ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ
KATEDRA ELEKTRONIKY A INFORMAČNÍCH TECHNOLOGIÍ

BAKALÁŘSKÁ PRÁCE

Metodika správy otevřené knihovny součástek

Ondřej Růžička **2021**

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta elektrotechnická

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ondřej RŮŽIČKA**
Osobní číslo:	**E18B0029P**
Studijní program:	**B2612 Elektrotechnika a informatika**
Studijní obor:	**Elektronika a telekomunikace**
Téma práce:	**Metodika správy otevřené knihovny součástek**
Zadávající katedra:	**Katedra elektroniky a informačních technologií**

Zásady pro vypracování

1. Seznamte se standardními nástroji pro návrh PCB a správu knihoven součástek.
2. Popište přístupy pro správu knihoven součastek v nejpoužívanějších návrhových systémech.
3. Definujte obecnou metodiku pro správu otevřené, sdílené knihovny součástek, zaměřte se na metodiku návrhu nových knihovních prvků a proces kvalifikace existujících komponent a jejich začlenění do knihovny, využijte vlastnosti verzovacích systémů a nástrojů pro podporu komunitniho vývoje.
4. S využitím navržené metodiky vytvořte knihovnu součástek, pro zvolené open-source nástroje, obsahující komponenty open hardware jednotky KETCube.

Rozsah bakalářské práce: **30 – 40 stran**
Rozsah grafických prací: **podle doporučení vedoucího**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. Záhlava, Vít. Návrh a konstrukce desek plošných spojů: principy a pravidla praktického návrhu. BEN-technická literatura, 2010.
2. Dokumentace komerčních i open-source elektronických nástrojů pro podporu návrhu (EDA) PCB, zejména: Altium a KiCAD

Vedoucí bakalářské práce: **Ing. Jan Bělohoubek**
Katedra materiálů a technologií

Datum zadání bakalářské práce: **9. října 2020**
Termín odevzdání bakalářské práce: **27. května 2021**

L.S.

Prof. Ing. Zdeněk Peroutka, Ph.D.
děkan

Doc. Ing. Jiří Hammerbauer, Ph.D.
vedoucí katedry

V Plzni dne 9. října 2020

Abstrakt

Otevřené nástroje pro návrh schémat a plošných spojů jsou velmi rozšířené a nabízejí mnoho možných přístupů k jejich používání. To může vést k nekonzistenci v použití uvnitř organizací. V této práci se zaměřuji na návrh metodiky pro správu knihoven, ze které vychází jednotný přístup k jejich tvorbě a udržování. Práce popisuje standardní přístupy pro správu knihoven. Je navržena metodika správy knihovny zaměřena na otevřený návrhový nástroj KiCAD, pro verzování a správu knihovny využívající systém správy revizí Git a platforma GitHub. Základem ověření je využití skriptů a nezávislého manuálního posouzení. Každá součástka obsahuje soubory s metadaty, která charakterizují mimo jiné stav součástky a její odpovídající úroveň ověření.

Klíčová slova

otevřená knihovna, knihovna součástek, KiCAD, správa knihovny, git, systémy správy revizí, verifikace, validace

Abstract

Open-source PCB design tools are very popular nowadays and they may be employed in diverse workflows. This can lead to inconsistency and insufficient specification of use inside organizations. This thesis is focused on a library management methodology, which leads to a proposal, defining a unified approach to library design and management. This thesis describes standardized approaches to library management. This thesis defines the component design, verification, and validation process, according to the proposed methodology. I used KiCAD as the main software for library design, and git for versioning, and GitHub platform for project management. Automated tools and independent manual reviews are used for the verification process. Every component contains metadata files, which characterize, among other things, the component's status and its corresponding verification level.

Keywords

open library, component library, KiCAD, library management, git, version control systems, verification, validation

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem svou závěrečnou práci vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 270 trestního zákona č. 40/2009 Sb.

Také prohlašuji, že veškerý software, použitý při řešení této bakalářské práce, je legální.

V Plzni dne June 17, 2021

Ondřej Růžička

. . . . . .

Podpis

Obsah

Seznam obrázků

Seznam tabulek

Seznam symbolů a zkratek

CAD	Computer aided design
CVS	Concurrent Versions System
DRC	Design rule check
EDA	Electronic design automation
GPL	General Public License
GNU	GNU's Not Unix
KET	Katedra materiálů a technologií
KLC	KiCAD library convention
PCB	Printed circuit board
RICE	Výzkumné a inovační centrum elektrotechniky
SC	SmartCAMPUS ZČU
SVN	Apache Subversion
VRC	Verification rule check
VCS	Version control system
ZČU	Západočeská univerzita

1

Úvod

V práci se zabývám rešerší aktuálně používaných návrhových systémů a jejich doporučených přístupů pro tvorbu a správu knihoven. Na základě těchto podkladů, informací získaných z doporučené literatury a vlastních zkušeností autora je následně vytvořen návrh implementace těchto metod do prostředí otevřeného softwaru, s využitím verzovacích systémů.

Cílem práce je navrženou metodiku uplatnit při tvorbě otevřené knihovny vycházející z interní součástkové knihovny používané v centru RICE (Výzkumné a inovační centrum elektrotechniky). Tato knihovna by měla být následně používána na univerzitě v rámci projektu SmartCAMPUS ZČU pro tvorbu otevřeného hardwaru pomocí otevřených nástrojů, jelikož v současné době je na FEL ZČU, a zejména v centru RICE používán systém Altium Designer. To nezapadá zcela do koncepce otevřených projektů a proto by bylo vhodné mít prostředky podporující práci s otevřenými EDA (Electronic design automation) systémy v tomto prostředí. Právě vytvoření odpovídající otevřené knihovny je tedy logickým krokem.

Jako ověření použitelnosti vytvořené metodiky a s tím spojené knihovny, by mělo být možné na konci práce nakreslit desku KETCube[1], jejíž původní návrh je vytvořen v systému Altium Designer.

V centru RICE se využívá standardizovaná součástková knihovna pro systém Altium designer, ve které je každá součástka zařazena do odpovídající kategorie a má unikátní interní číslo. Cílem je vytvořit otevřenou knihovnu, která bude na tuto napojena právě pomocí unikátních identifikačních čísel.

[1] `https://github.com/SmartCAMPUSZCU/KETCube-docs`

2

Systémy pro návrh desek plošných spojů

Nástroje pro návrh plošných spojů můžeme rozdělit na komerční, zpravidla s uzavřeným zdrojovým kódem a zpoplatněními či omezeními spojenými s jejich používáním a komunitní, které se vyznačují komunitní tvorbou a žádným omezením použití v závislosti na pořízené licenci. V následujícím popisu se budu zabývat především přístupem jednotlivých nástrojů k tvorbě a správě součástkových knihoven.

Poměrně obecně lze říci, že součástka v knihovně obsahuje několik částí, komponent. Jsou to schématický symbol, footprint a jemu odpovídající 3D model. Dalšími částmi, které již nejsou nezbytné, mohou být například elektrický model pro simulaci (SPICE) nebo soubor metadat specifikující důležité vlastnosti součástek, případně reference na dodavatelské řetězce. Ke každé části má konkrétní návrhový systém definované požadavky a doporučení, která se mohou napříč systémy lišit. Zpravidla také systém nabízí nějakou možnost kontroly vytvořené součástky, ať už ve formě grafického průvodce (wizardu) či skriptu. Jejich spuštění nemusí být automatické.

2.1 Komerční EDA nástroje

Existuje velmi mnoho různých komerčních EDA systémů, z nichž některé jsou interní záležitostí firem (například Advanced Design System – Keysight), jiné jsou dostupné celé komerční sféře. Z těch nejznámějších jsou to Altium Designer[1], Autodesk Eagle[2], Circuitmaker[3], Proteus Design Suite[4], CR-8000[5], OrCAD[6]. Některé z nich jsou zdarma, například Circuitmaker, některé jsou zdarma pro nekomerční použití s určitými omezeními, například Autodesk Eagle, jiné jsou plně placené a jejich licence mohou být dosti nákladné, typ-

[1] `https://www.altium.com/altium-designer/`
[2] `https://www.autodesk.com/products/eagle/overview`
[3] `https://www.altium.com/circuitmaker/overview`
[4] `https://www.labcenter.com/`
[5] `https://www.zuken.com/en/product/cr-8000/`
[6] `https://www.orcad.com/`

ickým příkladem je Altium Designer. Ohledně funkcionalit lze hodnotit mnoho parametrů, mimo jiné třeba dostupnost práce s 3D modely, schopnost návrh simulovat, či pokročilé funkce jako programování logických obvodů [9]. Volba je tedy značně komplexní a mnohdy jsou uvnitř podniků zavedené standardy pro používání jednoho konkrétního návrhového systému. Porovnání vybraných systémů je zachyceno v tabulce 2.1. Systémy byly vybrány tak, aby představovaly reprezentativní vzorek různých segmentů trhu s PCB EDA systémy - velmi rozsáhlé porovnání základních parametrů mnoha dalších návrhových systémů lze nalézt online[7]. K nejdůležitějším parametrům pro porovnání návrhových systémů patří: možnost kreslení schématu, možnost návrhu PCB a podpora 3D modelu. Dalšími důležitými parametry jsou: možnost simulace, podpora více platforem nebo rozšiřitelnost.

Systém	Základní funkcionality	Simulace	Rozšiřitelnost
Altium Designer	Ano	Ano	Delphi, JS, VB
Autodesk Eagle	Ano	Ano	ULP (Proprietální jazyk)
Circuitmaker	Ano	Ano	Ne
Proteus Design Suite	Ano	Ano	Interní
CR-8000	Ano	Ano	Ne
OrCAD	Ano	Ano	TCL/TK, SKILL

Tab. 2.1: Porovnání komerčních EDA nástrojů

2.2 Open source EDA nástroje

V současné době je k dohledání několik významných open source nástrojů. Jsou to zejména gEDA[8], Fritzing[9], XCircuit[10], PCB-rnd[11] a KiCAD[12]. Z uvedených nástrojů není pro tyto účely příliš vhodný Fritzing, jelikož nepodporuje simulace, skriptování a je obecně cílen spíše na méně pokročilé uživatele a nikoliv pro profesionální užití. Systém gEDA ve svém základu nepodporuje 3D modely při návrhu, tuto funkci je možné doinstalovat, ale představuje to komplikaci. Ani systém XCircuit nepodporuje práci s 3D modely, takže rovněž není příliš vhodný. Stejné porovnání jako pro komerční EDA nástroje je provedeno v tabulce 2.2.

Ještě se podívejme, jací autoři či instituce vytvořili jednotlivé nástroje. Nástroj Fritzing byl vytvořen v rámci univerzity aplikovaných věd Potsdam, nástroj gEDA byl vytvořen původně jedním člověkem, Alešem Hvezdou. V současné době je spravován komunitou a nové verze pravidelně vycházejí. Nástroj PCB-rnd byl tvořen Tiborem Palinkasem a

[7] `https://en.wikipedia.org/wiki/Comparison_of_EDA_software`
[8] `http://www.geda-project.org/`
[9] `https://fritzing.org/`
[10] `http://opencircuitdesign.com/xcircuit/`
[11] `http://repo.hu/projects/pcb-rnd/index.html`
[12] `https://kicad.org/`

týmem několika jednotlivců okolo něho. XCircuit je vyvíjen jen jedním člověkem, Timem Edwardsem. Původním autorem KiCADu je Jean-Pierre Charras a skupina vývojářů. V současné době je podporován organizacemi jako CERN, The Raspberry Pi Foundation, Arduino LLC či Digi-Key Electronics [3].

Z hlediska funkcionalit a na základě provedené analýzy lze říci, že neexistuje open-source nástroj, který by byl ve všech parametrech srovnatelný s uzavřenými/komerčními nástroji a který by poskytoval poskytoval všechny jejich funkcionality. Je ale patrné, že z uvedených otevřených nástrojů se z hlediska parametrů, funkcionalit i uživatelské podpory a předvídatelnosti vývojového cyklu nejvíce přibližuje komerčním nástrojům právě KiCAD.

Systém	Základní funkcionality	Simulace	Rozšiřitelnost	Podpora	Fórum
gEDA	Schéma + DPS	Ano	Guile	komunitní	Ano[13]
Fritzing	Schéma + DPS	Ne	Ne	komunitní	Ano[14]
XCircuit	Schéma	Ano	TCL	omezená	Ne
PCB-rnd	DPS + 3D model	Ano	Python, Perl, TCL, JS,...	komunitní	Ne
KiCAD	Ano	Ano	Python	komunitní i komerční	Ano[15]

Tab. 2.2: Porovnání open source EDA nástrojů

2.3 Správa knihoven

Metodice tvorby a správy knihovny se věnují takřka všichni poskytovatelé PCB EDA nástrojů. Dost často ovšem převážnou část uvedených materiálů věnují tvorbě a případně kontrole jednotlivých součástek, nikoliv metodice správy celé knihovny.

Altium designer má velmi rozsáhlý dokument [14], věnující se procesu verifikace celého návrhu, v rámci kterého se částečně věnuje i ověření (verifikaci) knihovny. Největší důraz se zde klade na lidský faktor a nezávislou kontrolu. Dokument upozorňuje na nutnost ověření footprintu, jeho vzhledu a rozměru v reálném světě před jeho implementací do návrhu. Dále upozorňuje na nutnost kontroly přiřazení správných pinů mezi symbolem a footprintem. Zcela pochopitelně uvádí, že mnoho problémů si uživatel ušetří, jestliže použije integrované knihovny. Struktura knihoven nástroje Altium designer je popsána v dokumentu [7].

[13] `https://answers.launchpad.net/pcb`
[14] `https://forum.fritzing.org/`
[15] `https://forum.kicad.info/`

Nástroj KiCAD disponuje souborem pravidel KLC, který lze považovat za standard pro tvorbu vestavěných knihoven KiCADu. Řeší pravidla pro každou část návrhu součástky, výslednou metodiku nezávislé kontroly však už ne. Zároveň neřeší rozdělení součástek na základě úrovně ověření, kterou prošly.

Eagle se z oficiálních zdrojů této tématice příliš nevěnuje, existují však dostupné materiály [15], které ho využívají. Zde je také kladen důraz na manuální kontrolu navržené součástky, na kontrolu správných rozměrů a přiřazení pinů. Nejde však o materiál příliš podrobný.

Dalším sektorem, který se věnuje metodice správy knihoven jsou společnosti, které poskytují multiplatformní součástkové knihovny. Mezi ně patří například SnapEDA či UltraLibrarian. SnapEDA nabízí možnost automatické kontroly součástky, pomocí vlastních interních algoritmů [16]. Kromě toho uvádí seznam obecných pravidel pro návrh součástky, které zahrnují obecné checklisty obdobné pro všechny návrhy [17]. Skupina UltraLibrarian nabízí také úkony pro kontrolu součástky ve formě checklistu [18], není ale příliš konkrétní a jedná se spíše o přehled. Jde o uzavřenou skupinu a konkrétní informace budou pravděpodobně čistě interní.

2.4 Správa knihoven v systému Altium

Altium designer je velmi rozšířený komerční systém. Z pohledu této práce je důležité, že je prakticky hlavním EDA nástrojem používaným na FEL ZČU. Po dohodě s vedoucím práce byl tedy zvolen jako referenční nástroj, se kterým budeme srovnávat nástroje otevřené.

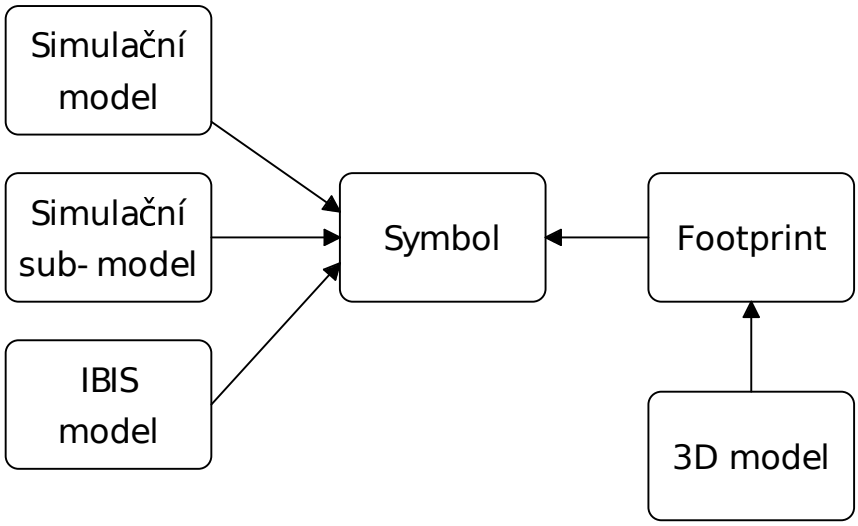
2.4.1 Struktura základní knihovny Altium

Nejjednodušším typem knihoven v Altiu jsou knihovny základní. Součástka je zde graficky i elektricky reprezentována schematickým symbolem, na který je navázán jeden či více footprintů. Na footprinty mohou být navázány 3D modely. Na schematický symbol je dále navázán elektrický model a případně model signálové integrity. Tato struktura je principiálně nejjednodušší a v případě plného využití všech funkcionalit se skládá ze šesti souborů. Diagram závislostí je znázorněn na obrázku 2.1.

V nejjednodušší metodologii schématický symbol reprezentuje součástku, protože jsou na něj navázány všechny důležité informace. Vzniká zde zde však problém s duplicitou, který kromě většího objemu výsledných dat má ještě další problém. Tím je potenciální vznik chyby, která nebude opravena ve všech souborech, kde se vyskytují nezávisle ta samá data. Tomuto problému se budu ještě v průběhu práce věnovat.

2.4.2 Struktura integrované knihovny Altium

V případě integrované knihovny Altium jde o velmi prostý koncept. Celá struktura z předchozí verze, tedy základní knihovny Altium, se spojí do jednoho souboru, který se



Obr. 2.1: Struktura základní knihovny Altium

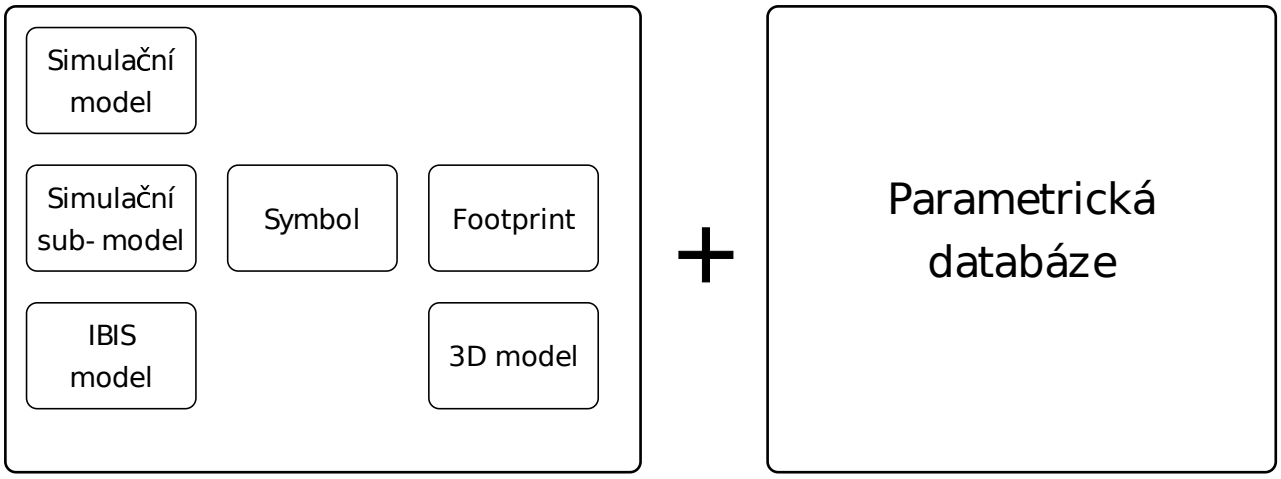
označuje jako integrovaná knihovna. Dojde tedy ke snížení počtu celkového množství souborů. Oproti první variantě jde o možnost bezpečnější, jelikož všechny vazby (symbol-footprint,...) jsou uvnitř jednoho souboru a nehrozí tedy chyby při případném kopírování. Bezpečnost je zároveň zajištěna tím, že výsledné soubory jsou určeny jen pro čtení, není tedy možné je upravovat a způsobit v nich chybu. Tento koncept je použitelný v případě potřeby snadné přenositelnosti knihoven, pro samotnou tvorbu a správu knihovny se příliš nehodí.

2.4.3 Struktura databázové knihovny Altium

Tato struktura je složitější než předchozí dvě, poskytuje však značnou výhodu pro větší organizace. Je totiž možné s její pomocí vytvořit centralizovanou knihovnu součástek, ve které jsou obsaženy všechny potřebné informace o součástkách na jednom místě. Databáze je organizována jako tabulka, kdy každý řádek specifikuje unikátní součástku, obvykle definovanou výrobním číslem či interním označením organizace, a každý sloupec odpovídá určité vlastnosti. Ve sloupcích se tedy pro součástku definují schematický symbol, footprint, 3D model, simulační model a jakýkoliv další parametr, který je vyžadován.

Podmínkou efektivity této struktury je důsledná dekompozice součástek. To následně umožní jednoduchou znovupoužitelnost jednotlivých částí a zjednoduší to znovupoužitelnost metadat v různých systémech, různými uživateli, s různými právy [7], [8].

Tato struktura umožňuje jednoduché parametrické vyhledávání součástek a eliminuje problém duplicity. Velmi usnadňuje správu velkého množství součástek, proto je žádoucí u větších organizací.



Obr. 2.2: Struktura databázové knihovny Altium

2.4.4 Struktura součástkové knihovny Altium

Nejvíce propracovanou je struktura součástkové knihovny, která kromě veškerých informací o součástce ještě obsahuje napojení na centralizovaný cloudový systém, označovaný jako „Vault“. V tomto případě jsou od samotné součástky odděleny informace o dodavatelském řetězci, tyto se udržují jako zvláštní položka v centrálním „vaultu“. Toto zajišťuje relevantnost každé součástky, jelikož informace v centrálním „vaultu“ jsou aktualizovány v reálném čase. Zjednodušuje to tedy práci s vývojem cen, dodavatelským řetězcem a zároveň celkovou správou života součástky, kterou centrální „vault“ též umožňuje.

Samotnou službu centralizovaného cloudového úložiště poskytuje Altium svým uživatelům, a kromě popsaných služeb poskytuje ještě další nadstavby, které už jsou pokročilejší a více specificky zaměřené.

2.5 Správa knihoven v systému KiCAD

Na základě porovnání, uvedeného výše v tabulce 2.2 byl zvolen pro tuto práci jako hlavní nástroj právě KiCAD, jež poskytuje všechny základní funkcionality EDA systémů. Mezi jeho základní funkce tedy patří kreslení schématu, tvorba a správa schematických symbolů, návrh plošného spoje a také tvorba a správa footprintů. Zároveň na rozdíl od jeho zmíněných konkurenčních otevřených systémů umožňuje i práci s 3D modely a simulace obvodů. Z těchto důvodů byl právě pro tuto práci zvolen a v dalším textu se budeme věnovat porovnání jeho možností s nástrojem Altium Designer. Napřed ale ještě popíšeme funkcionality KiCADu.

KiCAD poskytuje kromě standardních funkcionalit pro kreslení schémat a plošných spojů navíc ještě prohlížeč gerberů (souborový formát pro přenos dat z návrhu PCB do výroby. Je považován za průmyslový standard [4]). Navíc má KiCAD možnost skriptování, či doinstalace pluginů dle potřeby. Tohoto jsem následně využil při kreslení desky

KETCube, konkrétně se jednalo o plugin pro návrh RF částí desky[16].

Z pohledu kontroly knihovny je KiCAD velmi dobře uchopitelný, jelikož poskytuje volně přístupný soubor pravidel pro jejich tvorbu [2]. Tento se jmenuje KiCAD Library Convention (KLC) a věnuje se zvlášť všem jednotlivým komponentům tvořené součástky. Pro kontrolu těchto pravidel existují dostupné skripty, které umožňují část kontroly automatizovat [2].

2.6 Struktura vestavěných knihoven KiCADu

Pokud bychom porovnali koncepci knihoven KiCADu a dříve zmíněného Altia, pak by přístup KiCADu byl velmi podobný knihovní struktuře základních knihoven Altium. Jde tedy o první, principiálně nejjednodušší řešení, které má nevýhodu ve formě již zmiňované potenciální duplicity obsahu, které se v KiCADu nelze vyhnout úplně. Tento fakt je třeba vzít při návrhu knihoven v potaz. KiCAD umožňuje vytvářet uživatelské knihovny, poskytuje ale také základní, vestavěné knihovny, které uživatel obdrží při instalaci programu.

Vestavěné knihovny KiCADu jsou umístěny přímo v adresáři jeho instalace. Symbolové knihovny mají příponu ".lib" a obsahují více součástek z jedné kategorie. V každém symbolu je uveden odkaz na footprint, nebo skupinu footprintů, které jsou umístěny v jiném adresáři než je adresář symbolů. Adresář footprintů je opět dělen dle kategorií, zde však už připadá vždy jeden soubor na jeden footprint. V souboru footprintu je uveden odkaz na odpovídající 3D model, který je umístěn v podadresáři daného footprintu. Struktura vestavěné knihovny KiCADu je uvedena na obrázku 2.3.

Význam jednotlivých typů souborů je následující:

- .lib = soubor knihovny schématického symbolu
- .dcm = obsahuje popis, aliasy a klíčová slova symbolu
- .kicad_mod = soubor footprintu
- .step = 3D model používaný pro export
- .wrl = 3D model používaný pro vestavěný 3D prohlížeč

[16] `https://github.com/easyw/RF-tools-KiCAD`

```
kicad
  |
  +- library
  |   |
  |   +- Connector.lib
  |   +- Connector.dcm
  |   +- ...
  |
  +- modules
      |
      +- Connector.pretty
      |   |
      |   +- ConnectorXXX.kicad_mod
      |   +- ...
      |
      |
      +- packages3d
          |
          +- Connector.3dshapes
              |
              +- ConnectorXXX.step
              +- ConnectorXXX.wrl
              +- ...
```

Obr. 2.3: Struktura vestavěné knihovny KiCADu

3

Nástroje pro podporu komunitního vývoje

Komunitní vývoj je takový druh vývoje, ve kterém se na řešení určitého problému podílí větší množství pracovníků i organizací v rámci méně formálně definované skupiny (Tím se liší od vývoje konsorciálního, kde jsou skupiny jasně definované). Organizace poskytují na řešení problému své pracovníky a případně i finance. Tento druh vývoje je typický také pro univerzity [10].

Pro realizaci komunitního vývoje je třeba mít prostředky, které umožní efektivní spolupráci mnoha pracovníků na jednotlivých částech určitého problému.

Jednou ze základních funkcionalit, která je nezbytná pro tento vývoj, je bezpochyby možnost správy revizí. Tato umožňuje spravovat změny provedené jednotlivými pracovníky v daném projektu. Každá změna má své označení a je možné ji zpětně dohledat, případně vrátit část projektu, které se změna týkala, do původního stavu.

Další velmi užitečnou funkcionalitou je možnost označovaná jako „issue tracker“. V případě vzniku problému v rámci daného projektu, který odhalí libovolný vývojář či uživatel, je možné problém nahlásit s její pomocí tento problém komunitě, případně konkrétní osobě. V rámci tohoto systému pro sledování chyb se zpravidla uchovává informace o čase, druhu a závažnosti chyby, jejím popisu a zároveň i o člověku, který ji nahlásil a který ji má řešit.

Pro správnou funkci celého systému vývoje je nutné mít ošetřeno, že při pokusu dvou pracovníků o úpravu toho samého úseku projektu nedojde k destruktivnímu konfliktu. Různé systémy k tomuto problému přistupují odlišně, principiálně nejjednodušší způsob je uzamykání souborů. Toto zajistí, že vždy jen jeden vývojář může upravovat tentýž soubor, takže ke konfliktu dojít z principu nemůže. Dalším přístupem je „slučování verzí“, v angličtině „version merging“. Tento již umožňuje vícenásobný přístup k jednomu stejnému souboru od více uživatelů, přičemž změny provede postupně podle pořadí, ve kterém přišly. Může se jednat o náročnou operaci, která je ne vždy jednoduše řešitelná a mnohdy vyžaduje i manuální zásah.

Zpravidla je potřeba mít zajištěnu určitou ochranu, aby nemohlo dojít k neautorizo-

vanému přepsání programu nebo jeho části. Proto jsou v systémech správy verzí různá práva uživatelů, která následně omezují jejich pravomoci. Může jít kupříkladu o právo „vývojář“, které bude umožňovat libovolné přepsání určité části programu, nebo naopak o právo „jen pro čtení“, které žádné úpravy nebude umožňovat. Tato je zároveň možné přidělovat pro jednotlivé pracovníky s různou granularitou, například na různé části projektů.

3.1 Systémy správy revizí

Existuje mnoho různých verzovacích systémů, když se zaměříme na open source nástroje, byly by to například Concurrent Versions System (CVS), Apache Subversion (SVN), Git, Mercurial a další. Každý z těchto systémů má své výhody a nevýhody, kdy nejvýznamnější budou popsány níže. Jako důležité parametry hodnocení jsem zvolil rok posledního vydání, model repozitáře (zda-li jde o systém distribuovaný nebo o systém s centralizovaným serverem) a rozšířenost mezi uživateli. Tyto parametry jsou pro vybrané nástroje porovnány v tabulce 3.1.

3.1.1 Systém CVS

Systém CVS je dlouhodobě zavedený a praxí ověřený. V počátcích neumožňoval práci na jednom kódu více uživatelům a větvení projektů, poslední verze jej již umožňuje. Systém CVS byl dlouhodobě používán, nyní již však není příliš udržovaný a zastarává. V porovnání s dále uvedenými systémy se vyznačuje nízkým výkonem.

3.1.2 Systém SVN

Tento systém vychází ze systému CSV, je jeho potomek. Je mu tedy do značné míry podobný. Podléhá však jiné licenci, zatímco CVS využívá licenci GNU (General Public License), systém SVN využívá licenci Apache. Na rozdíl od CVS umožňuje lepší větvení projektu. Jak systém CVS, tak systém SVN využívají centralizované servery a nejsou distribuované.

3.1.3 Systém Git

Git umožňuje velmi snadné vytváření nových větví projektu, označované jako "branche". Je možné vytvořit si větev nějakého projektu, na které lze nezávisle pracovat a následně, pokud je uznáno za vhodné, připojit novou větev k té hlavní zpět. Git automaticky identifikuje rozdíly a v případě konfliktů se zeptá uživatele na řešení. To umožňuje, aby více osob pracovalo například na různých částech stejného kódu. Velmi to tedy zjednodušuje týmový vývoj. Je také podstatně rychlejší než systémy CVS a SVN [11].

3.1.4 Systém Mercurial

Systém Mercurial využívá distribuovaného modelu, stejně jako Git. To umožňuje jejich vyšší rychlost. Mercurial je značně podobný systému Git, na rozdíl od něj je však psán v jazyce Python a uživatelsky je jednodušší [11].

3.1.5 Systém Bazaar

Bazaar podporuje na rozdíl od výše uvedených jak distribuovaný, tak centralizovaný model repozitáře. Záleží tedy na uživateli a projektu, co je pro něj vhodnější. Bazaar je psán v jazyce Python a byl vyvinut v rámci projektu GNU.

Systém	Uživatelská základna [12]	Rok vydání poslední verze	Forma repozitáře
SVN	16,1%	2021	Centralizovaný
GIT	87,2%	2021	Distribuovaný
Mercurial	3,6%	2021	Distribuovaný
CVS	Minoritní	2008	Centralizovaný
Bazaar	Minoritní	2016	Centralizovaný i distribuovaný

Tab. 3.1: Verzovaci systémy

3.2 Online nástroje pro podporu kolaborativního vývoje

Komunitní vývoj značně usnadňuje, jestliže jsou nástroje k dispozici online, jako webová služba. Existuje řada institucí, které tyto služby poskytují. Podobné služby mohou poskytovat přístup k souborovému archivu, nebo propojení s verzovacím systémem, který je tím pádem možné používat v prohlížeči a provádět s jeho pomocí úpravy nebo review. Kromě propojení s verzovacím systémem jsou důležitým parametrem také systémy pro plánování změn nebo sledování chyb (issue tracker, bug tracker), podpora správy uživatelů a uživatelských rolí, a dále nástroje pro správu a řízení projektu. Rozsáhlý přehled a porovnání základních parametrů těchto systémů lze nalézt online[1,2].

V tabulkách 3.2 a 3.3 je provedeno srovnání vybraných parametrů platforem a nástrojů, které splňují základní podmínky - propojení s verzovacím systémem, podpora správy uživatelů a issue/bug tracker. Platformy Github, Gitlab, Sourceforge nebo Bitbucket jsou velmi populární v open-source komunitě[3], kde mají významnou uživatelskou zák-

[1] `https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems`
[2] `https://en.wikipedia.org/wiki/Comparison_of_source-code-hosting_facilities`
[3] `https://insights.stackoverflow.com/survey/2020#technology-collaboration-tools`

ladnu[4,5,6,7]. Po dohodě s vedoucím práce byly vybrány další nástroje reprezentující různé přístupy nebo významné open-source komunity. Nástroj Savannah[8] je spojen s projektem GNU, platforma Launchpad[9] s projektem Ubuntu. Nástroj Gitea[10] představuje méně komplexní nástroj zaměřený na podporu konkrétního verzovacího systému, nástroj Bugzilla[11] je spojen s projektem Mozilla a je zaměřen primárně na sledování chyb, podobně jako systém MantisBT[12].

[4] `https://github.com/about`
[5] `https://about.gitlab.com/company/`
[6] `https://sourceforge.net/about`
[7] `https://bitbucket.org/blog/bitbucket-cloud-5-million-developers-900000-teams`
[8] `https://savannah.gnu.org/`
[9] `https://launchpad.net/people`
[10] `https://gitea.io/en-us/`
[11] `https://www.bugzilla.org/`
[12] `https://www.mantisbt.org/`

Systém	Issue tracking	Bez reklam	Wiki	Soukromé repozitáře	Správa uživatelských rolí
Github	Ano	Ano	Ano	Ano	Ano
Gitlab	Ano	Ano	Ano	Ano	Ano
Launchpad	Ano	Ano	Ne	Ano	Ano
BitBucket	Ano	Ano	Ano	Ano[13]	Ano
Savannah	Ano	Ano	Ne	Ne	Ano
Sourceforge	Ano	Ne	Ano	Ano	Ne
Gitea	Ano	Ano	Ano	Ano	Ne
Bugzilla	Ano	Ano	Ano	Ano	Ano
MantisBT	Ano	Ano	Ano	Ano	Ano

Tab. 3.2: Verzovaci systemy I

Systém	Zdarma	Typ	Podporované verzovací systémy
Github	Ne[14]	Jen služba	Git, částečně SVN[15]
Gitlab	Částečně[16]	Software	Git
Launchpad	Ne	Software	Git, Bazaar, import z CVS, SVN a Mercurial
BitBucket	Ne	Služba	Git
Savannah	Ano	Software	CVS, Git, SVN, Bazaar, Mercurial
Sourceforge	Ano	Jen služba	Git, SVN, Mercurial
Gitea	Ano	Software	Git
Bugzilla	Ano	Software	Git, Mercurial, Bazaar, CVS, SVN, Perforce, Accurev
MantisBT	Ano	Software	Git, Mercurial, CVS, SVN

Tab. 3.3: Verzovaci systemy II

3.3 Projekt SmartCAMPUS ZCU

Všechny nástroje uvedené výše v tabulkách 3.1, 3.2 a 3.3 (a nejen ty) splňují základní požadavky důležité pro podporu vývoje knihovny součástek. Existuje tedy velmi mnoho alternativ a tato problematika je značně široká.

[13]Zdarma maximálně pro 5 uživatelů
[14]Pro Open-Source projekty a vzdělávání zdarma
[15]Repozitáře GitHub mohou být přístupné i z klienta SVN
[16]Gitlab FOSS je zdarma - verze bez neveřejných kódů

Jelikož cílem této práce je vytvořit knihovnu součástek kompatibilní s projektem SmartCAMPUS ZCU, je důležité aby byly používány totožné nástroje. V projektu SmartCAMPUS se používá verzovací systém Git, s online nástrojem Github. Na základě předchozího porovnání je patrné, že oba tyto nástroje splňují podmínky nutné pro efektivní týmovou spolupráci, jsou vzájemně dobře integrované, a je tedy možné s jejich využíváním pokračovat. Pro tuto práci tedy budou oba tyto nástroje taktéž využity.

4

Návrh obecné metodiky pro správu knihovny

Jak již bylo zmíněno, cílem této práce je s použitím výše uvedených nástrojů vytvořit metodiku, která umožní efektivní a deterministickou správu knihovny. Při návrhu bylo důležité užívat principu KISS („keep it stupid simple“ [6]). Tento zajistí jednoduché používání, které bude do značné míry intuitivní a logické i pro uživatele, který s celou prací není detailně seznámen.

4.1 Součásti návrhu metodiky

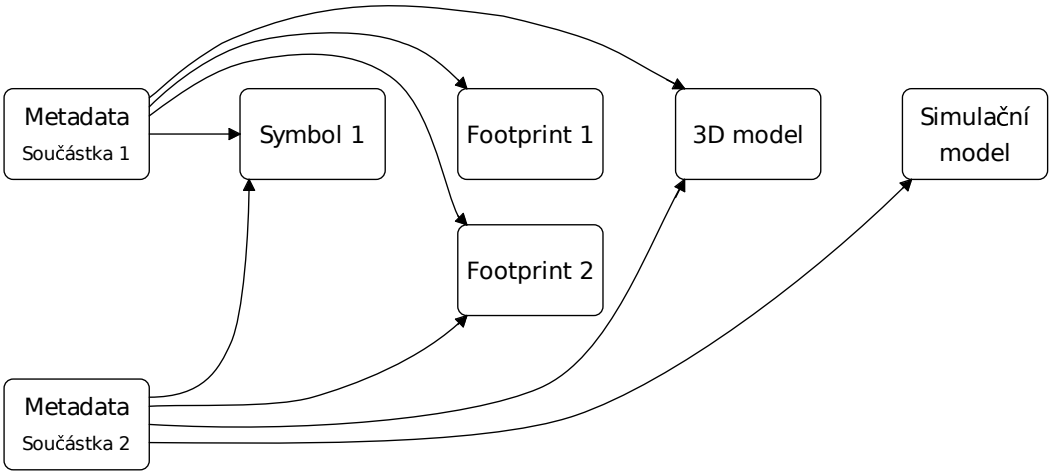
Návrh metodiky pro správu knihovny obecně zahrnuje řešení nejprve definice struktury knihovny a následně postupu ověření součástky v knihovně.

4.1.1 Definice struktury knihovny

V této části je třeba definovat základní strukturu, kterou bude knihovna používat. Abychom mohli říct, jak má knihovna vypadat, je napřed důležité osvětlit, z čeho je vlastně složena součástka v knihovně a čím je ve své podstatě definována.

Součástka jako taková je definována svým výrobním číslem a metadaty, vycházejícími z datasheetu od výrobce. V tom jsou obsaženy informace o pouzdře, elektrických parametrech i mechanických vlastnostech součástky. Tyto informace jsou postačující k základnímu popsání součástky. Dále je ovšem mnohdy vhodné mít dodatečné informace, jako například model pro elektrickou simulaci či informace o dodavatelském řetězci. Struktura obecné knihovny je navržena tak, aby se předešlo zdvojování jakéhokoliv obsahu. Struktura je naznačena na obrázku 4.1.

Z obrázku je patrné, že součástka jako taková je definována jen metadaty, na která jsou navázány všechny součásti, tedy symbol, footprint, 3D model i případný simulační model. Tato struktura a princip ne-zdvojování dat má kromě zjevné výhody zmenšení prostorové náročnosti ještě další benefit. Ten spočívá v jednoduchosti odstraňování chyb.



Obr. 4.1: Ideální struktura obecné knihovny

V případě, že se totiž například na footprintu najde závada, odstraní se na jednom místě a všechny součástky, které daný footprint používají, již budou mít okamžitě k dispozici opravenou verzi. Nedojde tedy k neúmyslnému ponechání nějaké starší verze s chybou u určité součástky.

Struktura obecné knihovny je více či méně využívána jednotlivými návrhovými systémy, přičemž každý systém má svá specifika či limitace, jak bylo patrné z popisu knihoven v kapitole 2. Vnitřní struktura databázové knihovny systému Altium odpovídá výše uvedené ideální struktuře.

4.1.2 Postup ověření součástek v knihovně

Běžný návrh součástky v knihovně zahrnuje zejména tvorbu schématického symbolu, footprintu a případně jemu odpovídajícímu 3D modelu. Bez dalšího ověření ale může velmi snadno dojít k chybě, která se odhalí až ve výsledném návrhu, kdy už může být pozdě. Proto tento postup není vhodné aplikovat samostatně. Pro následný proces ověřování jsou definovány následující pojmy: *verifikace*, čímž rozumíme proces zjišťování souladu chování navrženého zařízení (nebo jeho součásti) se specifikací [19] a *validace*, čímž rozumíme proces zjišťování souladu chování navrženého zařízení s reálnými požadavky na zařízení [19].

Po návrhu určité části knihovny (komponenty, součástky,...) a kontrole provedené samotným návrhářem (toho si označme *Design inženýr*) následuje proces verifikace. Tento proces je prováděn jiným člověkem, než autorem. Člověk který provádí verifikaci se nazývá *Verifikační inženýr* (verifikátor). Verifikátor obdrží od autora samotnou součástku a přiložený soubor s komentáři či doplňujícími informacemi. Tento soubor se obvykle označuje „log“. Verifikátor provede proces verifikace a výsledkem jeho práce je buď hlášení o chybách, které jde zpět k autorovi, nebo zverifikovaná součástka. Po procesu verifikace následuje proces validace, osoba která provádí proces validace se nazývá *validační inženýr* (validátor). Validátor obdrží zverifikovanou součástku a případné doplňující informace od verifikátora. Validátor provede proces validace a jeho výsledkem jeho práce je buď hlášení

o chybách, které jde zpět k autorovi, nebo zvalidovaná součástka. Zvalidovanou součástku lze označit za maximálně ověřenou. Pořadí těchto procesů je znázorněno na obrázku 4.2.

V případě, že je třeba ověřit externí komponentu, je proces ověření v podstatě identický s tím rozdílem, že do něj vstupují na začátku externí data, nikoliv data vytvořená designerem. Tento proces se pak označuje jako *kvalifikace*.

Obr. 4.2: Obecný proces ověření součástky

V případě, že je potřeba ověřit součástku, která se skládá z více částí, navzájem propojených, je třeba ověření provádět postupně dle vnitřních závislostí. Aby bylo možné ověřit komponentu, která závisí na něčem jiném, je potřeba napřed ověřit tyto závislosti a až pak přikročit k ověření samotné komponenty.

Aby byl proces co nejspolehlivější, je vhodné aby v případě design inženýra, verifikačního inženýra a validačního inženýra šlo o rozdílné osoby. Minimálně v případě design inženýra a verifikačního inženýra je to nezbytně nutné, ideální však je, když jsou všichni tři naprosto nezávislí.

5

Implementace navržené metodiky pro KiCAD

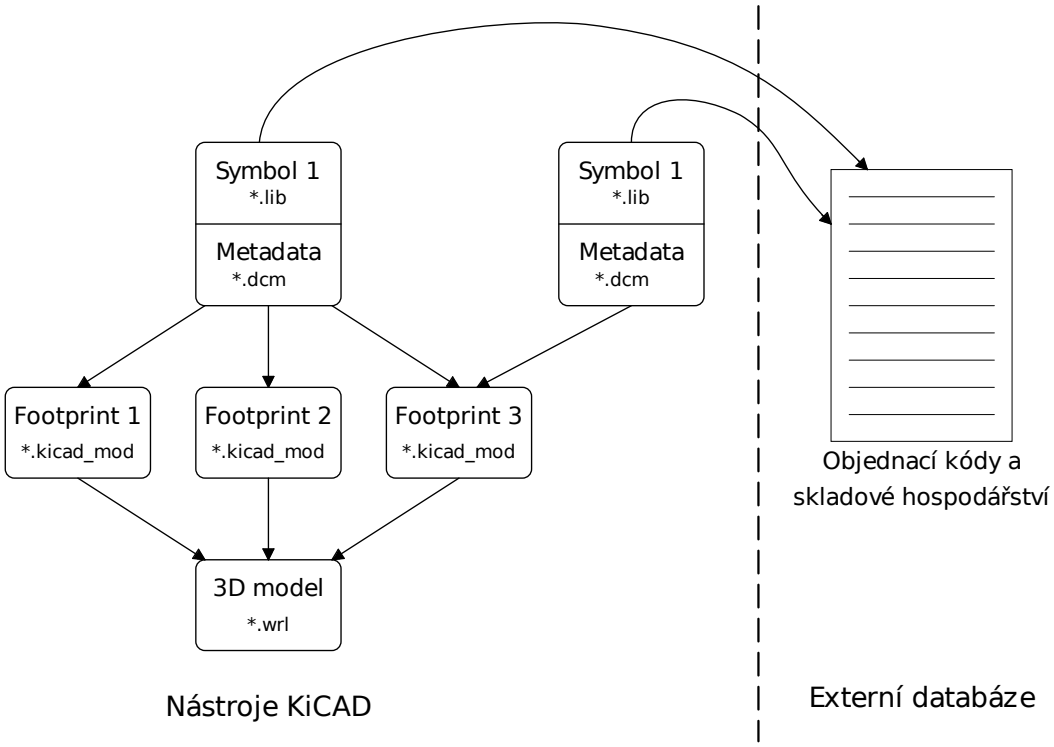
Na základě obecné metodiky a znalosti organizace databáze bylo možné postupovat ke konkrétní implementaci. Z dosavadního zkoumání vyplynulo několik důležitých požadavků: (1) symbolová knihovna KiCADu je sdružená do jednoho souboru odpovídajícímu kategorii. To ale není ideální pro verifikaci, kdy je vhodné mít oddělené jednotlivé symboly. Je tedy vhodné mít ve fázi vývoje knihovny soubory oddělené. Pro běžné používání knihovny toto však praktické není, proto je nutné umožnit uživateli generování sloučených knihoven odpovídajícího zařazení podle jeho výběru; (2) každá komponenta součástky by se měla verifikovat zvlášť a pokud možno jednotným způsobem; (3) každá komponenta u sebe musí nést informaci o svém zařazení v knihovně (stav ověření) ideálně tak, aby to bylo možné jednotně strojově zpracovat; (4) z důvodu umožnění verifikace součástky jako celku včetně příslušných metadat, je nutné součástky tvořit jako takzvaně „atomické“. To znamená, že každý schématický symbol má přesně definovaný použitý footprint (nebo množinu footprintů) již v rámci knihovny[1].

5.1 Struktura navržené souborové databáze pro knihovnu

V návrhu struktury knihovny je třeba brát v úvahu hierarchické závislosti jednotlivých komponent knihovny. Tyto závislosti a základní struktura naší navržené knihovny KiCADu jsou znázorněny na obrázku 5.1.

Z výše uvedených požadavků vyplynulo uspořádání souborové databáze, znázorněné na na obrázku 5.2. Všechny části databáze jsou rozděleny dle funkčních kategorií, inspirovaných standardem RICE. V případě schématického symbolu je jeden komponent reprezentován jedním adresářem v adresáři příslušné kategorie. Jeho název je shodný se jménem součástky a odpovídá výrobnímu číslu, případně identifikátoru výrobce, jelikož

[1] `https://kicad.org/libraries/klc/G2.1`



Obr. 5.1: Struktura navržené KiCAD knihovny a závislosti mezi komponentami

je zde požadavek na unikátnost. Formálně se jedná o samostatnou knihovnu KiCADu. Význam souborů .status a drc.log bude popsán v následující části.

V případě footprintu je situace mírně odlišná, názvy adresářů kategorií jsou zakončeny příponou .pretty a obsahují přímo soubory footprintů, které jsou pojmenovány v souladu s KLC. Byl zaveden adresář pro uchování dodatečných informací o footprintu, který je umístěn dle obrázku 5.2. Tento adresář má stejné jméno jako footprint, s příponou .data.

U adresáře 3D modelů je situace obdobná jako u schématických symbolů s tím rozdílem, že zde není vyžadován soubor drc.log.

Adresář tools obsahuje pomocné skripty, zbývajícím adresářům se budu věnovat postupně v dalším textu.

```
kicad-lib
  |
  +- symbol
  |   |
  |   +- Connector
  |   |   |
  |   |   +- ManufPartNumber
  |   |       |
  |   |       +- .status
  |   |       +- ManufPartNumber.lib
  |   |       +- ManufPartNumber.dcm
  |   |       +- drc.log
  |   +- Discrete
  |   +- ICs_Analog
  |   +- ICs_Digital
  |   +- Mechanic
  |   +- Modules
  |   +- Net
  |   +- Passive
  |   +- Power
  |   +- PXI
  |   +- Relays
  |   +- Switches
  |
  +- footprint
  |   |
  |   +- Connector.pretty
  |   |   |
  |   |   +- ConnectorXXX.kicad_mod
  |   |   +- ...
  |   |   +- ConnectorXXX.data
  |   |       |
  |   |       +- .status
  |   |       +- .drc.log
  |   +- Discrete.pretty
  |   +- ...
  |
  |
  +- 3d_model
  |   |
  |   +- Connector
  |   |   |
  |   |   +- ConnectorXXX.3dshapes
  |   |       |
  |   |       +- .status
  |   |       +- ConnectorXXX.wrl
  |   |       +- ConnectorXXX.step
  |   +- Discrete
  |   +- ...
  |
  +- build
  +- work
  +- templates
  +- tools
```

Obr. 5.2: Databázová struktura vytvořené knihovny

5.1.1 Stav součástky

Na základě obecné metodiky byly stanoveny čtyři úrovně, na kterých se může součástka či její komponenta vyskytovat. Nultá úroveň je označena „created“ a zahrnuje právě vytvořené komponenty nebo ty, na kterých se pracuje. První úroveň je označena „new“ a označuje komponenty, které prošly ověřením u návrháře a jsou určeny k verifikaci. Tyto komponenty není možné použít v návrhu. Druhá úroveň je označena „verified“ a je přiřazena komponentě, která úspěšně prošla procesem verifikace. Třetí a nejvyšší úroveň je označena „validated“ a týká se komponenty, která byla úspěšně použita v návrhu a odzkoušena v praxi. Aby bylo možné uchovávat informace o úrovních dané komponenty, byl definován soubor .status, který je příslušně umístěn. Jeho struktura a možné hodnoty jednotlivých proměnných jsou popsány na obrázku 5.3.

```
DRC = {pass | fail | void}
VRC = {pass | fail | void}
LibStatus = {created | new | verified | errorVRC | errorPCB | errorMAN | validated}
```

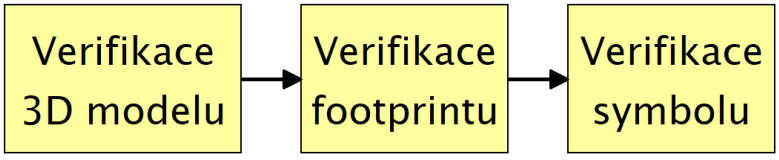
Obr. 5.3: Struktura souboru .status

kde DRC značí kontrolu provedenou návrhářem (design inženýrem) a VRC značí kontrolu provedenou verifikačním inženýrem.

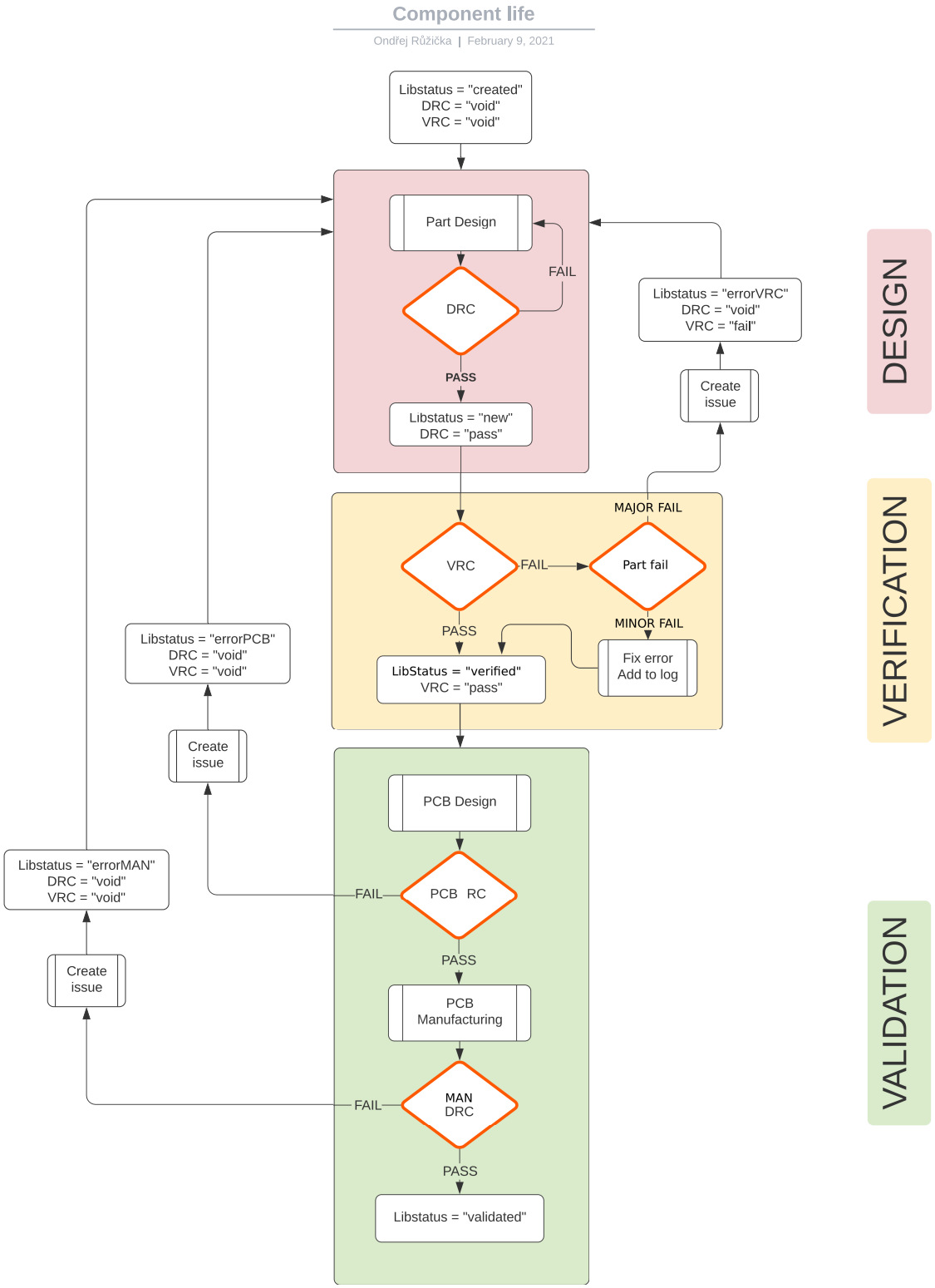
5.2 Proces ověření součástky

Na základě obecného procesu definovaného v předchozí kapitole byl navržen konkrétní postup ověření, který je zobrazen na obrázku 5.5. Tento vyjadřuje celý proces života komponenty od jejího vytvoření design inženýrem až po finální ověření validačním inženýrem. Pokud je součástka složená z více komponent (schématický symbol, footprint, 3D model,...) projde tímto procesem každá tato komponenta, s přihlédnutím k hierarchickým závislostem.

Na obrázku 5.4 je naznačena zmíněná situace, kdy na sobě hierarchicky závisí schématický symbol, footprint a 3D model, přičemž 3D model je z hlediska závislostí nejníže – je tedy potřeba jej ověřit jako první.



Obr. 5.4: Postup verifikace součástky



Obr. 5.5: Diagram zachycující proces schvalování součástky

5.2.1 Proces návrhu

Vstupem design inženýra (návrháře, autora) jsou buď informace o součástce, kterou tvoří (datasheet, označení výrobce), nebo jím vytvořená součástka, která neprošla nějakou fází ověření, spolu s příslušným issue, ve kterém je popsána podstata problému. Jako podklad pro příslušné kontroly používá design inženýr seznam návrhových pravidel, která jsou vypsána v příloze B.

Za návrh součástky je zodpovědný design inženýr, který návrh provede na základě podkladů pro danou součástku či komponentu. Po dokončení návrhu součástky či její komponenty nad ní spustí návrhář skripty na kontrolu pravidel KLC. Jejich výstup následně vloží do souboru drc.log a případná porušení patřičně okomentuje, smysluplná porušení pravidel jsou tedy povolena. Následně ověří, zda-li jsou splněna pravidla SC (SmartCAMPUS), která jsou dostupná v repozitáři knihovny a v příloze této práce (příloha B). Poté může v souladu s výše zmíněným diagramem vývoje vyplnit soubor .status a součástku či komponentu tak schválit k verifikaci.

Výstupem design inženýra je vytvořená součástka a soubor drc.log, obsahující okomentovaný výpis z automatické kontroly.

5.2.2 Proces verifikace

Vstupem verifikačního inženýra je součástka vytvořená design inženýrem a soubor drc.log. Jako podklad pro příslušné kontroly používá verifikační inženýr seznam návrhových pravidel, která jsou vypsána v příloze B.

Verifikační inženýr projde především soubor drc.log a posoudí, jestli jsou daná porušení pravidel opravdu nezbytná. Následně si otevře součástku v KiCADu a provede manuální kontrolu. Pokud neshledá žádný problém, označí součástku či komponentu v souladu s diagramem jako „verified“. Pokud problém najde, založí na GitHubu issue a součástku vrátí návrháři. Následně se proces opakuje.

V případě, že verifikační inženýr najde problém, který nemá zásadní vliv na funkčnost či vlastnosti dané komponenty, může provést jeho opravu, aniž by vracel komponentu návrháři. Tento postup se označuje jako „minor fix“ a je nutné ho takto označit v příslušném commitu. Minor fix je definován jako oprava „minor erroru“, který je jako pojem zaveden v úvodním dokumentu projektu SmartCAMPUSZCU/KiCAD-lib[2]. Postup využívající minor fix omezí nadbytečné vracení součástky návrháři v případech jako je třeba očividný překlep.

Výstupem verifikačního inženýra je buď zverifikovaná součástka s příslušným drc.log souborem, nebo součástka označená jako chybná a issue, ve kterém je popsána podstata problému: jsou vyjmenována porušená pravidla, případně specifikovány další problémy.

[2] `https://github.com/SmartCAMPUSZCU/KiCAD-lib`

5.2.3 Proces validace

Proces validace má dvě fáze. Vstupem validačního inženýra je v první fázi součástka zverifikovaná verifikačním inženýrem a soubor drc.log. Ve druhé fázi je vstupem fyzicky vyrobená a osazená deska, spolu se souborem pravidel pro validaci.

V první fázi součástka prochází ověřením při návrhu desky, tedy pomocí DRC kontrol v EDA nástroji. Pokud v této fázi dojde k odhalení chyby validačním inženýrem, je součástka označena dle diagramu 5.5 a stejně jako u procesu verifikace je vytvořeno issue s popisem podstaty problému obsahující zejména seznam porušených pravidel, případně výstup DRC nástroje. Součástka se pak vrací design inženýrovi.

Druhá fáze validace probíhá již po fyzickém vyrobení desky. Validační inženýr je v tomto případě zodpovědný za finální kontrolu správnosti součástky jako celku. Validační inženýr musí být člověk kvalifikovaný pro toto posuzování a provádí kontroly dle souboru validačních pravidel. Tato pravidla nebyla v rámci této práce specifikována. Pokud dojde při této kontrole k nalezení chyby, postup je stejný jako v předchozích případech a to založení issue s popisem chyby a vrácení součástky design inženýrovi.

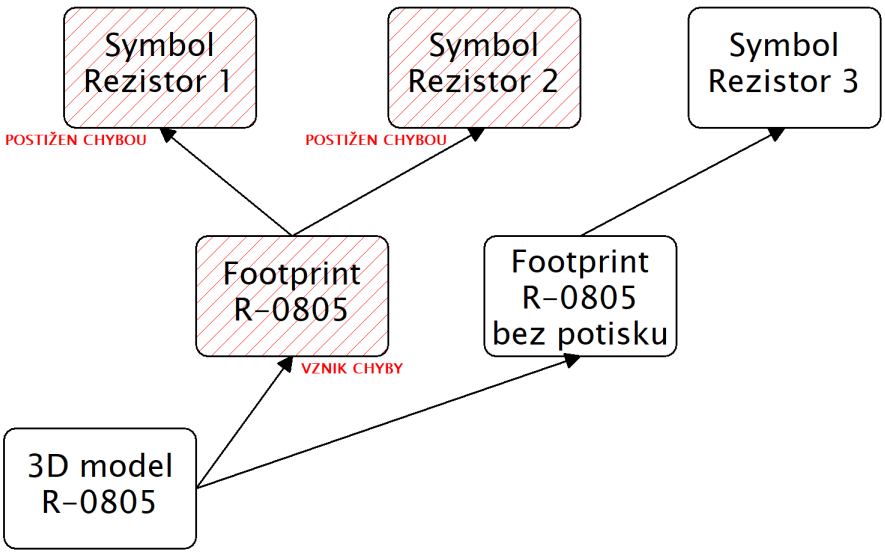
Výstupem validačního inženýra je buď zvalidovaná součástka s příslušným drc.log souborem, nebo součástka označená jako chybná a issue, ve kterém je popsána podstata problému.

5.2.4 Šíření chyby v knihovně

V případě, že se uvnitř knihovny vyskytují závislosti, obzvláště víceúrovňové, je potřeba definovat jak dochází k šíření chyby. Je patrné, že chyba vzniklá na komponentě v nižší hierarchické úrovni ovlivní všechny komponenty ve vyšší hierarchické úrovni, které jsou na ni navázány. To je třeba brát v úvahu při návrhu knihovny a způsobu ověřování součástek. Libovolnou komponentu ovlivněnou chybou je třeba považovat za potenciálně poškozenou a je nutné provést na ní opětovně proces ověření. Šíření chyby je znázorněno na obrázku 5.6, na stejném modelu hierarchických závislostí jako v předchozí sekci (3D model ← Footprint ← Symbol).

5.3 Generování uživatelské knihovny ze souborové databáze

Jak bylo výše zmíněno, pro účely vývoje knihovny je vhodné mít jednotlivé soubory rozdělené na úrovni součástek. To však není praktické při používání knihovny. Proto jsem vytvořil skript *mergeLibFiles.py*, který slouží ke spojení všech součástek příslušné kategorie do jednoho souboru obdobně, jako je tomu u vestavěných KiCAD knihoven. Návrhář tak má k dispozici hierarchickou knihovní strukturu odpovídající rozdělení součástek do definovaných kategorií. Tento skript je umístěn v adresáři tools a má tři vstupní



Obr. 5.6: Hierarchické závislosti komponent součástky a naznačení šíření chyby

parametry. Tyto definují výstupní soubor (-o), adresář kategorie v databázi (-d) a úroveň součástky (-s). Poslední zmíněný parametr má tři možné varianty, buď dojde ke generování pouze verifikovaných, pouze validovaných a nebo jak verifikovaných tak validovaných součástek. Výchozí volbou pro návrh běžné desky by bylo generování verifikovaných i validovaných součástek, záleží na konkrétní potřebě každého uživatele a procesu. Po spuštění skriptu dojde k vygenerování knihovních souborů *.lib a *.dcm. Příklad skriptu při spuštění by mohl vypadat takto:

```
python3 tools/mergeLibFiles.py -d symbol/Connector -o build/verified/Connector -s ver
```

Obr. 5.7: Příklad volání skriptu mergeLibFiles

5.3.1 Automatizované generování výstupních souborů

Ze své podstaty by bylo potřeba volat skript mergeLibFiles nad všemi kategoriemi, aby došlo k vygenerování potřebných knihovních souborů. To je uživatelsky poměrně nepřívětivé. Proto je v hlavním repozitáři soubor *Makefile*, který se zavolá jen s jedním parametrem. Může to být buď „verified“, „validated“ a nebo „all“. Ten automaticky spustí všechny příslušné skripty postupně a vygeneruje výstupní soubory do adresáře „build“. Tento se nezanáší do verzovacího systému a je určen pouze jednomu uživateli na konkrétním zařízení. Pro uživatele je doporučené používat jako základní možnost knihovnu vygenerovanou jako „all“, obsahující tedy verifikované i validované součástky. Pochopitelně záleží na aplikaci daného uživatele.

6

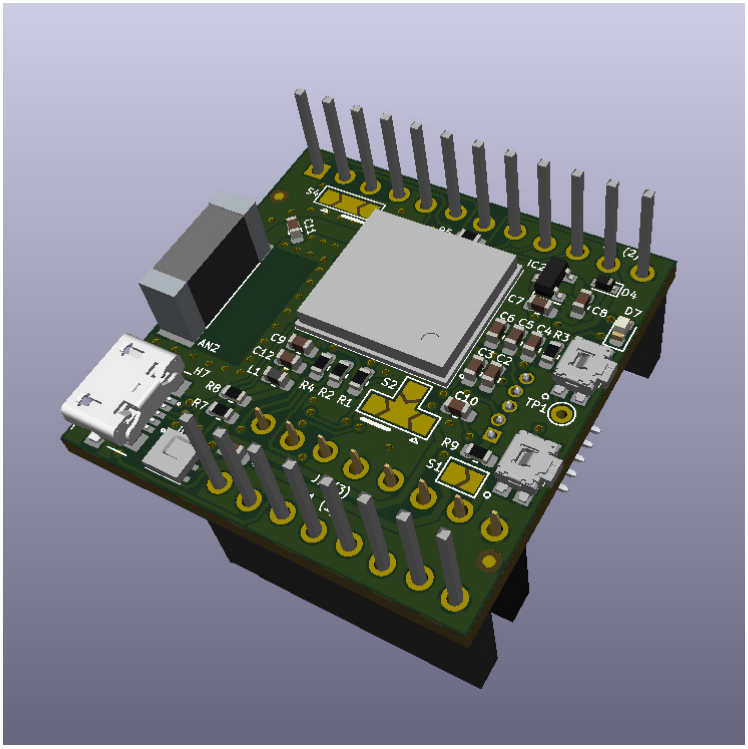
Návrh desky KETCube

Deska KETCube je otevřený projekt vyvíjený na Katedře materiálů a technologií (ZČU KET). Deska samotná je určena pro oblast IoT a slouží především ke snímání a měření. Je vybavena modulem od firmy Murata, který obsahuje mikrokontrolér řady STM32L0 a radiový LoRa modul, který zajišťuje datový přenos. Deska má rozměry 32x32mm a jedná se o dvouvrstvý plošný spoj.

V rámci této práce bylo cílem překreslit na základě poskytnutých podkladů celý projekt KETCube do programu KiCAD jen s použitím vytvořené knihovny. Tento krok byl velice užitečný, jelikož ukázal mnoho nesrovnalostí či problematických detailů v původně vytvořené knihovně i metodice a umožnil tak jejich detekci a odstranění.

Při samotném kreslení plošného spoje byla náročná především rádiová část, jelikož ta vyžadovala speciální plugin pro KiCAD, určený k návrhu RF částí desky. Šlo o plugin KiCAD RF tools[1], který umožňuje vytvářet zakulacené cesty, rozmísťovat kolem nich prokovené otvory a poskytuje několik dalších funkcí, které jsem však nevyužil. Bylo také třeba brát v úvahu správný návrh rozvodu napájení na desce a jeho blokování. Další kritickou součást tvořilo napojení USB konektoru, které bylo třeba vést jako diferenciální pár. Jinak šlo o standardní dvouvrstvý návrh, který byl komplikován jen fyzickými rozměry desky. Na obrázku 6.1 je ukázán 3D pohled navržené desky. Výstupy z návrhu, tedy schéma zapojení, nákres desky plosného spoje a další 3D pohledy, jsou připojeny v příloze A.1.

[1] `https://github.com/easyw/RF-tools-KiCAD`



Obr. 6.1: 3D pohled na desku KETCube

7

Závěr

V rámci této práce byla navržena metodika správy otevřené knihovny s použitím programu KiCAD, verzovacího systému Git a jeho webové nadstavby GitHub, které byly zvoleny jako vhodné. Přesto je metodika navržena tak, aby ji bylo možné použít s libovolným verzovacím systémem a issue trackerem. V rámci práce byla s použitím navržené metodiky vytvořena knihovna součástek potřebných pro navržení desky KETCube.

V rámci této práce jsem vytvořil celkem 37 kompletních součástek, které verifikoval vedoucí práce. Ten vytvořil celkem 11 součástek, které jsem verifikoval já. Většina mnou vytvořených součástek byla určena pro projekt desky KETCube. Vzhledem k počtu navržených a verifikovaných součástek bylo možné celý proces i metodiky doladit tak, aby byla konzistentní a dobře použitelná. Metodika prošla tedy základním ověřením uvnitř týmu.

Pro ověření správnosti a odladění nedostatků knihovny byla navržena deska KETCube, což se podařilo bez závažnějších problémů.

V současné chvíli je knihovna připravena pro další používání. Zatím její používání vyžaduje větší množství manuálních úkonů, které je však v plánu v budoucnu do určité míry zautomatizovat pomocí skriptů. Vývoj základních skriptů proběhl paralelně s touto bakalářskou prací a vedoucí práce plánuje jejich dopracování.

Literatura

[1] ZÁHLAVA, Vít.*Návrh a konstrukce desek plošných spojů: principy a pravidla praktického návrhu.* Praha: BEN - technická literatura, 2010. ISBN 978-80-7300-266-4.

[2] *KiCad Library Convention 3.0.28 [online].*[cit. 2021-03-11]. Dostupné z: https://kicad-pcb.org/libraries/klc/.

[3] *About KiCad [online].*[cit. 2021-03-17]. Dostupné z: https://kicad.org/about/kicad/.

[4] *Gerber Format [online].*[cit. 2021-03-17]. Dostupné z: https://www.ucamco.com/en/gerber.

[5] PETERSON, Zachariah. *Altium PCB library management.* Altium Designer [online]. San Diego, 2019 [cit. 2021-03-11]. Dostupné z: https://resources.altium.com/p/advanced-pcb-library-management.

[6] *KISS principle.* In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-11]. Dostupné z: https://en.wikipedia.org/wiki/KISS_principle.

[7] MAGYAR, John. *New User's Guide to Defining a Library Methodology*In: Https://resources.altium.com/ [online]. La Jolla, CA 92037, USA, 2020, 27.10. [cit. 2021-6-12]. Dostupné z: https://resources.altium.com/sites/default/files/uberflip_docs/file_1157.pdf

[8] HOWIE, Jason, ed. *Altium vault.* Altium Vault [online]. 2017 [cit. 2021-6-13]. Dostupné z: https://www.altium.com/documentation/Vault.

[9] *Comparison of EDA software*In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-16]. Dostupné z: https://en.wikipedia.org/wiki/Comparison_of_EDA_software.

[10] *Community source*In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-06-11]. Dostupné z: https://en.wikipedia.org/wiki/Community_source.

[11] RAWSON, Rob.*2020 Version Control Software Comparison: SVN, Git, Mercurial.* Time Doctor [online]. [cit. 2021-03-11]. Dostupné z: https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/.

[12] *Developer survey results 2018.* Stack overflow [online]. [cit. 2021-06-12]. Dostupné z: https://insights.stackoverflow.com/survey/2018work-_-version-control.

[13] PEHAM, Thomas. *GitLab vs GitHub: Key differences & similarities.* Usersnap [online]. [cit. 2021-03-11]. Dostupné z: https://usersnap.com/blog/gitlab-github/

[14] *Verifying Your Design in Altium Designer [online].* [cit. 2021-03-16]. Dostupné z: https://techdocs.altium.com/display/ADOH/Verifying+Your+Design+in+Altium+Designer.

[15] *PCB Component Footprint Validation Procedure [online].* [cit. 2021-03-16]. Dostupné z: https://wiki.apertus.org/index.php/PCB_Component_Footprint_Validation_Procedure.

[16] *Part Verification on SnapEDA [online].* 2015 [cit. 2021-03-16]. Dostupné z: https://blog.snapeda.com/2015/09/19/part-verification-on-snapeda/.

[17] *SnapEDA's Open Symbol and Footprint Standards Version 1.0 [online].*[cit. 2021-03-16]. Dostupné z: https://www.snapeda.com/standards/.

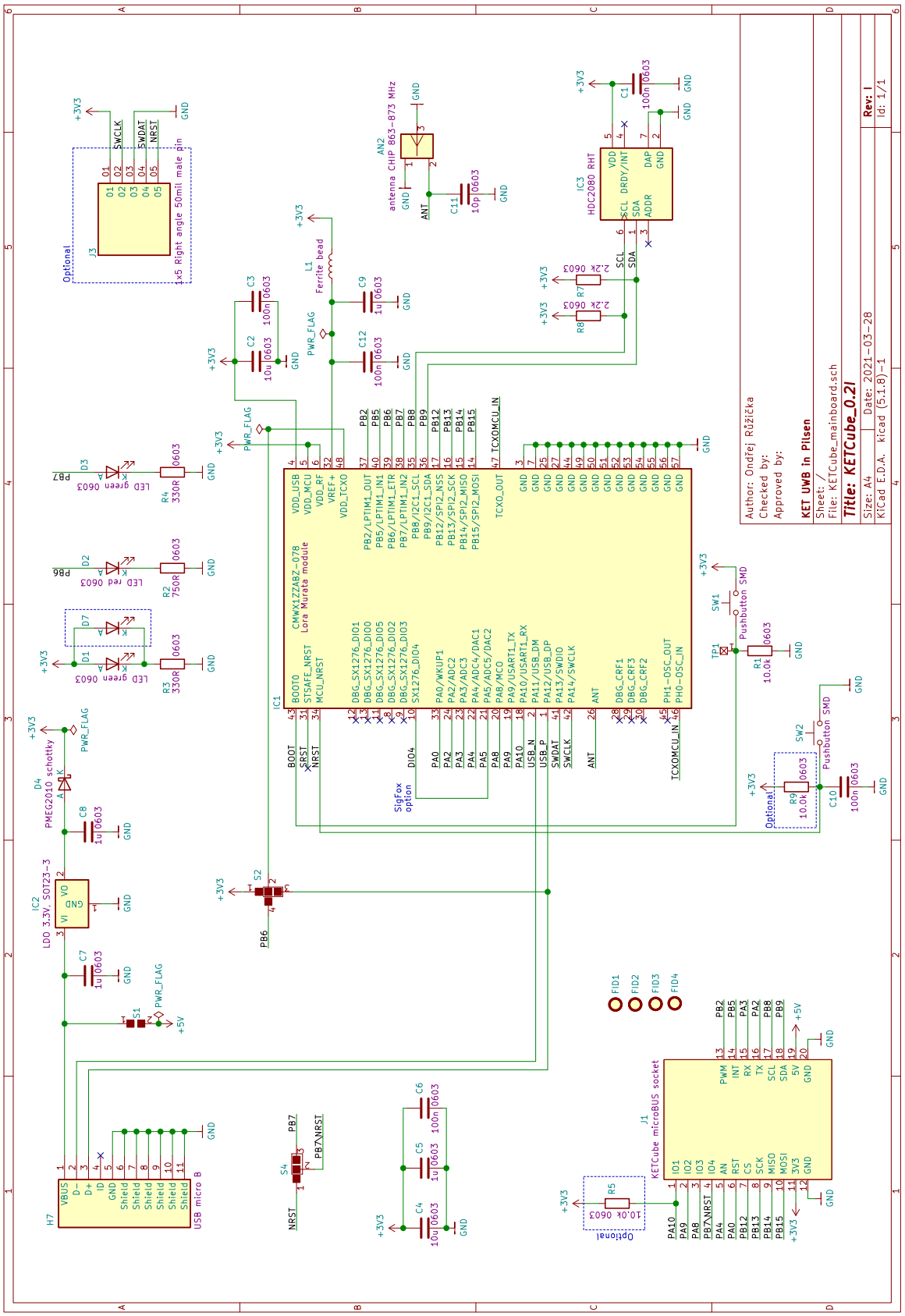
[18] *Component Verification Processes – Vital to Library Accuracy! [online].* 2017 [cit. 2021-03-16]. Dostupné z: https://www.ultralibrarian.com/resources/blog/2017/12/01/component-verification-processes-vital-to-library-accuracy/

[19] *IEEE Standard for System, Software, and Hardware Verification and Validation.* In IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017) , vol., no., pp.1-260, 29 Sept. 2017, doi: 10.1109/IEEESTD.2017.8055462.

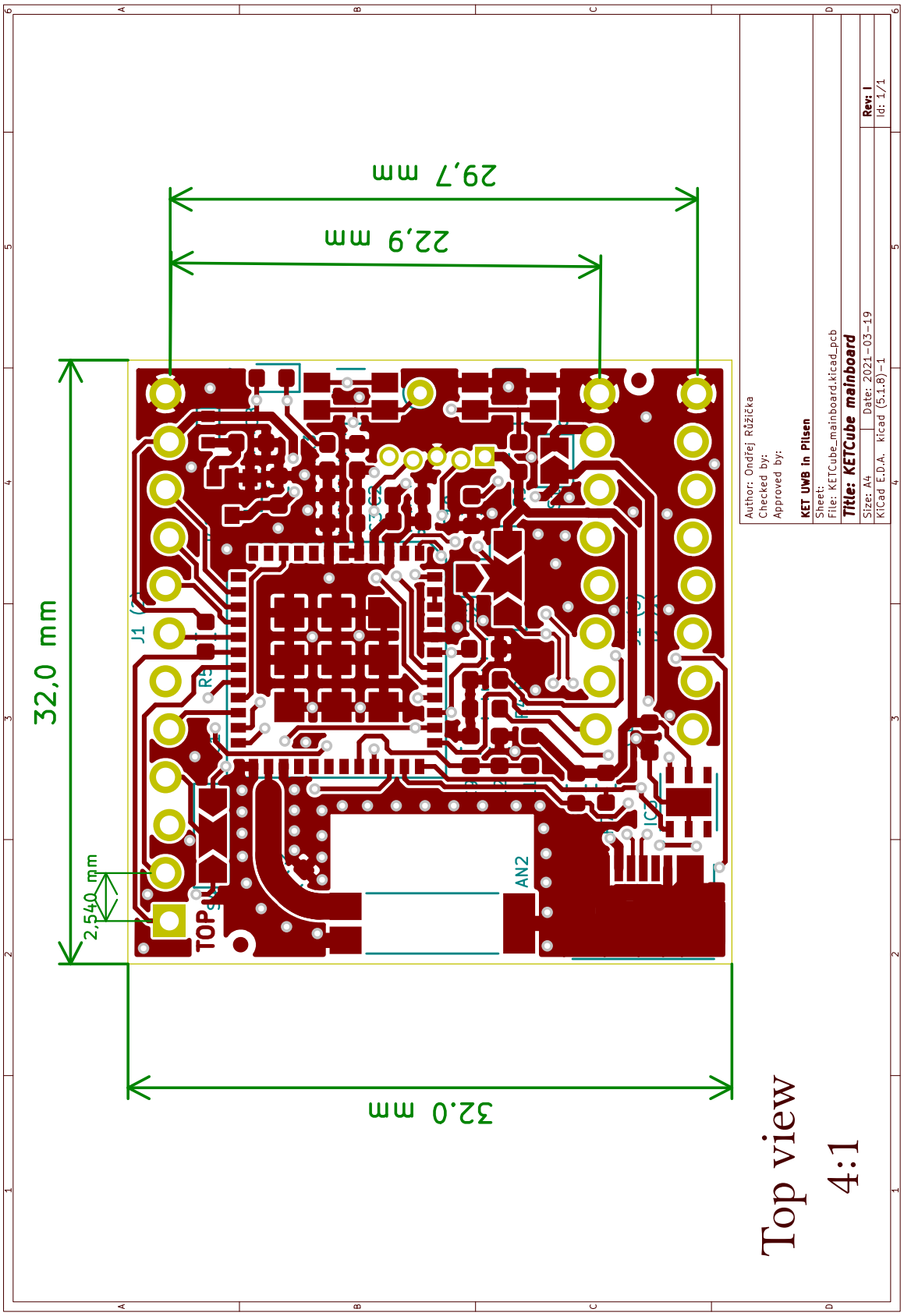
Příloha A

Výstupy z návrhu desky KETCube

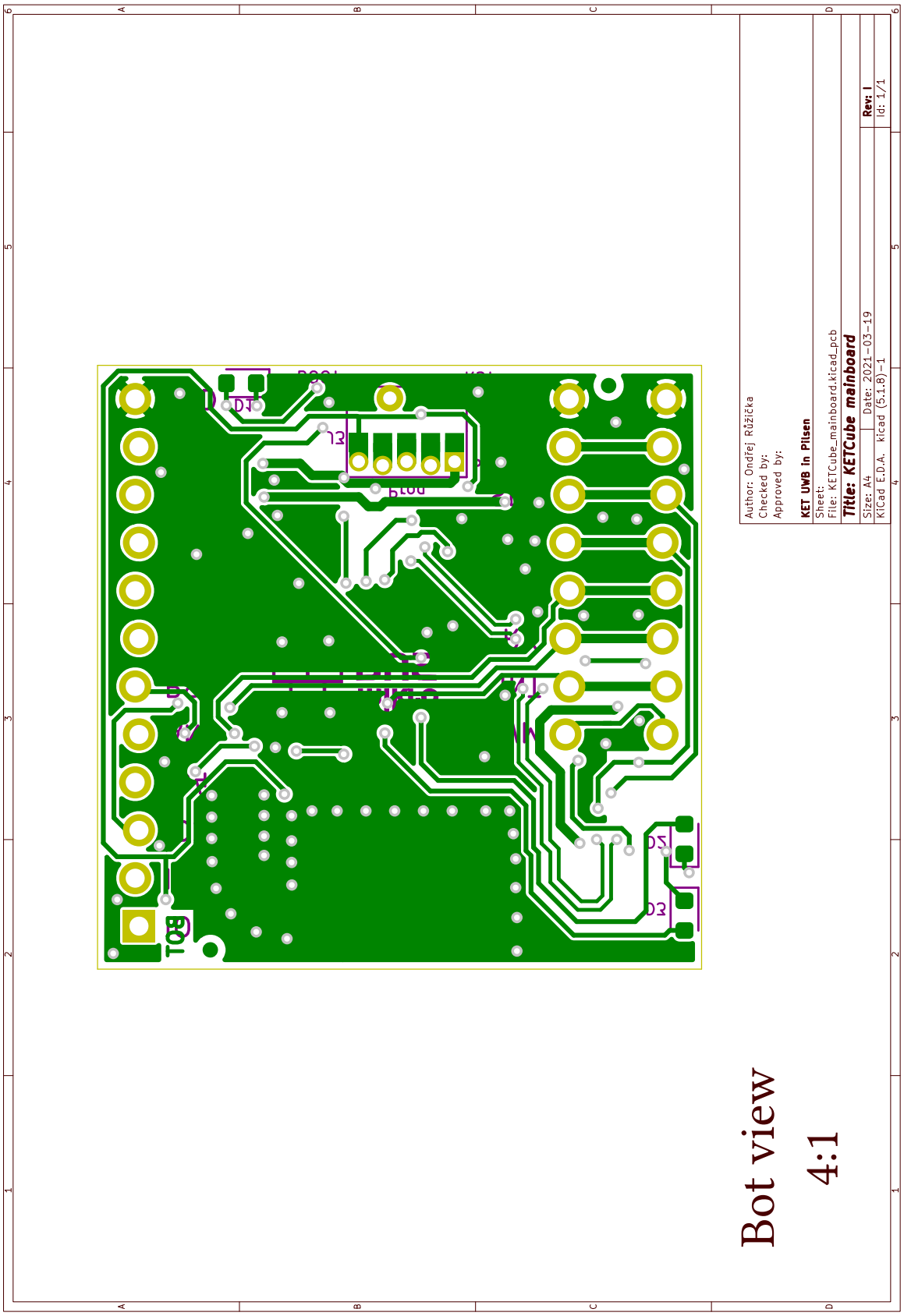
Tato příloha obsahuje výstupy z návrhu desky KETCube, které jsou vygenerovány ze systému KiCAD. Je zde uvedeno schéma zapojení, výkresy horní a spodní strany plošného spoje a osazovací plán. Dále jsou uvedeny 3D výstupy, které zobrazují osazenou desku.



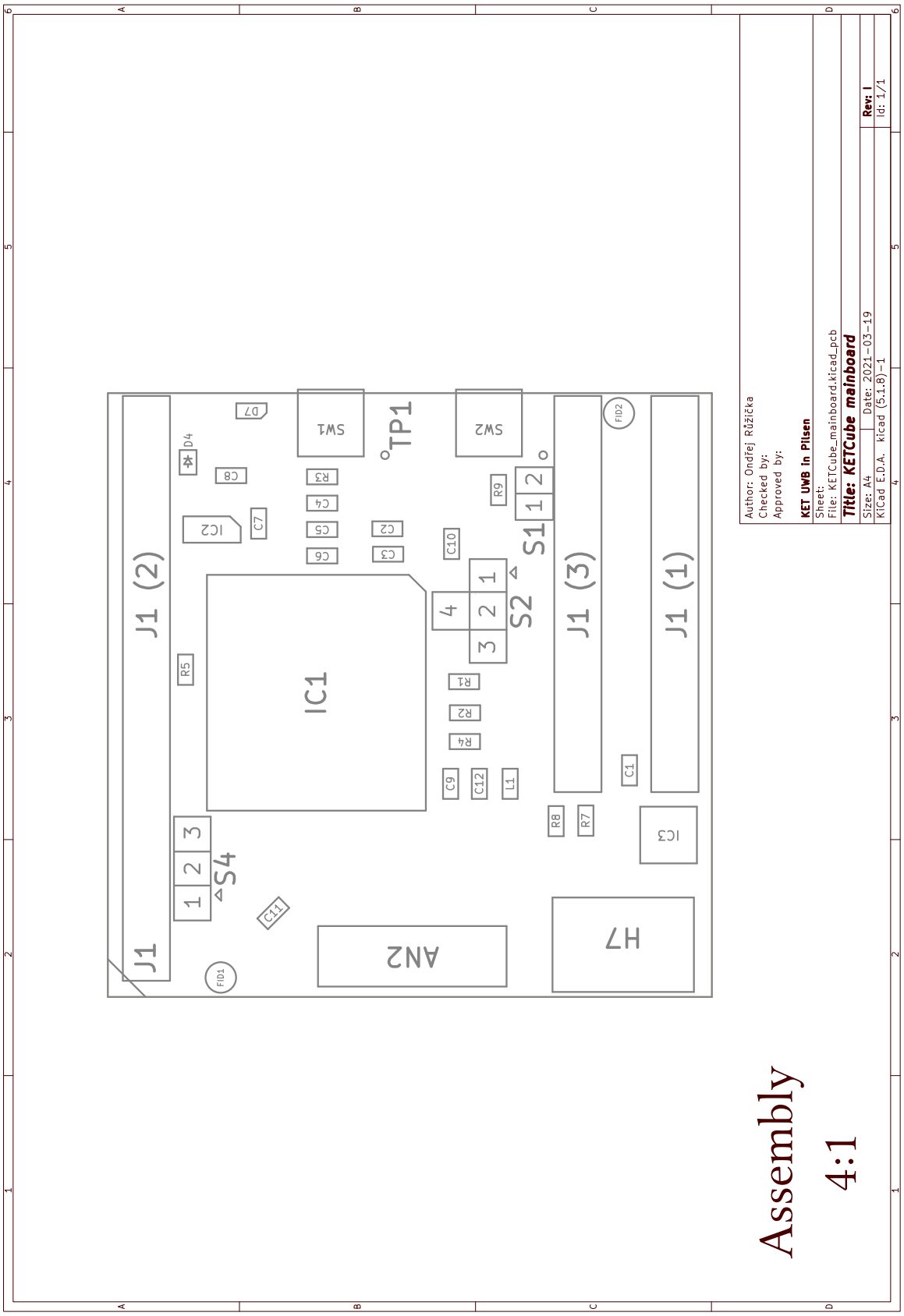
Obr. A.1: Schéma desky KETCube



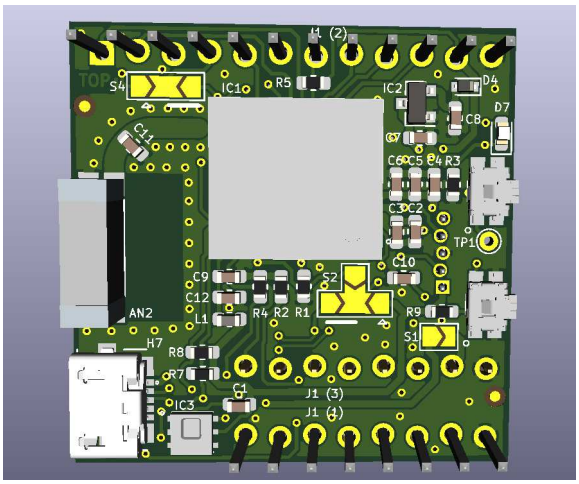
Obr. A.2: Strana TOP KETCube



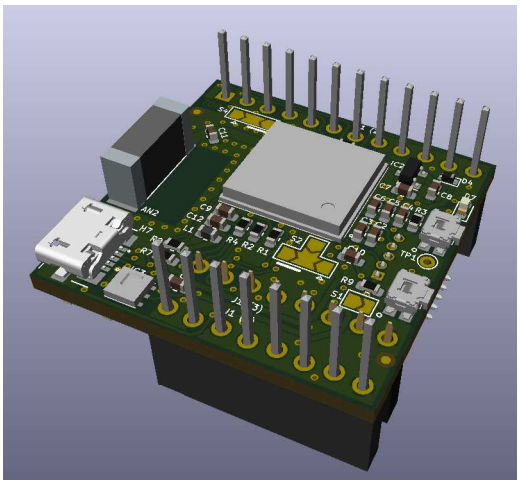
Obr. A.3: Strana BOTTOM KETCube



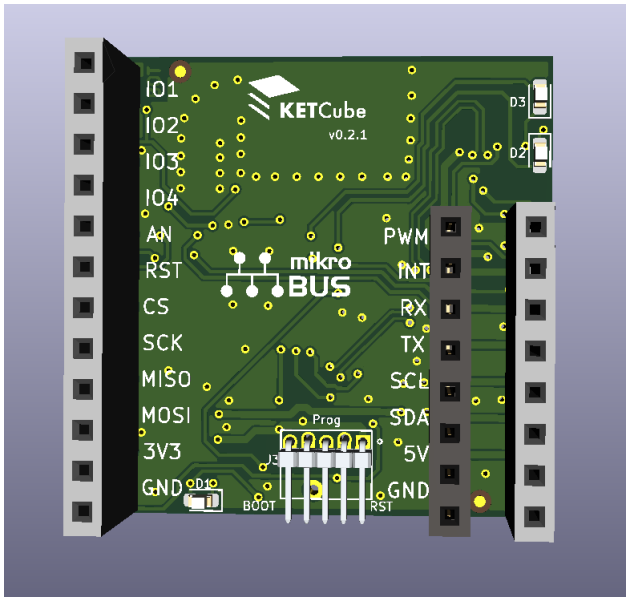
Obr. A.4: Osazovací plán KETCube



(a) Strana součástek - pohled shora



(b) Strana součástek - pohled ISO



(c) Strana spojů - pohled zdola

Obr. A.5: 3D pohledy na navrženou desku KETCube

Příloha B

Soupis pravidel pro návrh součástky

V této příloze jsou uvedeny pravidla SC pro návrh symbolu, footprintu a 3D modelu, v tomto pořadí. Jsou převzata ze souborů *README.md* v příslušných adresářích na GitHubu. Pravidla jsou psána v angličtině, protože je to jazyk celého projektu Smart-CAMPUS ZČU.

KiCAD-lib

SmartCAMPUS KiCAD library **symbol folder**

Symbol design rules should be used as stated in KLC (KiCAD library convention (https://kicad.org/libraries/klc)), except where SC Convention specifies otherwise.

Symbol SC Rules

Design and verification rules according to SC Convention are stated below.

SC_S1 KLC Rules

SC_S1.1 Symbol does not conflict with KLC rules

1. Symbol must be checked by KLC script
    - KLC helper script (checklib.py) must be used by a designer to check the symbol
    - Script output information level should be set to -vv
    - Output from the script must be saved to "drc.log" file and stored accordingly (see SC_S4.1)
    - Acceptable DRC check results:
        - clear (DRC passed) output
        - KLC rules violated due to conflict with SC rules - comment must be added to drc.log
        - Intentionally violated KLC rules are allowed, clear reasoning must be provided in drc.log - verifier must agree

SC_S2 Naming

SC_S2.1 Symbol name

- Symbol has all files named according to KLC

SC_S2.2 Categories

1. SC library is organized into the following categories corresponding to the SC convention:
    - Connector - all kinds of connectors, pins and connect points
    - Discrete - discrete semiconductors (transistor, diode, optocoupler...)
    - ICs_Analog - all analog ICs
    - ICs_Digital - all digital ICs
    - Mechanic - mechanical components, test points, fiducials,...
    - Modules - submodules like processor board
    - Passive - usual passive components (resistor, capacitor, inductor)
    - Power - power components
    - PXI - library for components of a testing system
    - Relays - relays
    - Switches - switch symbols, pushbuttons, coders...
2. KiCAD variables for footprint paths defined
    - To maintain readability, a path for a footprint in the symbol has to be defined with known variables
    - Following variables have to be created in KiCAD
        - SC_CONNECTOR

        - SC_DISCRETE
        - SC_ICS_ANALOG
        - SC_ICS_DIGITAL
        - SC_MECHANIC
        - SC_MODULES
        - SC_PASSIVE
        - SC_POWER
        - SC_PXI
        - SC_RELAYS
        - SC_SWITCHES
    - For 3D model path it is required to create SC_LIB variable pointing to the home folder of the library
    - Example variable definition:

```
SC_LIB          C:\KiCAD-lib
SC_DISCRETE     C:\KiCAD-lib\footprint\Discrete.pretty
SC_ICS_ANALOG   C:\KiCAD-lib\footprint\ICs_Analog.pretty
SC_ICS_DIGITAL  C:\KiCAD-lib\footprint\ICs_Digital.pretty
SC_MECHANIC     C:\KiCAD-lib\footprint\Mechanic.pretty
SC_MODULES      C:\KiCAD-lib\footprint\Modules.pretty
SC_PASSIVE      C:\KiCAD-lib\footprint\Passive.pretty
SC_POWER        C:\KiCAD-lib\footprint\Power.pretty
SC_PXI          C:\KiCAD-lib\footprint\PXI.pretty
SC_RELAYS       C:\KiCAD-lib\footprint\Relays.pretty
SC_SWITCHES     C:\KiCAD-lib\footprint\Switches.pretty
```

3. Design templates can be used to generate and manage path variables
    - For a detailed description on usage of templates see templates (https://github.com/SmartCAMPUSZCU/KiCAD-lib/tree/master/templates)

SC_S3 Library status

SC_S3.1 .status file is created in directory related to symbol

- .status file is stored in directory related to symbol, according to SC_S4.1
- .status file is used to track symbol life

SC_S3.2 .status file is filled correctly according to Component life

- Link to Component life diagram is here (https://github.com/SmartCAMPUSZCU/KiCAD-lib/blob/master/Component%20life.pdf)
- Example: status file for symbol checked by designer:

```
DRC = pass
VRC = void
LibStatus = new
```

SC_S3.3 Requirements on symbol footprint

1. Footprint linked to the symbol
    - footprint has to be linked to the symbol
2. Footprint .status according to requirements
    - To create component on a given level, there must be linked footprint on the same or higher level
    - Symbol can be labeled as "verified" level only when there is linked footprint on "verified" or "validated" level
    - Symbol can be labeled as "validated" level only when there is linked footprint on the "validated" level

SC_S4 Folder structure

SC_S4.1 Symbol folder

- Symbol folder is created in the corresponding category
- Symbol folder is named according to KLC symbol name
- Example: Folder structure for example symbol:

```
symbol
  |
  +- Connector
      |
      +- 2212S-08SG-85
          |
          +- .status
          +- drc.log
          +- 2212S-08SG-85.lib
          +- 2212S-08SG-85.dcm
```

SC_S5 Symbol requirements

SC_S5.1 RICE ID

1. Every symbol has its #rice ID
   - If the part does not exist in the RICE database it will have the temporary prefix "K_" and the first available number:

     ```
     K_CATEGORY_0000001
     ```

2. More specific categories for #rice ID
   - Each category has its more specific short version, which is used as prefix for #rice ID
   - depending on the type of part following prefixes are defined:
     - BAT – Batteries
     - CAB – Cables
     - CAP – Capacitors
     - COL – Coolers
     - CON – Connectors
     - DIO – Diodes
     - EMI – EMI filters (EMC/EMI filters)
     - ICS – Integrated circuits
     - IND – Inductors
     - MCH – Mechanical parts
     - MOD – Modules
     - MSD – Multilayer switching devices (diac, triac, thyristor, ...)
     - OPT – Optoelectronics (LED, optocouplers, phototransistors, ...)
     - OSC – Oscillators (xtals, oscilators, piezoelements, ...)
     - PDV – Protect devices (fuses, ...)
     - PWR – Power devices (power modules)
     - REL – Relays
     - RES – Resistors (resistors, trimmers, potentiometers, ...),
     - SWI – Switches

        - TRA – Transistors
    - Fully defined symbol naming examples:

```
BAT_000124
CAP_001400
K_RES_000012
...
```

SC_S5.2 Additional symbol fields

- Symbol has to have the following fields filled
    - Reference
    - Value ⇒ Manufacturer part number (must be unique!)
    - Footprint ⇒ Specific footprint must be linked to symbol in the corresponding category
    - DValue ⇒ Human readable value
    - #rice ⇒ RICE reference number
    - Published ⇒ Date of publishing symbol
    - Publisher ⇒ Designer of symbol
    - LastRevisionDate
    - LastRevisionNote
    - PackageIndex ⇒ empty string for most cases
    - SubParts (optional) ⇒ when there is a need to use additional parts during assembly (headers typically), this field is used for its #rice number. Then it will be exported to BOM
- Replaces KLC S6.2
- Example:

```
Reference: R
Value: ERJ3EKF1002V
Footprint: SC_Passive:R_0603_1608Metric
DValue: 10k R0603
#rice: RES_000097
Published: 15.11.2020
Publisher: Ondrej Ruzicka
LastRevisionDate: 11.12.2020
LastRevisionNote: footprint fix
PackageIndex: -
```

SC_S5.3 Footprint path definition

- Symbol has to have one default footprint assigned
- Assignment has to be done using category variables
    - SC_variable:footprint
- Example:

```
SC_Passive:R_0603_1608Metric
```

SC_S5.4 Footprint filter definition

- Symbol has to have valid footprint filter with complete path based on SC_S5.3

Last updated 2021-04-05 10:44:20 +0200

KiCAD-lib

SmartCAMPUS KiCAD library **footprint folder**

Footprint design rules should be used as stated in KLC (KiCAD library convention (https://kicad.org/libraries/klc)), except where SC Convention specifies otherwise.

Footprint SC Rules

Design and verification rules according to SC Convention are stated below.

SC_F1 KLC Rules

SC_F1.1 Footprint does not conflict with KLC rules

1. Footprint must be checked by KLC script
   - KLC helper script (check_kicad_mod.py) must be used by a designer to check the footprint
   - Script output information level should be set to -vv
   - Output from the script must be saved to "drc.log" file and stored accordingly (see SC_F4.1)
   - Acceptable DRC check results:
     - clear (DRC passed) output
     - KLC rules violated due to conflict with SC rules - comment must be added to drc.log
     - Intentionally violated KLC rules are allowed, clear reasoning must be provided in drc.log - verifier must agree

SC_F2 Naming

SC_F2.1 Footprint name

- Footprint has all files named according to KLC

SC_F3 Library status

SC_F3.1 .status file is created in directory related to footprint

- .status file is stored in ".data" directory related to footprint, according to SC_F4.1
- .status file is used to track footprint life

SC_F3.2 .status file is filled correctly according to Component life

- Link to Component life diagram is here (https://github.com/SmartCAMPUSZCU/KiCAD-lib/blob/master/Component%20life.pdf)
- Example: status file for footprint checked by designer:

```
DRC = pass
VRC = void
LibStatus = new
```

SC_F3.3 Requirements on footprint 3D model

1. 3D model linked to the footprint
   - 3D model has to be linked to the footprint
2. 3D model .status according to requirements
   - To create component on a given level, there must be a linked 3D model on the same or higher level
   - Footprint can be labeled as "verified" level only when there is a linked 3D model on "verified" or "validated" level
   - Footprint can be labeled as "validated" level only when there is a linked 3D model on the "validated" level

SC_F4 Folder structure

SC_F4.1 Footprint data folder

- Footprint data folder is created with the footprint file in corresponding directory
- Folder is used to store additional information related to the footprint
- Folder must have the same name as footprint, ending with ".data"
- Example: Folder structure for example footprint:

```
footprint
  |
  +- Connector.pretty
      |
      +- exampleConnectorFootprint.kicad_mod
      +- exampleConnectorFootprint.data
          |
          +- .status
          +- license.txt
          +- drc.log
```

4. SC_F5 Layer requiements

SC_F5.1 Fabrication layer requiements

1. Value placed on the F.Fab layer must be set as hidden
    - Fabrication layer is used to export assembly diagrams, therefore only references should be visible
    - Replaces KLC F5.2

Last updated 2021-04-05 10:44:20 +0200

KiCAD-lib

SmartCAMPUS KiCAD library **3D model folder**

3D model design rules should be used as stated in KLC (KiCAD library convention (https://kicad.org/libraries/klc)), except where SC Convention specifies otherwise.

3D model SC Rules

Design and verification rules according to SC Convention are stated below.

SC_M1 KLC Rules

SC_M1.1 Model does not conflict with KLC rules

1. 3D model must be checked against KLC
    - Acceptable DRC check results:
        - clear (DRC passed) output
        - Intentionally violated KLC rules are allowed, clear reasoning must be provided - verifier must agree

SC_M2 Naming

SC_M2.1 Model must have the same name as footprint it is associated with

- Each footprint is directly linked to a 3D model, therefore it must have a similar name

SC_M3 Library status

SC_M3.1 .status file is created in directory with model files

- .status file is stored in directory related to 3D model, according to SC_M4.1
- .status file is used to track 3D model life

SC_M3.2 .status file is filled correctly according to Component life

- Link to Component life diagram is here (https://github.com/SmartCAMPUSZCU/KiCAD-lib/blob/master/Component%20life.pdf)
- Example: status file for footprint checked by designer:

```
DRC = pass
VRC = void
LibStatus = new
```

SC_M4 Folder structure

SC_M4.1 Model is stored in model_name.3dshapes directory

- Each model can have multiple related files, which have to be in the 3D model folder
- Example: typical model.3dshapes folder:

```
.status
sm_model.step
sm_model.wrl
license.txt
```

Last updated 2021-04-05 10:45:12 +0200