should have some methods to act on its encapsulated data. The way that these methods are defined for a class in SKILL++ is that they specialize a generic function for the class. `LUDAExtractGraphNodes` is a method acts on a class whose type is "ConnectivityGraph". The specialization is defined by the argument type as shown in line 11 Code 6.4. In general, it would be better to define a generic function before specializing it for different class types; however, `defmethod` will create a generic method if it doesn't exists.

Exercise: Use `defgeneric` to define a generic function for `LUDAPrintGraphList` where it throws a warning printing the class type of the passed argument.

Solution: The solution is shown in the following Code where `className` and `classOf` is used to print the class type of the input argument when it is not of type "Connectivity".

```
1 defgeneric(LUDAPrintGraphList (graph)
2     let(()
3         warn("graph is not of type Connectivity: %s" className(classOf
              (graph)))
4     )
5 )
```

Figure 6.9 shows asymmetric MZI's layout design with its associated instance names. To illustrate how a graph adjacency list can be created for this design, the layout is converted to a signal flow diagram as shown in Fig. 6.7. Although this flow diagram shows how the optical mode is propagating through the components, for it to be amenable for coding, it needs to be converted to a graph data structure. Figure 6.8 shows the graph representation of the signal flow diagram. The graph nodes and edges are defined as follows:

- Graph nodes are instance terminals of the layout design (cellview)
- An edge between two instance terminal exists if either they belong to the same instance or distance between their pin figures' center is less than `tolerance` defined in the Connectivity class.

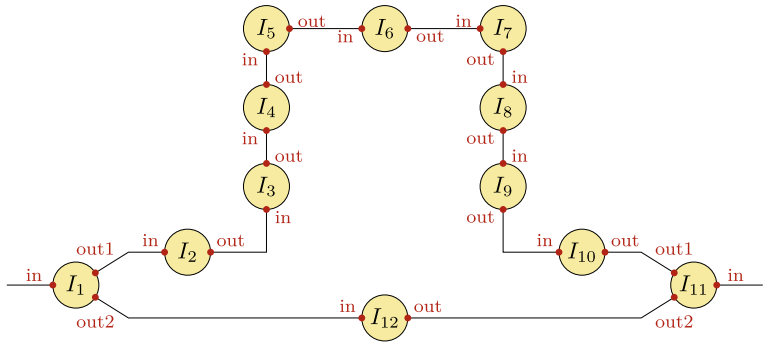


Fig. 6.7 Signal flow diagram of asymmetric MZI shown in Fig. 6.9

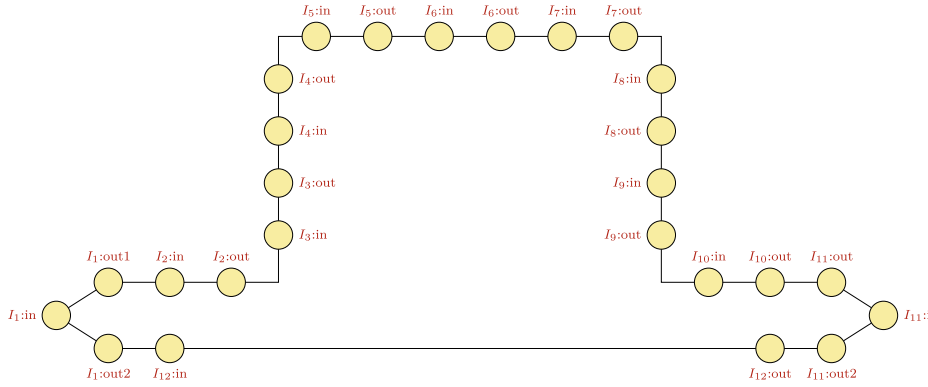


Fig. 6.8 Graph representation of asymmetric MZI shown in Fig. 6.9

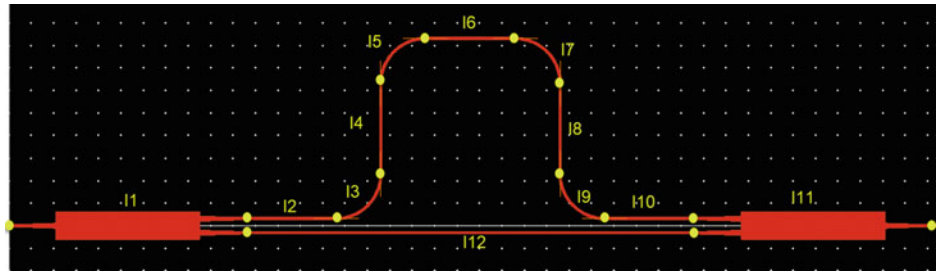


Fig. 6.9 Layout design of an asymmetric MZI with instance names

Two common ways to represent a graph as a data structure is an adjacency matrix or list. Adjacency list representation is used for its efficient implementation. Some elements of the adjective list shown in Eq. 6.1.

$$\begin{aligned} adjList[I_1 : in] &\rightarrow list(I_1 : out_1, I_1 : out_2) \\ adjList[I_1 : out] &\rightarrow list(I_1 : in, I_2 : in) \\ adjList[I_2 : in] &\rightarrow list(I_2 : out, I_1 : out_1) \end{aligned} \tag{6.1}$$

The methods defined for an object of type "Connectivity", Code 6.4, are: `LUDAExtractGraphNodes`, `LUDABuildGraph`, and `LUDAPrintGraphList`. The role of each method in adjacency graph generation is as follows:

- `LUDAExtractGraphNodes`: This methods populates graph nodes as a list and sort it based on pin figures proximity as discussed in this section. Each list member is a two-member list object: (1) an instance term's pin figure center location, (2) instance term objectId
- `LUDABuildGraph`: This method which should be called after `LUDAExtractGraphNodes` gets a graph's nodes and generates a graph's adjacency list data structure
- `LUDAPrintGraphList`: This method prints the adjacency list similar to the Eq. 6.1 for debugging purposes.

Code 6.4 Connectivity graph implementation in SKILL++

```
defclass(ConnectivityGraph
    ()
    (
        (cv @initarg cv @initform geGetEditCellView())
        (nodes @initarg nodes @initform nil @writer setGraphNodes)
        (tolerance @initarg tolerance @initform 1e-3)
        (adjList @initarg adjList @initform makeTable('adjList nil))
    )
)

defmethod(LUDAExtractGraphNodes ((graph ConnectivityGraph))
    let((cvInstTerms graphNodes)
        cvInstTerms = foreach(mapcan instTerms graph->cv~>instances~>
            instTerms instTerms)
        graphNodes = foreach(mapcar cvInstTerm cvInstTerms
            list(centerBox(dbTransformBBox(car(car(cvInstTerm~>term~>pins)
                ~>figs)~>bBox cvInstTerm~>inst~>transform)) cvInstTerm)
        )
        graphNodeSorted = LUDASortcarList(graphNodes)
        setGraphNodes(graph graphNodeSorted)
    )
)

defmethod(LUDABuildGraph ((graph ConnectivityGraph))
    let((instTermCurrent pinFigCenterCurrent instTermNext pinFigCenterNext)
        for(i 1length(graph->nodes)
            instTermCurrent = cadr(nthelem(i graph->nodes))
            pinFigCenterCurrent = car(nthelem(i graph->nodes))
```

```
27              instTermNext = cadr(nthelem(i+1 graph->nodes))
28              pinFigCenterNext = car(nthelem(i+1 graph->nodes))
29              graph->adjList[instTermCurrent] = append(graph->adjList[
                    instTermCurrent] setof(instTerm instTermCurrent~>inst~>
                    instTerms instTerm!=instTermCurrent))
30              when(instTermNext && LUDACartesianDist(pinFigCenterCurrent
                    pinFigCenterNext) < graph->tolerance
31                  when(!member(instTermNext graph->adjList[instTermCurrent])
32                      graph->adjList[instTermCurrent] = append1(graph->adjList
                            [instTermCurrent] instTermNext)
33                  )
34                  when(!member(instTermCurrent graph->adjList[instTermNext])
35                      graph->adjList[instTermNext] = append1(graph->adjList[
                            instTermNext] instTermCurrent)
36                  )
37              )
38          )
39      )
40  )
41
42  defmethod(LUDAPrintGraphList ((graph ConnectivityGraph))
43      let(()
44          foreach(key graph->adjList
45              printf("%s:%s ->" key~>inst~>name key~>name)
46              foreach(instTerm graph->adjList[key]
47                  printf(" %s:%s, " instTerm~>inst~>name instTerm~>name)
48              )
49              printf("\n")
50          )
51      )
52  )
```

6.4 Closure

Closure is a SKILL++ language construct that was enabled because of lexical scoping feature of SKILL++ compared to SKILL. To put is simply, a closure is a construct consist of function definitions and some free variable bind when the closure is defined. These free variables can be access by the functions even outside of a closure function's scope. To give you a closure example, Queue (FIFO) data structure is implemented as a closure as shown in Code 6.5. `QueueData` is closure data on which closure functions act. The function closures are as follows:

- `addItem`: This would add a new element to the end of the queue
- `removeItem`: This would remove a element from the beginning of the queue
- `printContent`: This would print the content of the queue
- `setUsingArray`: This would set the content of the queue based on the passed array argument.

As is shown in Code 6.5, the return value of the `LUDAMakeQueue` is a `lambda` function whose only argument is a string based on that a specific internal closure function gets returned. The rest of functions, `LUDAAddToQueue, LUDARemoveFromQueue, LUDAAddToQueue,LUDASetQueueUsingArray, LUDAPrintQueue`, are wrapper functions providing a cleaner way to access internal closure functions.

Code 6.5 Queue data structure implementation using SKILL++ closure construct

```
1  procedure(LUDAMakeQueue(initialData)
2      let((
3          (QueueData initialData)
4          )
5          procedure(addItem(data)
6              QueueData = append1(QueueData data)
7          )
8          procedure(setUsingArray(arrayData)
9              QueueData = arrayData
10         )
11         procedure(removeItem()
12             let((QueueHead)
13                 QueueHead = car(QueueData)
14                 QueueData = cdr(QueueData)
15                 QueueHead
16             )
17         )
18         procedure(printContent()
19             printf("%L\n" QueueData)
20         )
21         lambda((method)
22             case(method
23                 ("addItem" addItem)
24                 ("setUsingArray" setUsingArray)
25                 ("removeItem" removeItem)
26                 ("printContent" printContent)
27                 (t nil)
28             )
29         )
30     )
31 )
32
33 procedure(LUDAAddToQueue(queueInst item)
34     let((
35         (addMethod queueInst("addItem"))
36         )
37         addMethod(item)
38     )
39 )
40
41 procedure(LUDARemoveFromQueue(queueInst)
42     let((
43         (removeMethod queueInst("removeItem"))
44         )
45         removeMethod()
```

```
    )
)

procedure(LUDASetQueueUsingArray(queueInst arrayData)
    let((
        (setUsingArrayMethod queueInst("setUsingArray"))
        )
        setUsingArrayMethod(arrayData)
    )
)

procedure(LUDAPrintQueue(queueInst)
    let((
        (printContentMethod queueInst("printContent"))
        )
        printContentMethod()
    )
)
;Example
myQ = LUDAMakeQueue(list("First" "Second" "Third"))
; Add "Fourth" to the end of myQ
LUDAAddToQueue(myQ "Fourth")
; Extract myQ's first element
elem = LUDARemoveFromQueue(myQ); elem = "First"
; Print the myQ elements ("Second" "Third" "Fourth")
LUDAPrintQueue(myQ)
```