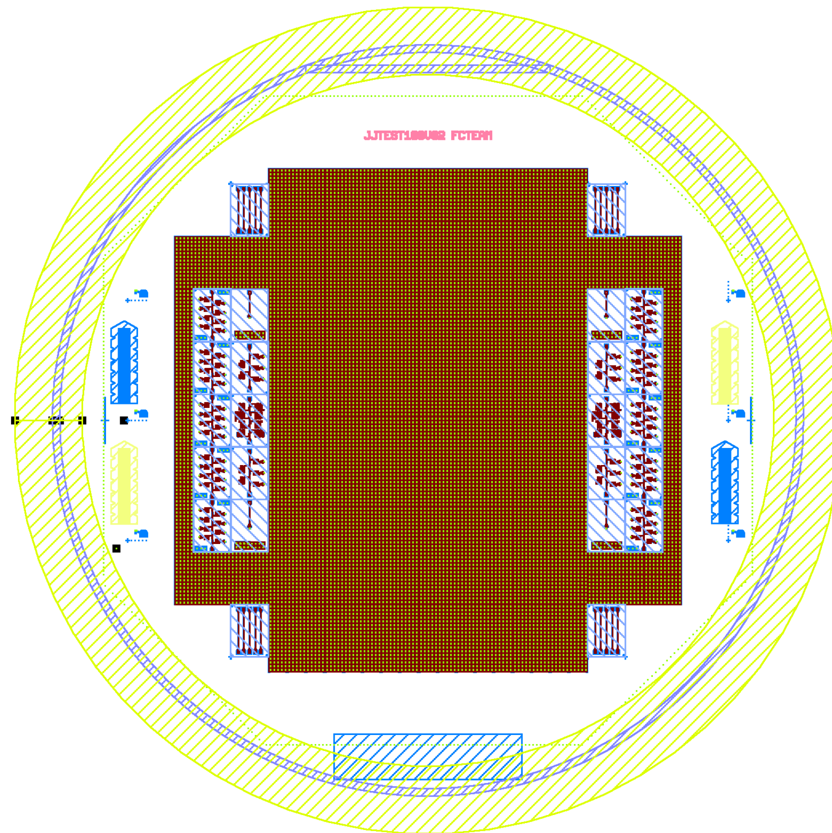
CHALMERS



Utveckling av ett Process Design Kit för supraledande kvantprocessorer

Ett projekt för att effektivisera designfasen av kvantchip

Kandidatarbete vid mikroteknologi och nanovetenskap

Arik Ben-Shabat, Nickoo Sadeghi, Carl Svensson

INSTITUTIONEN FÖR MIKROTEKNOLOGI OCH NANOVETENSKAP

CHALMERS TEKNISKA HÖGSKOLA

Göteborg, Sverige 2025

www.chalmers.se

Kandidatarbete 2025

Utveckling av ett Process Design Kit för supraledande kvantprocessorer

Ett projekt för att effektivisera designfasen av kvantchip

Development of a Process Design Kit for Superconducting Quantum Processors

A project to facilitate the design phase of quantum chips

Arik Ben-Shabat, Nickoo Sadeghi, Carl Svensson

Institutionen för mikroteknologi och nanovetenskap
Kvantteknologi
MCCX11-VT25-12
Chalmers tekniska högskola
Göteborg, Sverige 2025

Utveckling av ett Process Design Kit för supraledande kvantprocessorer
Ett projekt för att effektivisera designfasen av kvantchip
Arik Ben-Shabat, Nickoo Sadeghi, Carl Svensson

© Arik Ben-Shabat, Nickoo Sadeghi & Carl Svensson, 2025.

Handledare: Robert Renhammar & Andreas Nylander, forskare på institutionen för mikroteknologi och nanovetenskap
Examinator: Per Lundgren, studierektor och biträdande professor inom elektronikmaterial på institutionen för mikroteknologi och nanovetenskap

Kandidatarbete 2025
Institutionen för mikroteknologi och nanovetenskap
Kvantteknologi
MCCX11-VT25-12
Chalmers tekniska högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslag: Visualisering i KLayout av en prototyp ämnad att hamna på en skiva
Omslag upphovsrätt: Andreas Nylander

Typsatt i LaTeX
Göteborg, Sverige 2025

Abstract

This report describes the development of an existing Process Design Kit (PDK) for superconducting quantum processors. The purpose of the project is to facilitate the design phase of quantum chips. The process includes the development of parametric cells in Python, design rule checking in Ruby and documentation of the PDK. Development is carried out iteratively by addressing issues on GitLab. Version control is managed by Git and hosted on GitLab.

Part of the work has consisted of developing parametric cells in Python integrated with KLayout, including support for absolute and relative coordinates for coplanar waveguides and interactive handles for ”lumped capacitor”. Additionally, a parameterized test structure for Josephson junctions has been introduced, automating measurement processes and replacing previously manual procedures. DRC scripts in Ruby have been implemented to ensure compliance with design rules. DRC checks overlap, direction and length of Josephson junctions, as well as verifying galvanic connections. Finally, general improvements have been made to the PDK, including automatic version management, dynamic step length calculation, autorun scripts for checking the version of KLayout and PDK, the ability to reload PCell modules without restarting KLayout and preparing for releasing the PDK as open source.

These efforts have increased the automation and reliability of the PDK, freeing up time for continued optimization and expansion of the tool's functionality. Recommendations for further development include the addition of more PCells and more advanced design rules to further enhance both capability and quality.

Keywords: PDK, PCells, DRC, Python, Ruby, KLayout, Sphinx, Semiconductor Fabrication, Superconducting Quantum Processors, Quantum Technology

Sammanfattning

I denna kandidatuppsats beskrivs vidareutvecklingen av ett Process Design Kit (PDK) för supraledande kvantprocessorer, med målet att automatisera och effektivisera designfasen av kvantchip. Processen bygger på utveckling av parametriska celler i Python, designregelverifiering i Ruby och dokumentation av hela PDK:t. Utvecklingen genomförs iterativt genom bearbetning av ärenden på GitLab. Versionshantering sker via Git och lagras på GitLab.

Arbetet har delvis bestått av utvecklingen av parametriska celler i Python integrerat med KLayout, där stöd för både absoluta och relativa koordinater för koplanära vågledare samt interaktiva handtag för ”lumpade kondensatorer” har implementerats. Därtill har en parametriserad teststruktur för Josephson-övergångar införts vilken automatiserar mätprocesser och ersätter tidigare manuella rutiner. För att garantera att vissa designregler uppfylls implementeras DRC-skript i Ruby som kontrollerar överlapp, riktning och längd hos Josephson-övergångar samt verifierar galvaniska kontakter. Slutligen införs generella förbättringar i PDK:t, inklusive automatisk versionshantering, dynamisk steglängdsberäkning, autorun-skript för versionskoll för KLayout och PDK:t, möjlighet att omladda PCell-moduler utan omstart av KLayout och förberedelser för att släppa PDK:t som öppen källkod.

Genom dessa åtgärder har automationen och tillförlitligheten i PDK:t ökat, vilket frigör tid för fortsatt optimering och utökning av verktygets funktionalitet. Vidare rekommenderas en utvidgning med fler PCeller och fördjupade designregler för att ytterligare förbättra både kapacitet och kvalitet.

Nyckelord: PDK, PCeller, DRC, Python, Ruby, KLayout, Sphinx, halvledartillverkning, supraledande kvantprocessorer, kvantteknologi

Förord

Vi vill rikta ett stort tack till våra engagerade handledare, Andreas Nylander och Robert Rehammar, för deras värdefulla vägledning och stöd under projektets gång. Arbetet har inte bara fördjupat vår förståelse för mjukvaruutveckling i stort, utan även gett oss en inblick i nanofabrikation och kvantteknologi.

Arik Ben-Shabat, Nickoo Sadeghi & Carl Svensson, Göteborg maj 2025

Akronymer

Nedan är en lista av akronymer som förekommer i rapporten.

CPW **Coplanar Wave Guide**
Koplanära vågledare är plana strukturer där en signalledare och två jordplan löper parallellt på samma yta av ett dielektriskt underlag. De används som medium för transmission av kvantinformation mellan komponenter i supraledande kvantkretsar.

DRC **Design Rule Checking**
Designregelverifiering är en process där designens layout kontrolleras mot specifika designregler för att säkerställa att den kan tillverkas korrekt.

JJ **Josephson Junction**
En Josephsonövergång är en kvantmekanisk komponent som bygger på Josephsoneffekten, där ett tunt isolerande skikt mellan två supraledare tillåter överföring av elektroner utan spänningsfall.

LVS **Layout Versus Schematic**
En teknik för att jämföra en fysisk layout med dess schematiska representation för att säkerställa att de matchar.

PCell **Parametrisk Cell**
En cell skapad i en layout som kan modifieras via ändring av dess parametrar för att skapa olika varianter av samma cell.

PDK **Process Design Kit**
Verktyg som underlättar chipdesign. Består utav ett teknologipaket innehållandes parametriska celler (PCell), designregelverifiering (DRC), layout-mot-schema tester (LVS) och en semitredimensionell vy (2.5D).

QT **Quantum Technology**
En forskargrupp inom kvantteknologi på institutionen för mikroteknologi och nanovetenskap på Chalmers.

Innehåll

Lista av akronymer **ix**

Figurer **xiii**

1 Introduktion **1**
1.1 Bakgrund 1
1.2 Projektbeskrivning 2
1.2.1 Syfte 2
1.2.2 Avgränsningar 2

2 Teori **3**
2.1 Process Design Kit (PDK) 3
2.2 KLayout 3
2.3 Parametriska celler (PCeller) 4
2.4 Designregelverifiering (DRC) 4
2.5 Layout-mot-schema tester (LVS) 4
2.6 2,5D-vy 5
2.7 Litografi 5
2.8 Koddokumentation 5

3 Metod **7**
3.1 Utveckling av PCeller 7
3.2 Utveckling av DRC 9
3.3 Dokumentation via Sphinx 9

4 Resultat och diskussion **13**
4.1 PCeller 13
4.1.1 Valmöjlighet mellan relativa och absoluta koordinater 13
4.1.2 Ökad flexibilitet med handtag för ”Lumped capacitor” 14
4.1.3 Visuell förbättring av ”Lumped inductor” 15
4.1.4 Skapandet av ”JJ-tester” 15
4.2 DRC 16
4.2.1 Verifiering av Josephsonövergången 16
4.2.2 Verifiering av Josephsonövergångens sammankoppling till kringliggande komponenter 18
4.3 Generella förbättringar 20
4.3.1 Tillägg av parameter som sparar PDK-versionen för varje PCell 20

4.3.2 Smartare uppdatering av ”step-size” i hjälp-struktur 20
4.3.3 Automatisk versionskoll och laddning av teknologipaketet 21
4.3.4 Automatisk omladdning av moduler 21
4.3.5 Omstrukturering av teknologipaketet 22
4.4 Dokumentation 23
4.4.1 Automatisk koddokumentation via Sphinx 23
4.4.2 Uppdatering av README 23

5 Slutsatser 25

Litteratur 27

A Appendix I
A.1 PP_example.py I
A.2 Lumped Capacitor radie-logik III
A.3 Lumped Inductor borttagning av låda III
A.4 grain.xml III

Figurer

3.1 Figur som visar hur man får ut en PCells koordinater för dess former från dess parametrar. Bilderna är skapade med draw.io. 8

4.1 Figur som visar skillnaden mellan absoluta och relativa koordinater. Bilderna är skapade med draw.io. 14
4.2 PCellen "lumped capacitor" med tillägg i form av handtag. Bilderna är skapade i KLayout. 14
4.3 PCellen "lumped inductor" med och utan kontur av jord. Bilderna är skapade i KLayout. 15
4.4 En Josephsonövergångs-testare. Bilden är skapad i KLayout 16
4.5 En korrekt och tre felaktigt implementerade Josephsonövergångar. Bilderna är skapade i KLayout. 17
4.6 DRC-rapport för Josephson-övergången. Bilden visar det felmeddelande som matas ut då felaktig direktionalitet hos den vertikala armen föreligger. Bilden är skapad i KLayout. 18
4.7 Josephsonövergång (orange) och dess sammankoppling till de kringliggande komponenterna PICT (turkos), Patch (grön) samt jordplanen (vit och röd). Bilden är skapad i KLayout. 18
4.8 Felmarkerade lager vid körning av DRC-programmet för verifiering av Josephsonövergångens sammankoppling till Patch, PICT och jordplanen. Bilden är skapad i KLayout. 19
4.9 Motsvarande DRC-rapport som fångar samtliga fel relaterade till den uteblivna galvaniska kontakten mellan Josephsonövergången och jordplanet. Bilden är skapad i KLayout. 19
4.10 Infomationsrutan från autorunskriptet 21
4.11 Utdrag från de genererade HTML-filerna skapade med Sphinx. 23

1
Introduktion

Detta kandidatarbete berör vidareutvecklingen av ett Process Design Kit (PDK) för supraledande kvantprocessorer i layoutverktyget KLayout. Genom utveckling av parametriska celler (PCeller) och designregelverifiering (DRC) är målet att öka graden av automation och således underlätta designfasen av kvantchip på Chalmers. Detta kompletteras med underhåll och optimering av teknologipaketet i vilken PDK:t är integrerad, samt dokumentation av PDK:ts användning och vidareutveckling.

1.1 Bakgrund

Forskningen inom kvantdatorer befinner sig i en intensiv och omvälvande fas. Den har beskrivits som en kapplöpning mellan olika aktörer på marknaden, och samtidigt lett till nya framsteg [1]. Ett framstående exempel är Googles nyligen lanserade kvantchip Willow, som med sina 105 supraledande kvantbitar uppvisar en enormt ökad teoretisk beräkningshastighet och tyder på en överlägsenhet jämfört med dagens superdatorer [2]. En kvantdator kännetecknas inte bara av antalet kvantbitar som mått på prestanda, utan också av i vilken grad de lyckas upprätthålla stabilitet och bevara kvantinformationen över tid [3]. Det sistnämnda är en utmaning eftersom de störs av sin omgivning och som följd blir instabila. Detta leder till kvantbitsfel, som i varierande grad kan åtgärdas med kvantfelsrättande koder, vilket är essentiellt vid uppskalning till fler kvantbitar [4].

Inom den experimentella kvantteknologi-gruppen, på engelska Quantum Technology (QT) bedrivs forskning på supraledande kvantdatorer och kvantchip, samma typ som tidigare nämnda Willow. QT designar dessa i olika layoutverktyg och tillverkar dem sedan i Chalmers renrum genom litografi, en process där ljus används för att överföra mönster till ett materialskikt på ett chip [5]. I dagsläget har de utvecklat ett kvantchip med 25 kvantbitar [6], som mäter $14,3 \times 14,3$ mm^2 och har 75 anslutningar, ämnat att skalas upp till 40 kvantbitar inom ett par år och slutligen 100 stycken år 2030 [6]. Utöver detta utforskas nya implementationer av kvantbitar och hur dessa kan kopplas samman på nya sätt. I takt med att chipdesignen växer i komplexitet och antalet kvantbitar ökar, uppstår ett växande behov av struktur, standardisering och verktyg för effektivt samarbete inom forskningsgruppen. För att möta detta behov har QT i layoutverktyget KLayout tagit fram ett PDK – ett verktyg som består av PCeller, DRC, Layout Versus Schematic (LVS) och 2,5D-vy.

1.2 Projektbeskrivning

Projektet består av att i samarbete med QT vidareutveckla PDK:t och skräddarsy det för deras behov. Detta innebär förbättring av befintliga PCeller samt implementering av ofta använda PCeller. Projektet innefattar även att implementera DRC samt dokumentera hur PDK:t används och hur det kan vidareutvecklas. PDK:t är en del av ett teknologipaket [7]. Teknologipaketet innehåller dessutom tillgängliga layoutlager och deras respektive syften, vilket också innebär att uppgiften omfattar underhåll och optimering av paketet. Slutligen dokumenteras hela utvecklingsprocessen, inklusive de verktyg som används och annan relevant information. Målet med arbetet är att vidareutveckla det befintliga PDK:t till en mer komplett version.

1.2.1 Syfte

Syftet med projektet är att förenkla designfasen av kvantchip och stärka kontrollen av automation genom att vidareutveckla PCeller och implementera DRC. Detta förväntas minska behovet av manuell handpåläggning och därmed öka effektiviteten i QT:s arbete. DRC är dessutom avgörande för att säkerställa att en design kan realiseras i praktiken, vilket också är tid- och resursbesparande. Resultatet av projektet kommer göra det möjligt för QT, med det nya PDK:t, att i högre grad fokusera på sin kärnverksamhet och därmed stärka sin position som en ledande aktör inom kvant-revolutionen.

1.2.2 Avgränsningar

Ett PDK kan utvecklas i stor utsträckning. För att avgränsa projektet beslutade vi att fokusera på att förbättra den befintliga koden, identifiera och åtgärda buggar samt utveckla nya PCeller och DRC-program. Dokumentationen syftar till att visa hur PDK:t används och dess funktioner men även hur PDK:t kan vidareutvecklas för att möta framtida behov.

Vad gäller utvecklingen valde vi att implementera de PCeller som QT ofta återanvänder, då detta resulterar i en större tidsbesparing för dem. Detsamma gäller implementeringen av DRC. Eftersom scheman – som kan liknas vid en karta över en elektronisk krets som visar vilka komponenter som ingår och hur de är ihopkopplade – varken används av QT i dag eller planeras att införas i framtiden, har vi inte avsatt tid för LVS. Anledningen till detta är att kvantchip och deras scheman inte följer samma universella standard som exempelvis klassiska elektriska kretsar, vilket gör att LVS inte är relevant i detta sammanhang. Även 2,5D-vyn har uteslutits, eftersom den enbart erbjuder ett alternativt sätt att visualisera olika lager i en semi-3D-vy istället för en 2D-vy, vilket inte bidrar till en förbättring av arbetsflödet.

2

Teori

I detta avsnitt beskrivs först vad ett PDK är, vad det består av samt dess relation till teknologipaketet. Därefter förklaras de ingående delarna i PDK:t mer fördjupat. Avsnittet avslutas med att förklara litografi på en översiktlig nivå samt hur vi gått tillväga för att dokumentera vårt arbete.

2.1 Process Design Kit (PDK)

Ett PDK är ett verktyg som består av PCeller, DRC, LVS och en 2,5D-vy. Verktyget möjliggör effektiv design av integrerade kretsar och elektroniska chip enligt en specifik tillverkningsprocess [8]. Fördelarna är en standardiserad och strömlinjeformad process med vilken man kan framställa chip mer tillförlitligt och kostnadseffektivt. Verktyget är inbäddat i ett teknologipaket. I teknologipaketet återfinns lagerkonventionen.

Teknologipaketets lagerkonvention innefattar även maskdefinitioner som möjliggör en mer exakt kontroll av strukturer inom och mellan lager - exempelvis genom att addera, subtrahera eller på annat sätt kombinera lager vid layout och maskframställning. Detta för att möjliggöra för olika delsteg inom litografiprocesser. Lagerkonventionen är tätt sammankopplad till PCellerna då varje cell skapas i ett förutbestämt lager. Det finns även designerade lager som PCellerna automatiskt renderas i ifall det skett en bugg eller de inte registrerats rätt när de skapats.

För att säkerställa att designen är både korrekt och tillverkningsbar ingår även designregler och verifieringssteg genom DRC och LVS. DRC kontrollerar att geometrin mellan olika lager följer givna regler, exempelvis minsta avstånd mellan komponenter och otillåtna överlapp av lager. LVS verifierar att layouten stämmer överens med den logiska designen, så att alla elektriska kopplingar fungerar som avsett. Tillsammans hjälper dessa verktyg användaren att tidigt upptäcka fel utan krav på fullständig insikt i alla tillverkningsdetaljer. Sammanfattningsvis utgör PDK:t ett centralt verktyg som möjliggör tillförlitlig och effektiv chipdesign.

2.2 KLayout

KLayout är ett kostnadsfritt layoutverktyg med öppen källkod, utvecklat av Matthias Köfferlein, för visning och redigering av layoutfiler i GDS- och OASIS-format [9]. Det kan användas både för att öppna befintliga designer och för att skapa nya

från grunden. Programmet är plattformsoberoende och fungerar på Windows, Linux och macOS.

KLayout erbjuder ett kraftfullt visningsläge som bland annat stödjer överlagring och möjligheten att arbeta med flera layouter samtidigt i ett och samma fönster. Det har även stöd för PCeller, DRC, LVS samt en 2,5D-vy för bättre visualisering av strukturer. Funktionaliteten förstärks ytterligare genom att KLayout har inbyggda tolkar för både Python och Ruby, vilket möjliggör skriptning och automatisering direkt i verktyget.

2.3 Parametriska celler (PCeller)

En PCell är en programmerbar cell som genereras och tillåter designern att välja parametrar av intresse [10]. De olika parametrarna justeras för att generera varierade former och funktionalitet. Detta är essentiellt i en PCell då det låter designern att specificera dimensioner och andra karaktäristiska egenskaper hos cellen när den genereras, istället för att designa varje variant från början [11]. PCellernas komplexitet varierar och designas utifrån behov. PCellerna designas i Python och vid instansiering justeras dessa parametrar i KLayout.

2.4 Designregelverifiering (DRC)

Designregelverifiering är ett avgörande steg i chipdesignen som säkerställer att layouten följer en uppsättning regler, exempelvis avstånd mellan ledningar och korrekt placering av komponenter, för att undvika problem som kortslutningar och brus [12]. DRC kontrollerar både att enskilda komponenter uppfyller kravspecifikationerna och att deras kopplingar till omgivande komponenter är korrekta. Genom att köra DRC tidigt i designprocessen kan fel identifieras och åtgärdas innan de leder till kostsamma problem i senare stadier. Det bidrar alltså inte bara till att minska risken för tillverkningsfel, utan sparar även tid och resurser [13]. Dessutom fungerar DRC som ett pedagogiskt verktyg: designaren får direkt återkoppling när något i layouten bryter mot reglerna, vilket möjliggör en successiv inlärning av tillverkningskrav. På så sätt bidrar DRC till en effektivare och mer produktiv designcykel. Antalet designregler varierar mellan olika typer av chip, men en tumregel är att det ökar avsevärt i takt med att chipstorleken minskar [14]. Beroende på tillverkningsteknik kan antalet regler variera från färre än hundra till ett par tusen [15].

2.5 Layout-mot-schema tester (LVS)

LVS är en annan viktig del av layoutverifiering, då det säkerställer att kretsen överensstämmer med dess schematiska representation [16]. Netlists beskriver komponenternas kopplingar, vilket resulterar i att det är relationen mellan komponenterna, och inte deras exakta positioner, som är viktig. Detta kan liknas vid graf-isomorfier, där

komponenter utgör noder och kopplingar kanter [17]. En isomorfism föreligger om två grafer har samma topologi, det vill säga samma kopplingar, men inte nödvändigtvis samma utseende. LVS verifierar funktionaliteten och förhindrar produktionsfel tidigt i designfasen, vilket förbättrar tid- och kostnadseffektiviteten, precis som med DRC [8].

2.6 2,5D-vy

KLayout erbjuder en 2,5D-vy som ger en semi-3D-visualisering av layouten [18]. Istället för en fullständig 3D-modell extruderas lagren vertikalt med en specificerad tjocklek. Denna vy gör det möjligt att visualisera hur de elektriska ledningarna löper eller jämföra de vertikala dimensionerna för olika delar av lagerstrukturen. Genom detta kan användaren få en bättre förståelse för layoutens uppbyggnad, särskilt när det gäller att upptäcka problem med ledningarnas placering och vertikala relationer mellan olika lager. 2.5D-vyn programmeras på liknande sätt som DRC, då de båda läser in och modifierar samt matar ut lager.

2.7 Litografi

Litografi är en tryckteknik som bygger på att överföra mönster från en plan yta. Ordet härstammar från grekiskan och översätts till ”sten” (lithos) och ”skriva” (graphē). Ursprungligen byggde tekniken på att mönster ristades i sten och överfördes till papper genom tryck [19]. Fotolitografi är en modern tillämpning av litografi och används inom halvledartillverkning för att skapa mikroskopiska mönster på kiselskivor med hjälp av ljus. Fotolitografi kan därmed förstås som ”processen av att skriva med ljus” [5].

Fotolitografiprocessen inleds med att ett ljuskänsligt material, kallat fotoresist, appliceras på en kiselskiva där mikrochippen tillverkas [20]. Skivan exponeras därefter för ljus genom en fotomask som innehåller det önskade mönstret. De ljusbelysta delarna av fotoresistet genomgår en kemisk förändring, vilket möjliggör framkallning av mönstret. De exponerade delarna avlägsnas, vilket blottar det underliggande substratet. Dessa öppna områden kan sedan etsas bort eller beläggas med nytt material. Slutligen tas det återstående fotoresistet bort, och mönstret är därmed permanent överfört till skivan. För mindre komponenter, där behovet på högre upplösning är stort, används andra metoder såsom elektronstrålelitografi [21].

PDK:ts lagerkonvention korresponderar till olika litografisteg. Efter att chippen designats fabriceras de i renrummet och resulterar i färdiga kvantchip.

2.8 Koddokumentation

I tekniska utvecklingsprojekt av det här slaget, där ett komplext verktyg som ett PDK utvecklas och underhålls, är tydlig och välstrukturerad dokumentation avgö-

rande. Den spelar en central roll både för att nya användare snabbt ska kunna sätta sig in i arbetet och för att möjliggöra vidareutveckling samt säkerställa spårbarhet i det som redan har skapats.

Dokumentation kan utformas på många olika sätt, men i det här projektet för QT har vi valt att använda verktyget Sphinx. Detta är ett verktyg som QT påbörjat arbeta med, och som projektet nu vidareutvecklar. Valet föll på Sphinx eftersom det erbjuder ett effektivt sätt att skapa, organisera och underhålla teknisk dokumentation.

Sphinx är ett verktyg som genererar dokumentation genom att tolka och konvertera filer skrivna i reStructuredText (.rst) eller Markdown (.md) [22]. Med Sphinx är det möjligt att producera dokumentation i form av HTML-filer, PDF-filer, hemsidor, bloggar med mera [23]. Verktyget är särskilt användbart för att skapa dokumentation för Python-kataloger, eftersom Sphinx är en Python-baserad applikation.

Sphinx konfigureras via en inställningsfil som låter användaren styra hur verktyget tolkar och hanterar den Python-kod som ska dokumenteras. Genom olika tillägg kan Sphinx automatiskt identifiera och tolka klasser, funktioner, dokumentationssträngar och typangivelser. För att detta ska fungera korrekt krävs att koden är välstrukturerad och tydligt dokumenterad. Det innebär bland annat att varje klass och funktion ska ha en doksträng där argumentens och returvärdets datatyper anges.

Konfigurationsfilen styr även delvis skapandet av de .rst-filer som representerar varje Python-fil i katalogen. Det är i .rst-filerna som innehållet och strukturen för sidan definieras.

I detta projekt är målet att dokumentera det befintliga PDK:t och säkerställa en automatiserad dokumentationsprocess som kontinuerligt uppdateras vid förändringar i PDK:t. Detta kan exempelvis uppnås genom att använda GitLabs CI/CD-skript, vilket automatiskt uppdaterar både .rst-filer och de genererade HTML-sidorna.

3

Metod

Arbetsprocessen har innefattat hämtning och hantering av GitLab-ärenden från QT, vilka har bearbetats parallellt för att täcka samtliga delar av PDK:t. Versionshantering har genomförts i Git. Arbetet har praktiskt sett bestått av mjukvaruutveckling av PDK:ts teknologipaket, med särskilt fokus på implementeringar som möjliggör automatisering, ökad effektivitet och fortsatt vidareutveckling. Detta har genomförts med hjälp av Visual Studio Code och KLayout.

En central aspekt av projektet har varit att utforma korrekt och tydlig dokumentation. Allteftersom bearbetade ärenden färdigställts har de noggrant dokumenterats i GitLab. Vidare har Sphinx använts för att generera en omfattande dokumentation av PDK:t i sin helhet.

3.1 Utveckling av PCeller

I KLayout finns flera sätt att skapa PCeller, och dessa kan programmeras antingen i Ruby eller Python. Vi har valt att använda Python. För att effektivt designa en PCell är det praktiskt att först rita upp designen. I PCellen definieras de lager cellen ska tillhöra och därmed även vilka lager den kommer att renderas i. En PCell kan generera polygoner i flera lager.

Vid skapandet av PCeller använder vi ärvda metoder från superklassen `PPCellDeclarationHelper`, ett tidigare etablerat element i PDK:t som vi blev tilldelade i början av projektet. `PPCellDeclarationHelper` ärver i sin tur metoder från `PCellDeclarationHelper` som är en del av KLayouts egna paket `pya`. `Pya` är KLayouts integrerade Python-API och innehåller samtliga metoder för att generera PCeller i layouten. De vanligaste metoderna som används är följande:

`__init__(self, vp):`	Här definieras parametrarna och lagren som designen använder sig av.
`display_text_impl(self):`	Returnerar ett beskrivande namn på PCellen som är synlig i layouten samt cell-menyn i KLayout.
`coerce_parameters_impl(self):`	Tvingar ändring av en parameters värde i PCell-parameters-menyn i KLayout.

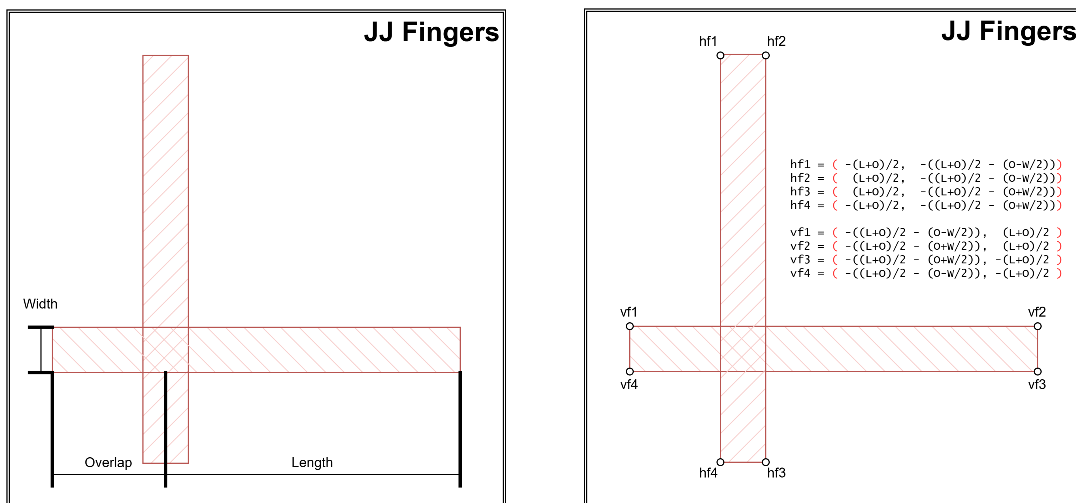
`transform_from_shape_impl(self):` Sätter PCellens centrum till utgångspunkt för att bestämma transformationer.

`produce_shapes(self):` Den metod som bygger ihop cellen och sedan ritar ut den i en layout.

Syftet med `PPCellDeclarationHelper` är att samla gemensam funktionalitet som används av flera PCeller.

KLayout erbjuder en integrerad utvecklingsmiljö, men det är även möjligt att öppna och redigera filerna i externa program. I detta projekt har vi huvudsakligen använt utvecklingsmiljön Visual Studio Code (VS Code) för detta ändamål.

Vid skapandet av PCeller är det vanligt att utgå från `PP_example.py` som mall (A.1), där de vanligaste metoderna redan är definierade och kan anpassas efter behov för den aktuella PCellen. Det första steget är att identifiera de parametrar som krävs för att implementera PCellen. Här definieras även de lager som PCellen ska använda. Därefter identifieras de koordinater och punkter som behövs för att skapa designen med hjälp av de tillgängliga parametrarna. Samtliga koordinater definieras med PCellens mittpunkt som origo.



(a) Parametrarna för att skapa fingrarna i en Josephsonövergång

(b) Koordinaterna för fingrarna i en Josephsonövergång från parametrarna

Figur 3.1: Figur som visar hur man får ut en PCells koordinater för dess former från dess parametrar. Bilderna är skapade med draw.io.

När PCellen är definierad måste den läggas till i biblioteket för att kunna användas samt renderas i KLayout. Det första steget är att importera den nya PCellklassen från PCellsfilen i huvudfilen `Python parametric cells.lym`.

```
from QTQC.PP_example import example
```

Sedan måste PCellen registreras i layouten. Det görs längre ner i huvudfilen.

```
self.layout().register_pcell("Example PCell", PP_example(self.vp))
```

Nu är PCellen registrerad i biblioteket och kan nås från KLayout.

3.2 Utveckling av DRC

Inbyggt i KLayout finns ett DRC-verktyg som är tillgängligt via verktygsfältet. Där kan användaren implementera sina designregler i programspråket Ruby. Programflödet är följande:

- Skapa en DRC-rapport med kommandot `report(DRC report)`. Det är i denna resultatet ska skrivas till samt presenteras.
- Läs in de lager som berör designreglerna. Dessa återfinns i teknologipaketet. Tilldela dem till variabler med lämpliga namn.
- Använd variablerna som operander i funktioner som utgör designreglerna. Specificera dessutom lämpliga felmeddelanden och till vilket lager felen ska markeras i, för enklare detektering. Dessa presenteras vid körning, i DRC-rapporten.
- Kör skriptet för att automatiskt skapa och öppna DRC-rapporten.

Om layouten uppfyller samtliga designregler matas inget meddelande ut, och layouten renderas felfritt. Ifall något fel föreligger presenteras det specificerade felmeddelandet i det specificerade lagret. Ofta förekommande verifieringsfunktioner importeras via det Ruby-automatiserade biblioteket och används enkelt genom att mata in de lager man vill utföra kontrollen på, samt ett lämpligt felmeddelande. Inbyggt i de flesta av funktionerna är även illustrationer av det fel som fångas. Ett konkret exempel är att kontrollera att avståndet mellan två lager inte understiger ett visst värde. Om denna regel bryts genereras det specificerade felmeddelandet och avståndet mellan dessa lager markeras i rött i layouten. Mer komplexa regler programmeras genom att kombinera importerade funktioner med varandra eller genom egna implementationer.

3.3 Dokumentation via Sphinx

Nedan följer den steg-för-steg-metod som användes för att skapa dokumentationen för PDK:t med hjälp av Sphinx. Samtliga steg har utförts på en Mac-dator med macOS, vilket kan påverka vissa kommandon och filvägar. Terminalkommandon, sökvägar och miljöinställningar har anpassats därefter.

Sphinx är ett Python-baserat dokumentationsverktyg och installeras via pip, Pyt-

hons pakethanterare som används i terminalen [24]. För att skapa en isolerad miljö för installationer rekommenderas det att först upprätta en virtuell miljö i projektkatalogen genom att köra följande kommandon:

```
$ python -m venv .venv
$ source .venv/bin/activate
(.venv) $  python -m pip install sphinx
```

Därefter skapas en katalog för dokumentationen, benämnd `docs`, med hjälp av följande kommando:

```
$ sphinx-quickstart docs
```

Katalogen placeras i `tech`-katalogen och innehåller de grundläggande filerna som krävs för Sphinx, inklusive konfigurationsfilen `conf.py`. I denna fil ställer användaren in hur Sphinx ska tolka de Python-filer som dokumentationen baseras på. För att Sphinx ska kunna tolka och dokumentera Python-filerna korrekt, måste alla externa moduler som används i projektet installeras via pip. Detta gäller även moduler som `pya` [25], se del 3.1.

```
docs/
|- build/
|- make.bat
|- Makefile
|- source/
   |- conf.py
   |- index.rst
   |- _static/
   |- _templates/
```

Alla `.rst`-filer placeras i `source`-katalogen, medan `build`-katalogen används för att lagra de genererade HTML-filerna. För att skapa dessa HTML-sidor, körs följande kommando från `docs`-katalogen:

```
$ make html
```

HTML-filerna konfigureras därefter utifrån projektets behov och kan sedan öppnas i valfri webbläsare direkt från `build`-katalogen.

För att dokumentationen ska genereras automatiskt från kodens dokstringar krävs att specifika Sphinx-tillägg installeras via pip och läggs till i `conf.py`-filen. Dessa tillägg möjliggör bland annat följande:

- Generering av dokumentation direkt från dokstringar (autodoc).
- Visning av datatyper för funktioners in- och utparametrar.

- Inkludering av källkod i dokumentationen.

Vidare måste filvägen till projektets Python-kataloger anges i `conf.py` för att Sphinx ska kunna navigera korrekt. Det går också att utesluta specifika filer från dokumentationen genom att konfigurera motsvarande inställningar – mer information om detta finns på Sphinx officiella webbplats [26]. I detta projekt dokumenterades Python-filerna för samtliga PCeller, hjälpfiler för skapandet av diverse geometriska former och tillägg samt basklasser och lagerkonventionerna.

Slutligen har HTML-dokumentationens utseende anpassats via en CSS-fil som placerats i `_static`-katalogen (se 3.3). I denna fil har bland annat färgscheman, layout, marginaler och textstorlek justerats för att skapa en mer visuellt tilltalande och lättläst dokumentation.

4

Resultat och diskussion

I följande avsnitt presenteras projektets resultat. För varje resultat ingår en kort diskussion där vi jämför utfallet och bedömer om det uppnår vårt syfte. Under projektets gång har vi använt oss av issues på GitLab för ärendehantering och tilldening av arbetsuppgifter. Vi har även fått uppgifter direkt från våra handledare och andra medarbetare på QT. Resultaten är uppdelade i kategorierna PCeller, DRC, generella förbättringar och dokumentation.

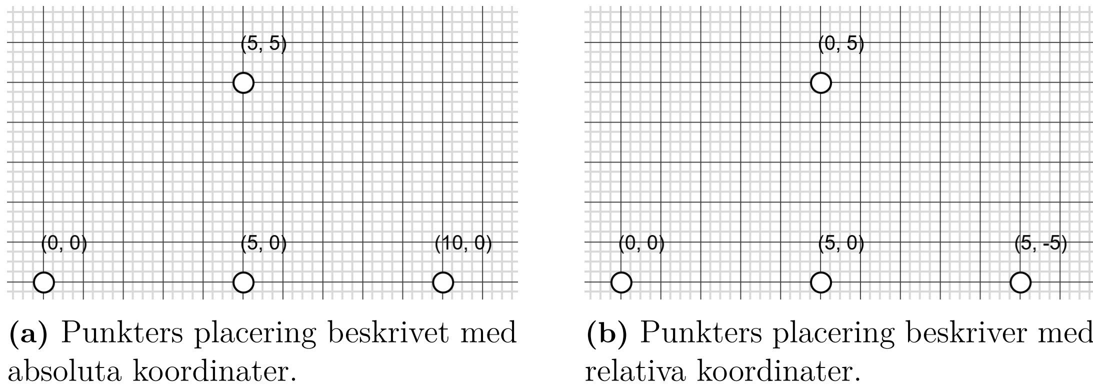
4.1 PCeller

Detta avsnitt presenterar och diskuterar de implementationer som rör PCeller. Detta innefattar dels skapandet av nya PCeller men även förbättringar av redan existerande PCeller.

4.1.1 Valmöjlighet mellan relativa och absoluta koordinater

Tidigare utgick man alltid från en utgångspunkt i PCellen när man använde sig av absoluta koordinater för att rita ut en CPW. Relativa koordinater implementerades för att förenkla designfasen och slippa behöva tänka med koordinaterna från utgångspunkten. Det innebär att X- och Y-koordinaten, istället för att ange punktens avstånd i X- och Y-led från utgångspunkten (4.1a), anger avståndet från föregående punkt (4.1b). För att byta mellan absoluta och relativa koordinater har en checkbox i PCell Parametermenyn lagts till.

Införandet av relativa koordinater i CPW är ett tydligt exempel på hur projektets syfte att förenkla designfasen har uppnåtts. Genom att kunna välja mellan absoluta och relativa koordinater får designern möjlighet att välja den implementation som passar bäst för den aktuella situationen. Det är dock fortfarande viktigt att kunna använda absoluta koordinater, eftersom det möjliggör att flytta en koordinat i mitten av CPW:n utan att påverka de efterföljande koordinaterna. Med relativa koordinater kommer en förflyttning av den andra koordinaten att även påverka den tredje.



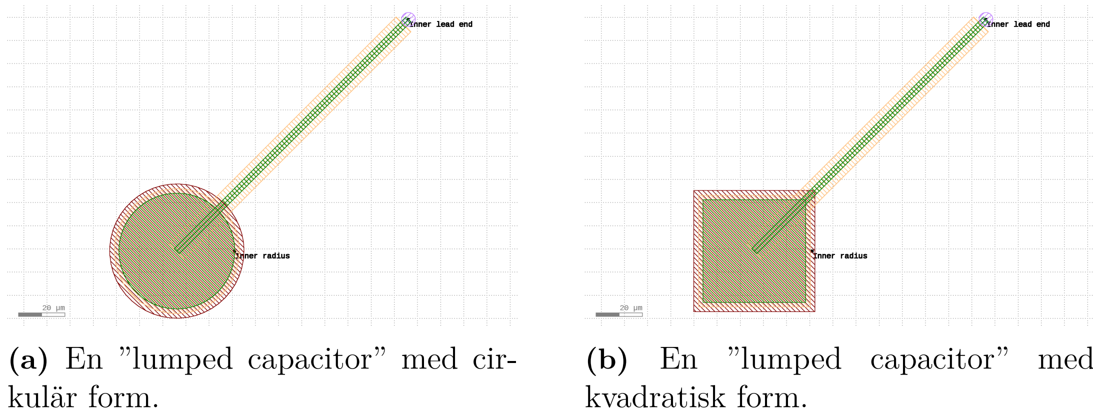
(a) Punkters placering beskrivet med absoluta koordinater.

(b) Punkters placering beskriver med relativa koordinater.

Figur 4.1: Figur som visar skillnaden mellan absoluta och relativa koordinater. Bilderna är skapade med draw.io.

4.1.2 Ökad flexibilitet med handtag för ”Lumped capacitor”

Syftet med detta ärende var att skapa ett handtag som gör det möjligt att dynamiskt justera PCellen ”Lumped capacitor”:s radie. Vanligtvis ändras längder och storlekar till fasta värden i parameterfält i KLayout, vilket inte tillåter användaren att finjustera måtten i förhållande till exempelvis andra PCeller. Med ett handtag kan användaren istället noggrant anpassa PCellens radie utan att behöva uppskatta ett värde och därefter testa sig fram.



(a) En ”lumped capacitor” med cirkulär form.

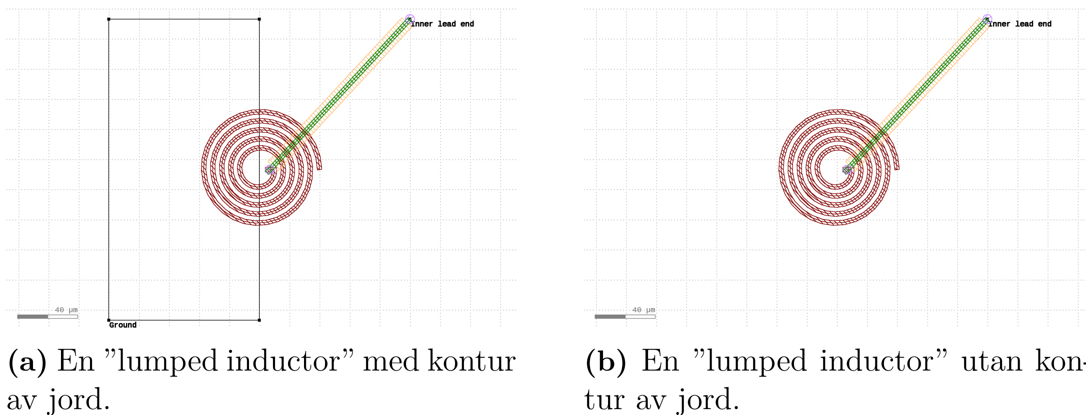
(b) En ”lumped capacitor” med kvadratisk form.

Figur 4.2: PCellen ”lumped capacitor” med tillägg i form av handtag. Bilderna är skapade i KLayout.

Användaren har nu möjlighet att välja om handtaget eller parameterfältet ska användas. Detta görs genom att klicka i en ruta i PCell Parametermenyn. Ett problem som uppstod i samband med detta var att radien inte synkroniserades till parameterfältet när handtaget användes. Detta löstes genom att införa en styrmekanism i metoden `coerce_parameters`, som avgör när radien ska uppdateras via handtaget respektive genom parameterfältet (se bilaga A.2). Därefter justeras värdet på radien i enlighet med de förändringar som görs och synkroniserar med handtaget.

4.1.3 Visuell förbättring av ”Lumped inductor”

Denna PCell har ett valbart tillägg i form av en "jord-struktur kopplad till induktansen. Vid rendering av PCellen visas dock konturen av denna jord ovanpå själva strukturen, vilket gör layouten svåröverskådlig och visuellt rörig. För att lösa detta problem sattes parameterns hidden-egenskap till True, vilket gör att jordens kontur inte längre visas i layouten vid initial rendering. Se bilaga A.3 för koden.



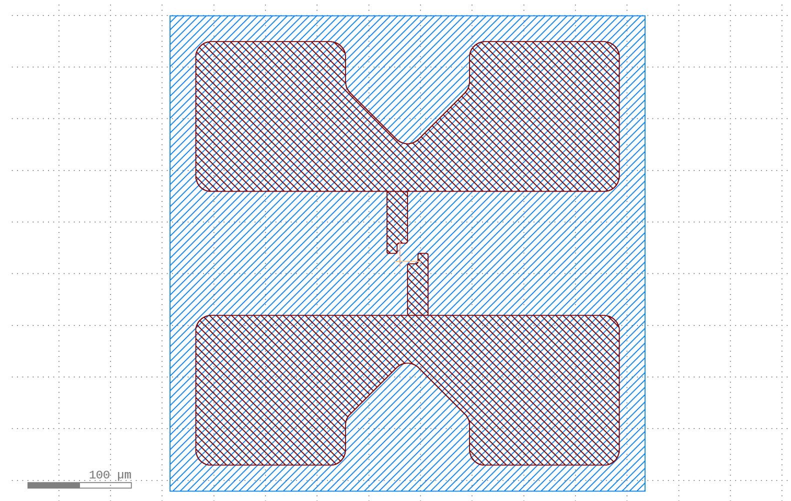
(a) En ”lumped inductor” med kontur av jord.

(b) En ”lumped inductor” utan kontur av jord.

Figur 4.3: PCellen ”lumped inductor” med och utan kontur av jord. Bilderna är skapade i KLayout.

4.1.4 Skapandet av ”JJ-tester”

För att testa Josephsonövergångar har QT tagit fram en statisk krets med kontaktpunkter och en Josephsonövergång, vilket möjliggör avläsning och mätning av övergångens resistans med en probemaskin. För att förenkla och automatisera designfasen har vi skapat en PCell med samma dimensioner och kontaktpunkter som QT tidigare har använt, men som är parametriserad och anpassningsbar för framtida bruk. Detta resulterar i en PCell där man enkelt kan justera bredden, och därmed arean på Josephsonövergångarnas fingrar, vilket i sin tur möjliggör mätning av den resulterande resistansen.



Figur 4.4: En Josephsonövergångs-testare. Bilden är skapad i KLayout

Genom att göra teststukturen för Josephsonövergångar parametriserad har projektet bidragit till att automatisera ett tidigare statiskt moment i designfasen. Detta underlättar inte bara återanvändning och justeringar för framtida tester, utan minskar även behovet av framtida manuella modifieringar.

4.2 DRC

Detta avsnitt presenterar implementeringen av en mindre uppsättning designregler med användning av DRC-verktyget i KLayout. Verifiering rör Josephsonövergången som fristående komponent, men även verifiering av dess sammankoppling till kringliggande komponenter.

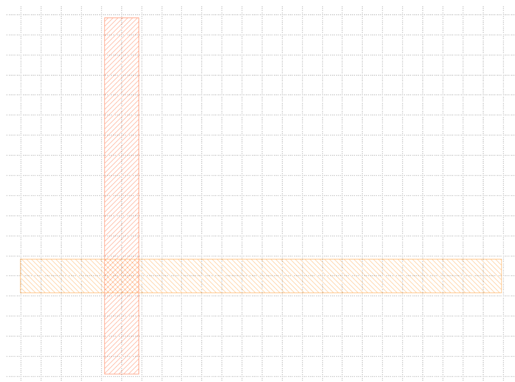
4.2.1 Verifiering av Josephsonövergången

En Josephsonövergång är en grundläggande komponent i supraledande elektronik, där två supraledande material separeras av ett tunt isolerande skikt. Övergången består av en horisontell och en vertikal arm som korsar varandra, genom vilka ström passerar. För att uppnå bästa möjliga resultat krävs att övergångens geometriska egenskaper uppfyller en specificerad kravspecifikation. Denna har utformats i samråd med handledarna och innefattar att det finns ett överlapp mellan armarna och att den horisontella armen ligger under den vertikala. Överlappsarean ska ligga inom ett visst intervall. Den övre delen av den vertikala armen, mätt från korset, ska vara större än den nedre, eftersom elektronerna strömmar i den riktningen. Den nedre delen måste dessutom vara längre än ett fastställt minimivärde.

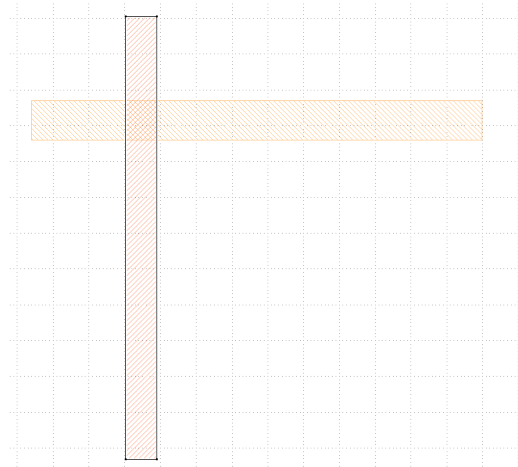
Det sistnämnda kravet beror på skuggavdunstning [27], där den horisontella armen först skapar en skuggzon. För att den vertikala armen ska deponeras korrekt och bilda en fungerande Josephsonkontakt måste den nå utanför denna skugga, vilket kräver tillräcklig längd.

För att verifiera att dessa krav uppfylls har ett DRC-program för Josephsonövergången utvecklats. Programmet analyserar armarna utifrån den specificerade regeluppsättningen. Genom denna automatiserade process säkerställs att varje övergång är korrekt utformad före fabrikation.

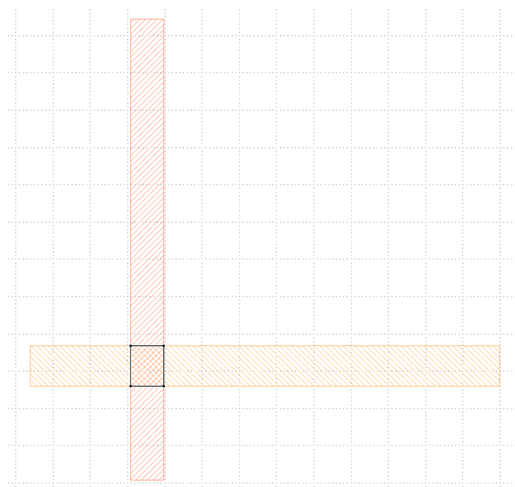
Som framgår av figurerna (4.5) markeras inga kanter för den korrekta implementeringen, medan de kanter som motsvarar de genererade felen framhävs för den uppsättning regler som är gällande för josephsonövergången och dess kravspecifikation.



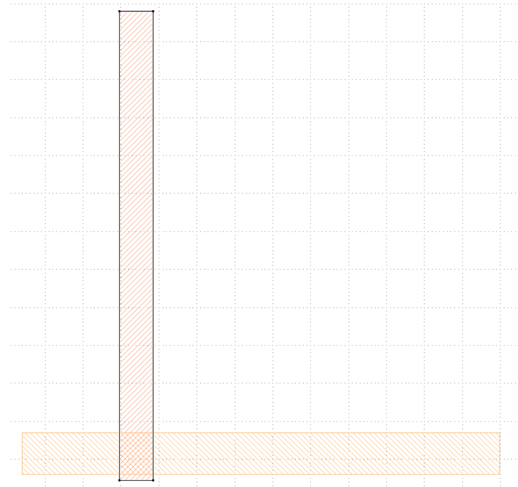
(a) En korrekt Josephsonövergång.



(b) En felaktigt implementerad Josephsonövergång då fel direktionalitet hos den vertikala armen föreligger.



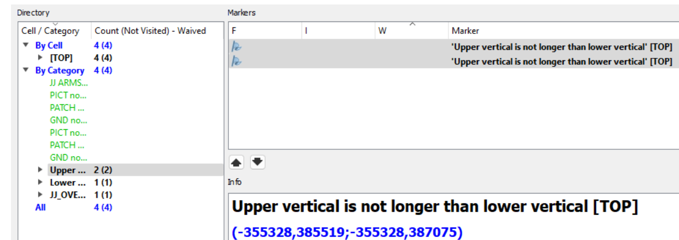
(c) En felaktigt implementerad Josephsonövergång då överlappsarean ligger utanför ett specifikt intervall.



(d) Felaktigt implementerad Josephsonövergång då den vertikala armen har för kort utsträckning nedanför överlappet.

Figur 4.5: En korrekt och tre felaktigt implementerade Josephsonövergångar. Bilderna är skapade i KLayout.

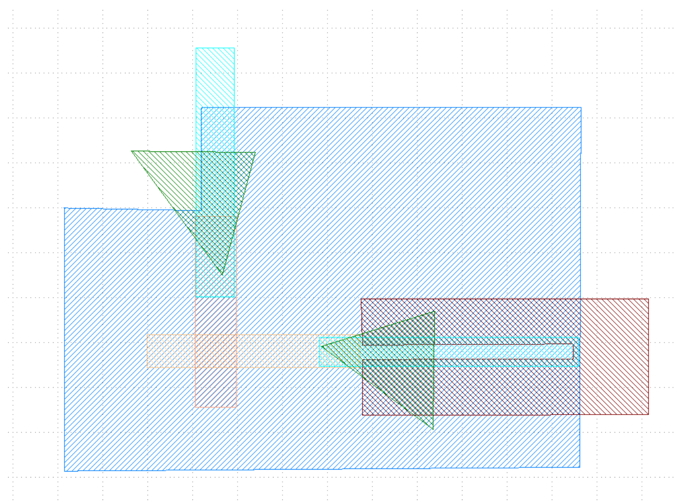
DRC-rapporten för programmet visas nedan. Noterbart är att det finns fler DRC-regler för Josephsonövergången, rörandes dess sammankoppling till kringliggande komponenter som inte belysts då Josephsonövergångarna i dessa exempel är fristående. Detta är inget fel, varför dessa tester passerar felfritt - motsvarande gröna rader i rapporten.



Figur 4.6: DRC-rapport för Josephson-övergången. Bilden visar det felmeddelande som matas ut då felaktig direktionalitet hos den vertikala armen föreligger. Bilden är skapad i KLayout.

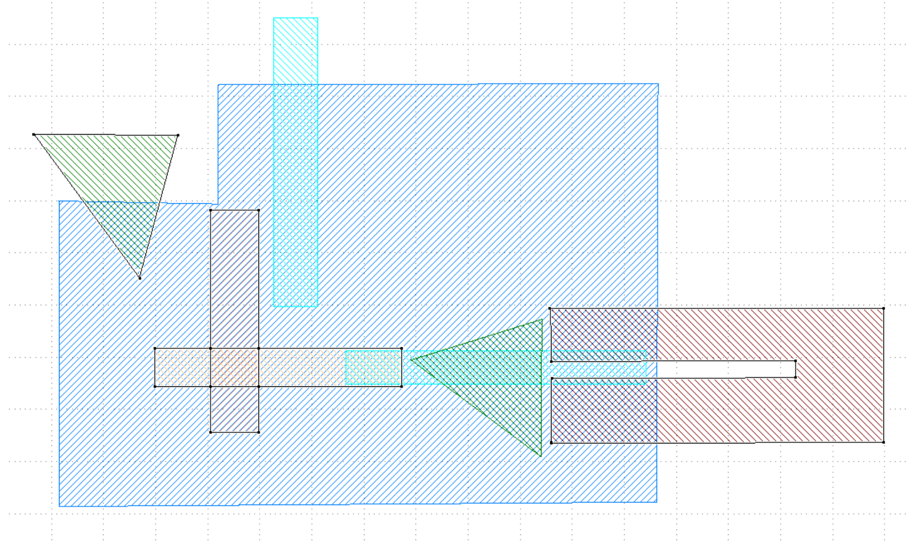
4.2.2 Verifiering av Josephsonövergångens sammankoppling till kringliggande komponenter

Nedan illustreras hur Josephsonövergångarna är kopplade till komponenterna PICT (Patch Integrated Cross-Type) och Patch. Dessa komponenter möjliggör en galvanisk kontakt mellan Josephsonövergången och jordplanet, vilket minimerar dielektriska förluster – en avgörande faktor för koherensen och den parametriska stabiliteten hos kvantbiten som Josephsonövergången är delkomponent utav [28]. Att verifiera att en galvanisk kontakt föreligger är alltså essentiellt, varför ett DRC-program för just detta syfte implementerades.

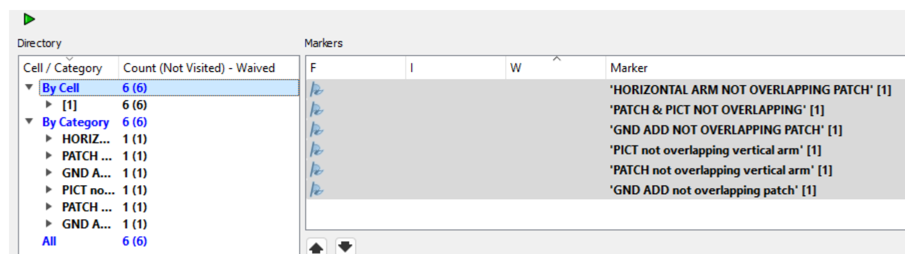


Figur 4.7: Josephsonövergång (orange) och dess sammankoppling till de kringliggande komponenterna PICT (turkos), Patch (grön) samt jordplanen (vit och röd). Bilden är skapad i KLayout.

Implementeringen bygger på validering av överlapp. DRC-programmet kontrollerar om det finns ett överlapp mellan jordplanet och komponenterna genom att, likt en kedja, kontrollera om armarna är anslutna till PICT, PICT till Patch och slutligen Patch till jordplanet. På så sätt säkerställs att armarna är korrekt anslutna till jordplanet, och därmed även att Josephsonövergångens förväntade egenskaper säkerställs.



Figur 4.8: Felmarkerade lager vid körning av DRC-programmet för verifiering av Josephsonövergångens sammankoppling till Patch, PICT och jordplanen. Bilden är skapad i KLayout.



Figur 4.9: Motsvarande DRC-rapport som fångar samtliga fel relaterade till den uteblivna galvaniska kontakten mellan Josephsonövergången och jordplanet. Bilden är skapad i KLayout.

De två DRC-programmen för Josephsonövergången bedömer vi som essentiella för att öka graden av automation i PDK:t. Genom att möjliggöra snabb och automatiserad verifiering frigörs inte bara värdefull tid i designprocessen – tillförlitligheten ökar dessutom markant. Risken för manuella fel minskar, felsökningen förenklas och arbetsprocessen blir således även mer robust, speciellt i takt med att fler och mer komplexa designregler implementeras, vilket lämpligen bör vara nästa steg. Detta har dessutom potentialen att flytta DRC från ett verktyg som fångar uppenbara och ödesdigra fel som sällan sker i praktiken men ändå behöver undvikas till att

säkerställa optimalitet genom att programmera in starkare villkor med mindre feltoleranser. För JJ-exemplet hade detta exempelvis kunnat vara centrering av PICT och Patch till armarna, och inte bara överlapp, vilket kan föreligga utan att centrering nödvändigtvis gör det.

4.3 Generella förbättringar

Utöver förbättringar med PCeller och implementering av DRC-program har vi även gjort mer allmänna förbättringar som vi presenterar här.

4.3.1 Tillägg av parameter som sparar PDK-versionen för varje PCell

För ett PDK som ständigt förbättras och vidareutvecklas är det viktigt att kunna säkerställa att användaren arbetar med den senaste versionen. Därför har en dold parameter lagts till i konstruktorn för varje PCell-klass. Denna använder funktionen `get_pdk_version(self)` för att hämta och lagra aktuell PDK-version. Informationen lagras sedan i den genererade OASIS- eller GDS-filen.

Metoden `get_pdk_version(self)` definieras i `PPCellDeclarationHelper`, vilket alla PCell-klasser ärver från, och extraherar PDK-versionen från `grain.xml`-filen, se 4.3.5. Metoden har följande implementation:

```
def get_pdk_version(self):
    grain_path = Path(__file__).resolve().parents[3] / "grain.xml"
    tree = ET.parse(grain_path)
    root = tree.getroot()
    return root.find("version").tex
```

4.3.2 Smartare uppdatering av ”step-size” i hjälp-struktur

För att generera punkter längs en kurva har funktionen `find_points` implementerats i klassen `CurvePoint`, som används för att lagra punkter längs en kurva. Klassen ingår i modulen `smooth.py`, vilken innehåller funktioner för att skapa olika typer av jämna spår. Funktionen `find_points` tar som indata ett parametriserat kurvpar $(x(t), y(t))$ och returnerar en lista med punktobjekt jämnt fördelade längs kurvan, med ett specificerat avstånd d mellan varje punkt.

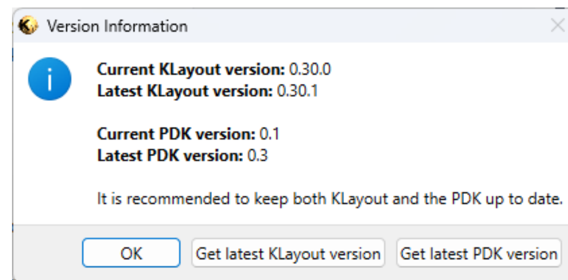
Tidigare användes ett fast inkrement, $t+1$, som första gissning i den numeriska lösningen för att hitta nästa punkt. Detta kan vara opålitligt eftersom kurvans verkliga längd per steg i t varierar beroende på kurvans lokala geometri. För att förbättra stabilitet och precision, ersattes detta med ett dynamiskt steg, $t+dt$, där $dt = d/l(t)$ och $l(t)$ motsvarar kurvans lokala hastighet. Detta gör att stegen anpassas efter kurvans faktiska utsträckning, vilket resulterar i en jämnare punktfördelning.

4.3.3 Automatisk versionskoll och laddning av teknologipaketet

Under projektets gång har vi undersökt andra implementationer av PDK och teknologipaket i KLayout. I ett fall öppnades automatiskt en tom layout med teknologipaketet aktiverat och korrekt lagerkonvention laddad. Vi fick även önskemål från våra handledare om att implementera en kontrollfunktion som jämför den installerade versionen av KLayout med den senaste publicerade versionen. Samma sak gällde för den installerade versionen av PDK:t, vilken jämförs med den senaste releasen. Vi valde att slå ihop dessa två idéer och skapade ett autorun-skript som körs vid varje uppstart av KLayout. Vid uppstart visas en informationsruta.

I informationsrutan visas den installerade samt den senast tillgängliga versionen av KLayout. Versionen av PDK:t man kör jämförs med den senaste utgåvan. Användaren ges möjlighet att välja mellan tre alternativ.

- `OK` stänger ner informationsrutan.
- `Get latest KLayout version` öppnar nerladdningssidan för KLayout i webläsaren.
- `Get latest PDK version` öppnar GitLab-sidan för PDK:t i webläsaren för enkel nerladdning av senaste versionen.



Figur 4.10: Infomationsrutan från autorunskriptet

När man tycker ner rutan laddas en tom GDS fil in som aktiverar teknologipaketet och med det även laddar in alla lager enligt lagerkonventionen.

Autorunskriptet bidrar till att medarbetare på QT håller koll på nya releases av KLayout och PDK:t. Detta leder till ett förbättrat arbetsflöde eftersom båda bidrar med förbättringar, utökad funktionalitet och buggfixar i varje uppdatering.

4.3.4 Automatisk omladdning av moduler

I PDK:t importeras alla filer som innehåller PCeller som moduler för att möjliggöra instansiering av dessa. Enligt Pythons standardbeteende laddas moduler endast vid första importen och uppdateras inte automatiskt vid ändringar. Tidigare krävdes en

omstart av KLayout för att ändringar i PCellerna skulle börja gälla. För att förenkla arbetsflödet skapades en funktion ”reload()” som laddar om samtliga PCell-moduler utan att KLayout behöver startas om.

Funktionen är definierad enligt följande:

```
def reload():
    for module in list(sys.modules):
        if module.startswith("QTQC") and
           sys.modules[module] is not None:
            # print(module)
            importlib.reload(sys.modules[module])
    QTQC().redraw()
    print("Forcibly reloaded QTQC")
```

Denna funktion leder till en stor tidsbesparing och är mycket användbar i utvecklingsfasen av PCeller. Istället för att starta om KLayout vid varje mindre ändring kan funktionen ”reload()” anropas för att se resultatet direkt.

4.3.5 Omstrukturering av teknologipaketet

När vi började på projektet fanns det redan ett teknologipaket som vi fortsatte att arbeta på. Teknologipaketet fanns i ett repository, repo på GitLab och var strukturerat enligt nedan.

Detta innebar att man var tvungen att ladda ner eller klona repot till `C:\Users\USER_NAME\KLayout\tech` för att KLayout ska kunna använda sig av teknologipaketet. KLayout har en inbyggd pakethanterare som heter Salt. För att i framtiden göra PDK:t och teknologipaket kompatibelt med Salt har vi valt att strukturera om våra filer enligt nedan.

Filstruktur innan

```
Klayout_tech/
|- lagerkonventioner
|- pymacros/
   |- variables.py
   |- QTQC/
      |- PCeller
```

Filstruktur efter

```
Klayout_tech/
|- grain.xml
|- tech
   |- lagerkonventioner
   |- docs/
   |- drc/
   |- pymacros/
      |- variables.py
      |- QTQC/
         |- PCeller
```

Den uppdaterade filstukturen gör att repot nu måste laddas ner eller klonas till

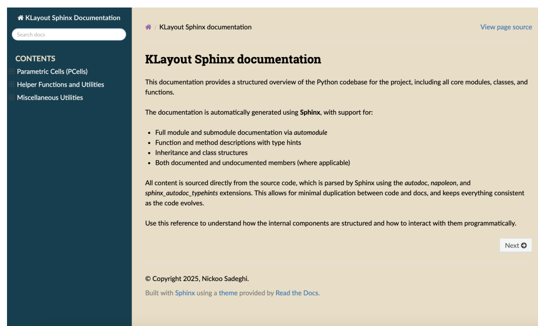
C:\Users\USER_NAME\KLayout\salt. Den uppdaterade filstukturen innehåller även filen grain.xml (A.4) som innehåller namn på paketet och även vilken version det är. Förändringen av filstrukturen innebär ingen större förbättring för QT:s nuvarande arbetsprocess, men den utgör en förberedelse inför en framtida eventuell publicering av PDK:t som öppen källkod.

4.4 Dokumentation

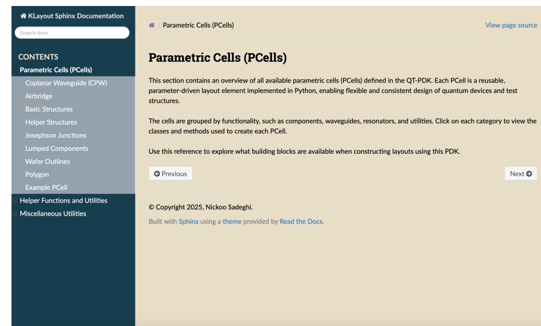
Följande avsnitt behandlar dokumentationen som skapats för PDK:t.

4.4.1 Automatisk koddokumentation via Sphinx

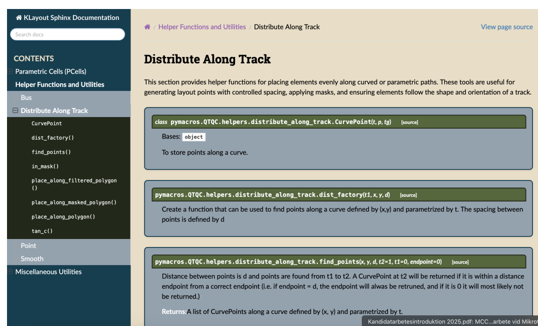
En fullständig dokumentation över alla PCeller, lagerkonventioner samt övriga hjälpceller har genererats med Sphinx. HTML-sidorna har grupperats efter PCeller, hjälpfunktioner och hjälp-klasser samt övriga filer. Nedan följer ett utdrag av sidorna.



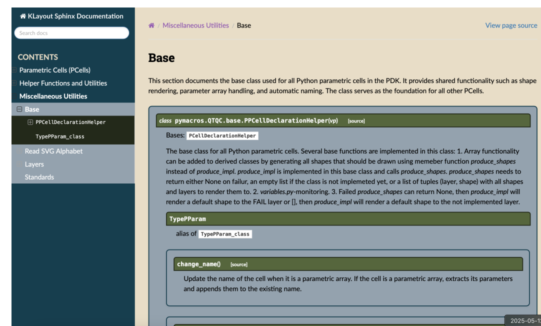
(a) Förstasidan genererad av index.rst.



(b) Introduktionssida för PCeller.



(c) Sidan för distribute_along_track-filen.



(d) Sidan för base-filen.

Figur 4.11: Utdrag från de genererade HTML-filerna skapade med Sphinx.

4.4.2 Uppdatering av README

Avslutningsvis har README-filen för salt-mappen uppdaterats för att återspegla de förändringar och förbättringar som gjorts i PDK:et.

5

Slutsatser

De åtgärder och implementationer som har genomförts i projektet visar tydligt hur graden av automation och användarvänlighet i designflödet har ökat. Genom att skapa nya samt förbättra befintliga PCeller har behovet av manuell handpåläggning i designfasen minskat, vilket har lett till en mer effektiv process.

Vidare säkerställer DRC att designen följer de definierade reglerna, vilket stärker kvalitetssäkringen och minskar behovet av manuell granskning. Detta bidrar till ett mer effektivt och tillförlitligt arbetsflöde.

Dokumentationen innehåller tydliga beskrivningar av funktionaliteten hos varje PCell, vilket underlättar inlärning och förståelse för nya användare och utvecklare. Detta möjliggör en smidigare vidareutveckling av PDK:t och minskar risken för fel vid framtida modifieringar.

För fortsatt utveckling rekommenderas att fler PCeller implementeras, särskilt sådana som kombinerar mindre PCeller och bildar större och mer komplexa enheter som enkelt kan återanvändas. Det bör även tas fram fler designregler och motsvarande DRC-skripts för dessa bör utvecklas för att täcka fler komponenter.

Genom att vidareutveckla PCeller och utöka designregelverket skapas förutsättningar för en mer automatiserad, pålitlig och skalbar designprocess, vilket stärker QT:s möjligheter att effektivt utveckla och realisera kvantteknologiska komponenter.

Litteratur

[1] Devdatt Dubhashi. *Quantumstack: Programmering av kvantdatorn*. Chalmers Research Projects. Hämtad: 2025-02-06. 2023. URL: https://research.chalmers.se/project/10712.

[2] Rebecka Adin. "Googles superchip sparar en kvadriljon år i problemlösning". I: *Göteborgs-Posten* (2025). Hämtad: 2025-02-06. URL: https://www.gp.se/ekonomi/googles-superchip-sparar-en-kvadriljon-ar-i-problemlosning.e17b2ba8-13b5-47c5-90d8-4c7672d0bd26.

[3] Chalmers University of Technology. *Kvantdatorer*. Hämtad: 2025-04-27. 2025. URL: https://www.chalmers.se/centrum/wacqt/upptack-kvantteknologi/kvantdatorer/.

[4] Riverlane. *Quantum Error Correction*. Hämtad: 2025-04-27. 2024. URL: https://www.riverlane.com/quantum-error-correction.

[5] Wikipedia contributors. *Lithography — Wikipedia, The Free Encyclopedia*. Hämtad: 2025-05-9. 2025. URL: https://en.wikipedia.org/wiki/Lithography.

[6] Ingela Roos. *Ny svensk kvantdator blir tillgänglig för industrin*. https://www.chalmers.se/aktuellt/nyheter/wacqt-ny-svensk-kvantdator-blir-tillganglig-for-industrin/. Hämtad: 2025-04-27. 2023. URL: https://www.chalmers.se/aktuellt/nyheter/wacqt-ny-svensk-kvantdator-blir-tillganglig-for-industrin/.

[7] UNIC Training. *Introduction to PDKs*. Hämtad: 2025-04-27. 2025. URL: https://unic-cass.github.io/training/1.2-pdk-intro.html#contents-of-a-pdk.

[8] AnySilicon. *Process Design Kit – Ultimate Guide*. https://anysilicon.com/process-design-kit-ultimate-guide/. Hämtad: 2025-04-27. 2025.

[9] Matthias Köfferlein. *KLayout - Layout Viewer and Editor*. Hämtad: 2025-04-28. 2025. URL: https://www.klayout.de/.

[10] Sobhana Tayenjam m. fl. "A PCell design methodology for automatic layout generation of spiral inductor using skill script". I: *2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)* (2017). DOI: 10.1109/icmdcs.2017.8211572.

[11] Jyoti Gautam m. fl. "Generic Parametrized Cell Design Methodology for Analog and Mixedsignal Circuits". I: *2024 First International Conference on Electronics, Communication and Signal Processing (ICECSP)* (2024). DOI: 10.1109/icecsp61809.2024.10698627.

[12] 911EDA. *PCB Design Rule Checks: Keys to Layout Success*. Hämtad: 2025-04-27. 2024. URL: https://www.911eda.com/articles/pcb-design-rule-checks-cut-down-board-respins/.

[13] Semiconductor Engineering. *Design Rule Checking (DRC)*. Hämtad: 2025-05-10. 2023. URL: https://semiengineering.com/knowledge_centers/eda-design/verification/design-rule-checking-drc/.

[14] Luca Alloatti m. fl. "Photonics design tool for advanced CMOS nodes". I: *arXiv preprint arXiv:1504.03669* (2015). Hämtad: 2025-05-10. URL: https://arxiv.org/abs/1504.03669.

[15] Ana Paula Valente Pais, Manuel Lois Anido och Carlo Emmanoel Tolla de Oliveira. "Developing a Distributed Architecture for Design Rule Checking". I: *Proceedings of the 44th IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*. Vol. 2. Hämtad: 2025-05-10. IEEE, 2001, s. 678–681. DOI: 10.1109/MWSCAS.2001.986279.

[16] KLayout Documentation. *LVS Overview*. Hämtad: 2025-04-27. 2025. URL: https://www.klayout.de/doc-qt5/manual/lvs_overview.html.

[17] Johannes Coetzee. *A Physical Design and Layout Versus Schematic Framework for Superconducting Electronics*. Hämtad: 2025-04-27. 2025. URL: https://scholar.sun.ac.za/.

[18] Matthias Köhler. *The 2.5d View*. https://www.klayout.de/doc/about/25d_view.html. Hämtad: 2025-05-10.

[19] Polymetaal. *Lithography - The Process*. Hämtad: 2025-05-10. n.d. URL: https://www.polymetaal.nl/contents/en-uk/d1905_Lithography.html.

[20] HORIBA Semiconductor. *Lithography*. Hämtad: 2025-04-28. 2025. URL: https://www.horiba.com/int/semiconductor/process/lithography/.

[21] JEOL USA. *Overview of Electron Beam Lithography*. Hämtad: 2025-05-10. n.d. URL: https://www.jeolusa.com/NEWS-EVENTS/Blog/overview-electron-beam-lithography.

[22] Sphinx Documentation Team. *Sphinx Documentation*. https://www.sphinx-doc.org/en/master/. Hämtad: 2025-04-28. 2025.

[23] Sphinx Documentation Team. *Sphinx Quickstart Guide*. https://www.sphinx-doc.org/en/master/usage/quickstart.html. Hämtad: 2025-04-29. 2025.

[24] Sphinx Documentation Team. *Installing Sphinx*. https://www.sphinx-doc.org/en/master/usage/installation.html. Hämtad: 2025-04-29. 2025.

[25] KLayout Development Team. *KLayout Python Scripting Reference*. https://www.klayout.de/doc/programming/python.html. Hämtad: 2025-05-10. 2025.

[26] Sphinx Documentation Team. *Sphinx Configuration Guide*. https://www.sphinx-doc.org/en/master/usage/configuration.html. Hämtad: 2025-05-10. 2025.

[27] Federico Borsoi, Kevin A. van Hoogdalem, Mart Brauns m. fl. *Sn–InAs nanowire shadow-defined Josephson junctions*. https://www.researchgate.net/publication/389946944_Sn-InAs_nanowire_shadow-defined_Josephson_junctions. Hämtad: 2025-05-11. 2025. DOI: 10.48550/arXiv.2503.13725.

[28] A. Osman m. fl. "Simplified Josephson-junction fabrication process for reproducibly high-performance superconducting qubits". I: *Applied Physics Letters* 118.6 (2021), s. 064002. DOI: 10.1063/5.0037093.

A

Appendix

Följande bilagor består av kod tillhörande olika delar av resultatet.

A.1 PP_example.py

```
"""
Example PCell to use when creating a new PCell.
"""

# You must import these
import pya
from QTQC.layers import QTL_layers
from QTQC.base import PPCellDeclarationHelper

# Change `example` to something short but descriptive
class example(PPCellDeclarationHelper):
    """
    The PCell declaration for `example`
    """

    def __init__(self, vp):
        super().__init__(vp)

        # Define the parameters you need to draw your design
        self.param('width',     self.TypeDouble, 'Example width',    default = 15)
        self.param('height',    self.TypeDouble, 'Example height',   default = 10)
        self.param('radius',    self.TypeDouble, 'Example radius',   default = 1)

        # Define the layer(s) your design will utilize
        self.param("layer",     self.TypeLayer,  "Example layer",
                    default = QTL_layers['gnd.add'], hidden=False, readonly=True)

        # A must to keep track of which PDK version the PCell was drawn with
        self.param("pdk_version", self.TypeString, "PDK Version",
                    default = self.get_pdk_version(), readonly=True, hidden=True)

    def display_text_impl(self):
        # Provide a descriptive text for the cell
        return "Example PCell"

```

```
36      def coerce_parameters_impl(self):
37          """ Used to coerce the parameters and updating them in
38              the PCell parametes menu in KLayout
39          """
40          pass
41
42      def transformation_from_shape_impl(self):
43          """ Implement the "Create PCell from shape" protocol:
44              we use the center of the shape's bounding box to
45              determine the transformation
46
47          Returns:
48              Point: The center of the cells bounding box
49          """
50          return self.shape.bbox().center()
51
52      def produce_shapes(self):
53          """ Main method! This is where all the code that draws your PCell is.
54
55          Returns:
56              list: a list of shapes that will be drawn on the layout.
57          """
58          # Printing for debugging reasons
59          print('Example')
60          print("Dbu: ", self.layout.dbu)
61
62          # A must to correct for float number presenation issues
63          self.layout.dbu = round(self.layout.dbu, 5)
64          dbu = self.layout.dbu
65
66          # Parameters in dbu
67          width = self.width / dbu
68          height = self.height / dbu
69          radius = self.radius / dbu
70
71          # List of shapes that will be returned
72          shapes = list()
73
74          # Coordinates for the point making the box
75          upperleft   = pya.Point(-width/2, height/2)
76          upperright  = pya.Point(width/2, height/2)
77          lowerright  = pya.Point(width/2, -height/2)
78          lowerleft   = pya.Point(-width/2, -height/2)
79
80          # Create a polygon (in this case a box) from the points
81          box = pya.Polygon([upperleft, upperright, lowerright, lowerleft])
82
83          # Round the corners of the box box.round_corners(inner radius,
84          # outer radius, number of points per full circle)
85          box = box.round_corners(radius, radius, 100)
86
87          # Append the shapes list with what layer you want to add what shape.
88          # In out example we will use layer 12, GND add
89          # and we want to add our box shape
90          shapes.append((self.layer, box))
91
```

```
        return shapes

```

A.2 Lumped Capacitor radie-logik

```
  def coerce_parameters_impl(self):
    if self.use_handle:
      if self.handle_radius: # Checks if handle is being dragged
        temp = np.sqrt(self.handle_radius.x**2 + self.handle_radius.y**2)
        self.r0 = "{:.1f}".format(temp) # Sets r0 to coordinates of the
        ↪  handle_radius

      if self.r0 != self.extract(self.R): # Checks if radius has changed
        self.R = str(self.r0)
    else:
      self.handle_radius = pya.DPoint(self.extract(self.R), 0) # Handle still
      ↪  follows the edge of capacitor
```

A.3 Lumped Inductor borttagning av låda

```
self.param("ground",        self.TypeShape, "Ground",                     default
↪  = pya.DPolygon([pya.DPoint(-100, -100), pya.DPoint(0, -100), pya.DPoint(0,
↪  100), pya.DPoint(-100, 100)], 0), hidden=True)
```

A.4 grain.xml

```
<?xml version="1.0" encoding="utf-8"?>
<salt-grain>
 <name>QTQC Tech</name>
 <token/>
 <hidden>false</hidden>
 <version>0.0</version>
 <api-version/>
 <title>Quantum Technology</title>
 <doc>The standard technology used by QT at Chalmers University of Technology</doc>
 <doc-url>
    https://git.chalmers.se/mc2/qt/qc/Klayout_tech/-/wikis/home/KLayout-Technology
 </doc-url>
 <url/>
 <license>GPLv3</license>
 <author>
    Robert Rehammar, Andreas Nylander,
    Carl Svensson, Arik Ben-Shabat, Nickoo Sadeghi
 </author>
 <author-contact/>
 <authored-time/>
 <installed-time>2025-04-08T13:53:24</installed-time>
</salt-grain>
```

INSTITUTIONEN FÖR MIKROTEKNOLOGI OCH NANOVETENSKAP
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se