UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

Jože Dedič

# ENOVITO RAZVOJNO OKOLJE ZA SOČASNO NAČRTOVANJE STROJNE IN PROGRAMSKE OPREME

DOKTORSKA DISERTACIJA

Mentor: doc. dr. Andrej Trost

Ljubljana, 2006

# *Zahvala*

Zahvaljujem se mentorju doc. dr. Andreju Trostu za izkazano pomoč in usmerjanje pri raziskovalnem delu. Zahvala gre tudi sodelavcem v Laboratoriju za načrtovanje integriranih vezij za vso pomoč in prijetno delovno vzdušje. Posebna zahvala za nesebično pomoč gre dr. Matjažu Fincu.

Zahvaljujem se Ministrstvu za šolstvo, znanost in šport Republike Slovenije za sredstva, ki so mi omogočila raziskovanje in usposabljanje na doktorskem študiju.

# *Povzetek*

V doktorski disertaciji smo se osredotočili na metodologijo načrtovanja sistemov, ki temeljijo na porazdeljenem obdelovanju podatkov. Tovrstno raziskovalno področje uvrščamo v domeno sočasnega načrtovanja strojne in programske opreme. Jedro disertacije predstavlja metodologija visokonivojske obravnave sistemov. Predstavili smo razloge, zakaj je takšna obravnava potrebna, in prednosti, ki jih nudi. V disertaciji je poudarek na konceptih obravnave sistema, ki omogočajo učinkovitejše obvladovanje načrtovalske kompleksnosti sodobnih elektronskih sistemov. Določili smo korake sočasnega načrtovanja strojne in programske opreme ter jih tesno povezali s koncepti obvladovanja kompleksnosti. V okviru naše metodologije smo predlagali razširjeno uporabo abstrakcije, ki predstavlja enega izmed temeljnih konceptov visokonivojske obravnave sistemov, ki pripomorejo k povečanju načrtovalske produktivnosti. Veliko truda smo vložili v izgradnjo podpornega ogrodja, ki omogoča izvajanje načrtovalskega poteka in nudi podporo za koncepte predlagane metodologije. Raziskali smo napredne možnosti modeliranja, ki smo jih uporabili pri implementaciji naše metodologije. Uporabnost postavljene metodologije in predlaganega višjega nivoja abstrakcije smo prikazali na visokonivojskem primeru načrtovanja praktičnega sistema.

Ključne besede:
sočasno načrtovanje strojne in programske opreme, visokonivojska obravnava sistema, abstrakcija, modeliranje, SystemC

# *Abstract*

The focus of this thesis is set on a design methodology of distributed data processing systems. Such research area is classified as a hardware/software co-design methodology. High-level system design methodology represents the core of this thesis. The reasons for high level approach and advantages offered by it are defined. The emphasis is set on the system-level development concepts that provide effective means for handling the design complexity of the state-of-the-art electronic systems. Design flow stages of hardware/software co-design are defined and closely integrated with the concepts, offering effective means for handling the design complexity. As a part of our methodology, we have introduced an extended concept of abstraction, providing grounds for high-level system design that offer a way for increasing the designer's productivity. A substantial amount of effort was put into building a framework offering support for the design flow and the concepts of our methodology. Advanced modelling techniques were studied and used within the implementation of our methodology. A high-level case study of designing a practical system at higher level of abstraction demonstrates the applicability of our methodology.

Key words:
hardware/software co-design, high-level system development, abstraction, modelling, SystemC

# Kazalo

# Seznam slik

# *Seznam tabel*

# 1 Uvod in predstavitev doktorske disertacije

## 1.1 Motivacija

Živimo v dobi nenehnega tehnološkega napredka, ki v naše življenje prinaša vse večjo varnost, udobje in storilnost. Obdaja nas množica elektronskih naprav, ki se vključujejo v posamezne dele našega življenja z enim samim namenom – da ga izboljšajo. Pomagajo nam premagovati razdaljo. Sogovornik je lahko na drugi strani sveta, pa ga s pomočjo naprave, ki se skorajda izgubi v žepu in ki lahko brez priklopa na električno omrežje deluje več dni, slišimo in vidimo povsem jasno. Količina informacij, ki jo hranijo za nas, postaja nepredstavljivo velika. V žepu srajce lahko nosimo količino informacij, ki je ekvivalentna nekaj knjižnim policam, pa še vseeno lahko skorajda hipoma najdemo iskan podatek. Pomagajo nam, da pridemo hitreje na cilj, nas varujejo, nam avtomatizirajo postopke dela ... Zaradi vsega tega smo sposobni narediti veliko več. In ironično – zahtevamo vse več. Od sebe, od ostalih ljudi in od naprav.

Zahteve po naprednih elektronskih napravah predstavljajo gonilno silo tehnološkega razvoja. Istočasno pa množičnost zahtev stroške izdelave posameznega izdelka navkljub tehnološko naprednejšim postopkom vrtoglavo nižajo. Integracija današnjih kompleksnih sistemskih rešitev sloni povsem na napredku tehnologije izdelave integriranih vezij. Močno submikronska tehnologija dandanes omogoča uporabo stotine milijonov tranzistorjev na eni sami silicijevi rezini. To pomeni, da lahko na vsakem integriranem vezju implementiramo praktično neomejeno število funkcij. Te seveda izrabimo za to, da bodo naprave, ki nam služijo, svojo nalogo opravljale še bolje. Ob vsem tem tehnološkem napredku pa ne smemo pozabiti na dejstvo, da je potrebno vso to napredno tehnologijo izdelave integriranih vezij inteligentno povezati v zaključeno celoto, kjer ima še najmanjši tranzistor točno določeno vlogo. Zgodilo se je namreč nekaj, čemur rečemo razkorak v tehnologiji izdelave integriranih vezij in zmožnostih sistemske integracije. Preprosto povedano, si to lahko predstavljamo tako, da je leta nazaj en sam človek lahko naredil razvoj sistema, za implementacijo katerega je bila potrebna velika količina integriranih vezij. Danes je situacija obratna. Integrirana vezja so

tako zelo velika, da je običajno potrebnih veliko človek-let dela, da vse razpoložljive tranzistorje povežejo v skladno delujočo celoto.

Sistemska kompleksnost ne predstavlja edinega problema, s katerim se danes ukvarjajo načrtovalci najsodobnejših elektronskih sistemov. Z manjšanjem dimenzij gradnikov se hitrost delovanja integriranih vezij neprestano dviga. Če je prvo integrirano vezje lahko delovalo s hitrostjo pod 1 MHz (740 kHz, Intel 4004), smo danes v GHz območju vsaj tri dekadne razrede hitrejši. Čeprav je danes poraba moči na en tranzistor mnogo manjša kot pri prvih vezjih, je zaradi kombinacije visoke frekvence delovanja in velikega števila tranzistorjev poraba vezij lahko zelo velika. To pa ni skladno s prej predstavljeno idejo vseprisotnih naprav, ki dopolnjujejo naše življenje. Rabimo naprave, ki bodo delovale, ne da bi jih bilo potrebno imeti neprestano vključene v električno omrežje in za delovanje katerih ne bo treba velikih ter nerodnih hladilnih teles.

Rešitev omenjenih težav predstavlja načrtovanje sistemov, ki bodo sposobni opraviti več pri nižji hitrosti delovanja. *Več* in *pri nižji hitrosti delovanja* pa pomeni, da je potrebno opravila opravljati sočasno. S stališča razpoložljive strojne opreme oz. števila tranzistorjev to ne predstavlja nikakršnega problema. Problem nastane pri sistemski integraciji, ko želimo narediti *več pri nižji hitrosti*. Povedali smo, da je integracija sodobnih elektronskih sistemov že sama po sebi izredno kompleksna naloga, če pa temu dodamo še dodatno zahtevo čim večje stopnje sočasne obdelave, pa kompleksnost povečamo z novo prostorsko stopnjo. Če smo na začetku povedali, da množičnost zahtev integriranih elektronskih rešitev vrtoglavo niža produkcijsko ceno končnega izdelka, lahko sedaj dodamo, da prihajamo v obdobje, ko intelektualni dodatek, ki omogoča inteligentno izrabo tehnologije, predstavlja večinski del cene. Pridemo do kompromisov funkcionalnosti, zmogljivosti in cene.

Naprave naj bodo hitre, z majhno porabo energije, z ne prevelikimi fizičnimi dimenzijami, z visoko stopnjo prilagodljivosti, naj bodo poceni ... Načrtovanje sodobnih elektronskih sistemov je vse prej kot preprosta rutinska naloga.

## 1.2 Cilji

Povedali smo, da želimo narediti več pri nižji hitrosti, istočasno pa moramo skrbeti še za množico kompromisov. To pomeni, da moramo k načrtovanju sistema pristopiti celovito. Celovito načrtovanje predstavlja več kot samo prisiljeno sodelovanje sicer ločenih področij. Je tudi več kot določanje in spoštovanje vmesnih točk med temi področji. Zahteva se popolno

razumevanje načrtovalskih tehnik vseh vpletenih strani. K obvladovanju kompleksnosti načrtovanja elektronskih sistemov bomo lahko prispevali, če bomo povsem seznanjeni s celotno sliko problematike. Ker predstavlja takšen pristop še razmeroma mlado smer raziskovanja, se bomo v pričujoči doktorski disertaciji osredotočili na postavitev temeljev celovitega pristopa načrtovanja sodobnih sistemov.

Da bi lahko raziskovali v smeri produktivnejšega načrtovanja sistemov, moramo najprej videti pasti klasičnega načrtovanja. Ugotoviti moramo, katere odločitve so bolj pomembne in se jim je zato potrebno podrobneje posvetiti.

Ugotoviti moramo, kateri so tisti koncepti, ki pripomorejo k obvladovanju kompleksnosti sistemov, in na kakšen način jih lahko uporabimo, da povečamo produktivnost načrtovanja. Želimo postaviti ogrodje načrtovalskega poteka, ki bo določilo korake, potrebne za učinkovito izpolnitev načrtovalskih zahtev sistema. Ker je optimalno oz. vsaj dovolj dobro izpolnjevanje načrtovalskih zahtev tesno povezano z ovrednotenjem posameznih načrtovalskih odločitev, je naslednja logična zahteva avtomatizacija postopka ovrednotenja načrtovalskih odločitev. Raziskali bomo, kateri so tisti mehanizmi, ki to omogočajo, in predstavili predlog rešitve.

Gledano globalno, ciljamo na ogrodje, ki bo natančno definiralo postopke celovitega načrtovanja kompleksnih elektronskih sistemov ter kot tako postavilo temelje za nadaljnjo uporabo ožjih in podrobneje usmerjenih rešitev. Zanima nas celovita krovna rešitev, ki bo omogočala izpeljavo celotnega načrtovalskega postopka in skladno povezala ožje usmerjene algoritme načrtovanja kompleksnih sistemov.

## *1.3 Organizacija disertacije*

V drugem poglavju bomo poglobljeno predstavili razloge za razkorak v tehnoloških in načrtovalskih zmožnostih. Bralcu bomo razložili, zakaj je sedanja situacija takšna in zakaj je potreben nov pristop k načrtovanju sodobnih sistemov. Nakazali bomo možne rešitve.

V tretjem poglavju se bomo osredotočili na pregled sorodnega dela. Predstavili bomo vse raziskovalne skupine, ki so tako ali drugače pomembneje vplivale na potek našega raziskovanja. V glavnem so to večje raziskovalne skupine z že več letno tradicijo in ustaljenimi metodologijami. Predstavili bomo, v kateri del celotnega načrtovanja so se v svojem raziskovanju osredotočili.

Četrto poglavje predstavlja jedro naših raziskovalnih dosežkov in se osredotoča na določitev najpomembnejših konceptov sočasnega načrtovanja strojne ter programske opreme. Znotraj tega poglavja je predstavljen tudi naš izvirni prispevek k znanosti, tj. načrtovanje na višjem nivoju abstrakcije. Pokazali bomo, da prepletena uporaba različnih konceptov načrtovanja pripelje do zelo kompleksnih metodologij, zato je razumevanje konceptov ključnega pomena za razumevanje namenov posamezne metodologije.

V petem poglavju bomo prikazali koncepte modeliranja in povezavo na koncepte sočasnega načrtovanja strojne ter programske opreme. Razložili bomo, zakaj gradnja lastnega modelirnega okolja ni dobra ideja, in pokazali, kaj pomeni generično modeliranje.

Šesto poglavje bo doktorsko disertacijo zaokrožilo s prikazom metodologije sočasnega načrtovanja strojne in programske opreme. Prikazali bomo način uporabe pomembnih konceptov načrtovanja in se še posebej osredotočili na prikaz prednosti visokonivojskega načrtovanja sistema.

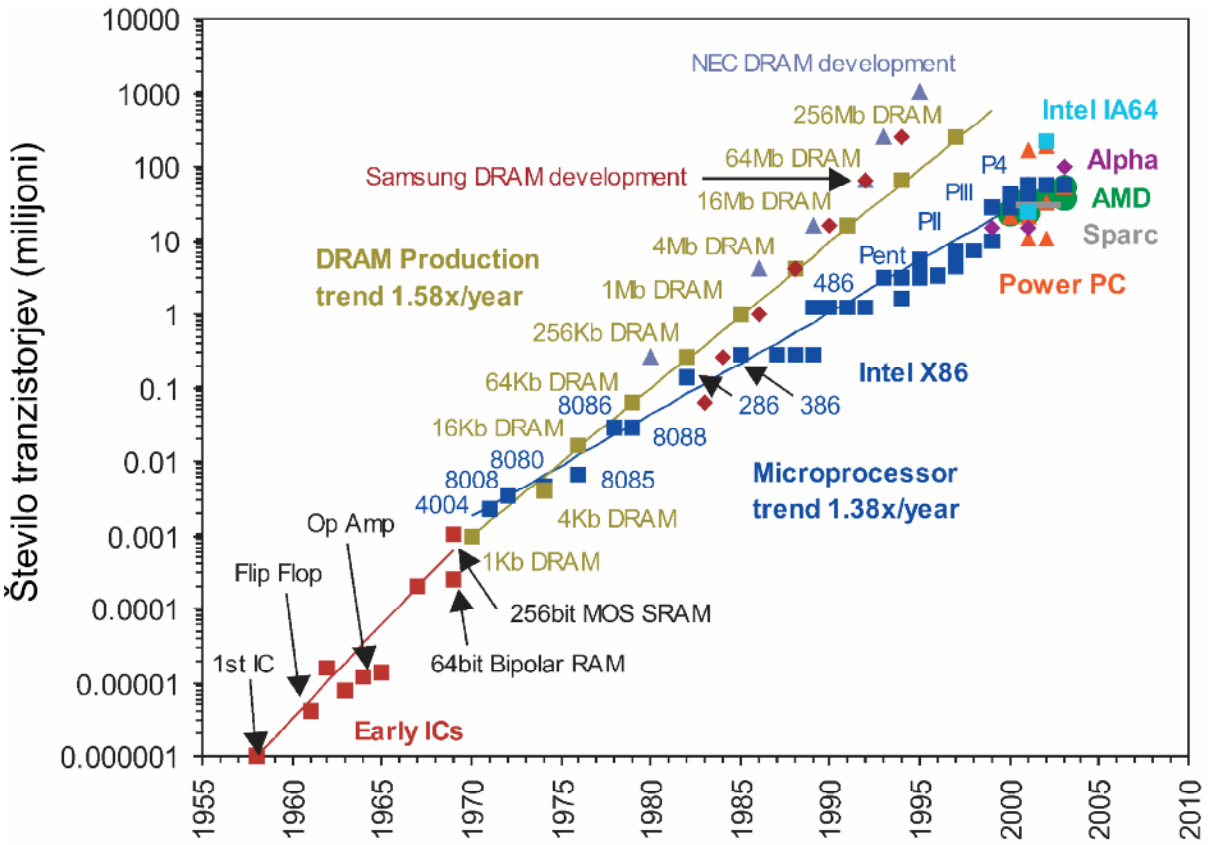
# 2 Razkorak v tehnoloških in načrtovalskih zmožnostih

Zaradi napredka v tehnologiji izdelave integriranih vezij smo zadnjih nekaj desetletij priča silnemu razmahu izdelkov, katerih delovanje bodisi temelji na vgrajenih elektronskih komponentah bodisi je njihova učinkovitost zaradi dodatnih elektronskih komponent močno povečana. Napredek v tehnologiji izdelave in povečanje povpraševanja po integriranih vezij sta omogočila premik ciljnih izdelkov stran od prvotnih, izredno dragih, ki so bili namenjeni ali vojski ali raziskovanju vesolja[1] v področje masovne proizvodnje. Ravno masovna proizvodnja je bila tista, ki je povzročila tako obsežno zanimanje za področje obvladovanja izdelave integriranih vezij, zato je tako računalniška kot tudi elektrotehniška znanost osvojila že zelo bogat repertoar razumevanja in znanja, ki se kaže v številnih obstoječih metodologijah, pristopih in orodjih, ki jih imamo na voljo.

Preden se lotimo neposrednega jedra doktorske naloge, se nam zdi pomembno, da preostanek tega poglavja namenimo kratki predstavitvi razvoja, ki nas je pripeljal do trenutnih metod načrtovanja, kjer smo naleteli na težavo, s katero se aktivno ukvarja tako raziskovalna sfera kakor tudi industrija. Prišlo je namreč do razkoraka med tem, kaj tehnologija izdelave integriranih vezij nudi, in tem, kaj od tega trenutni načrtovalski postopki omogočajo *učinkovito* izrabiti. Najpogosteje uporabljeni postopki načrtovanja so namreč osnovani na metodah, ki ne nudijo učinkovitih možnosti ovrednotenja pomembnih sistemskih odločitev, ravno te pa imajo velik vpliv na celovito ovrednoteno učinkovitost sistema. Dovolj velika hitrost obdelave podatkov ni več edini parameter, h kateremu stremimo. Pri načrtovanju sistema imamo tako na eni strani uveljavljene rešitve, ki temeljijo na ponovni uporabi obstoječih delov sistema in na do sedaj vloženemu trudu v razvoj pripadajočih rešitev –

[1] Prva integrirana vezja, namenjena vojski ali raziskovanju vesolja, so imela vlogo dopolnjevanja funkcionalnosti naprav, v katere so bila vgrajena, zato so ta vezja istočasno tudi prvi primeri vgrajenih sistemov; tj. računalnika, ki je vgrajen v napravo in je namenjen opravljanju točno določene funkcije. Čeprav je bilo število tranzistorjev prvih integriranih vezja majhno, t. i. vezja SSI (Small-Scale Integration – nizka stopnja integracije) so vsebovala nekaj deset tranzistorjev, je bila njihova vloga ključnega pomena pri prvih letalskih in vesoljskih projektih ter obratno. Programa Minuteman missile in Apollo sta potrebovala izredno lahek digitalni računalnik za interno voden računalnik poleta. Računalnik za vodenje Apolla je vodil in motiviral tehnologijo integriranih vezij, medtem ko jo je projekt Minuteman missile prisilil v masovno proizvodnjo [wp1].

takšno načrtovanje je hitro in poceni. Na drugi strani imamo namenske in po meri prilagojene rešitve, ki načrtovalske zahteve zajamejo celostno – torej učinkovite rešitve. Kljub svoji kompleksnosti bomo prve rešitve označili za "enostavne"; pomembno je namreč razumeti, da je večina kompleksnosti skrita v spodaj ležečih orodjih, ki s svojo učinkovitostjo vso kompleksnost skrijejo in naredijo načrtovanje – relativno gledano – enostavno. Pri teh rešitvah je "učinkovitost" žrtvovana za "enostavnost" pridobljene rešitve; žrtvovana je za čas za nastop na trg, ekonomičnost prodaje, končno ceno in za možnost lažjega obvladanja implementacijske kompleksnosti. Podprta je tudi abstrakcija, kar načrtovalcem omogoča, da lahko razmišljajo v domeni aplikacije in ne implementacije. Pomembno je dejstvo, da je razlika v ceni silicija (tj. izdelave, ne pa načrtovanja integriranih vezij) obeh rešitev zanemarljiva ali če se izrazimo še malce drzneje, da je cena silicija obeh rešitev zanemarljiva. Poglavitno vrednost izdelka določa izbira med "učinkovito" ali "enostavno" rešitvijo. Rešitev sama je tista, ki določa dodano vrednost.




Slika 1 – Število tranzistorjev v integriranih vezjih [a3]

Tako kompleksnost aplikacij kot tudi njihova statična sistemska sredstva vztrajno naraščajo (število tranzistorjev, Slika 1) – nadaljnjo učinkovito obvladovanje sodobnih

sistemov bo mogoče le z avtomatiziranimi načrtovalskimi orodji, ki bodo povečala produktivnost načrtovalcev. Avtomatizacijska industrija načrtovanja elektronskih sistemov sicer nudi trajno 23% letno povečanje produktivnosti načrtovalcev [a5], po drugi strani pa se kompleksnost spodaj ležeče tehnologije izdelave integriranih vezij vztrajno povečuje 60% letno (Moorov zakon). Načrtovalski razkorak predstavlja razliko teh dveh številk. Rešitve, ki bi pomagala premostiti razkorak v tehnoloških in načrtovalskih zmožnostih na kar se da učinkovit način, še nimamo na dlani, zato je potreben širši vpogled v dano problematiko, da je omogočen celovit pristop k raziskovanju problema. Bralcu bo lažje razumljivo sedanje stanje in kompleksnost problema, s katerim se ukvarja področje sočasnega načrtovanja strojne in programske opreme (SNSPO), če mu predstavimo krajši opis poteka razvoja integriranih vezij in arhitekturnih rešitev, ki so bile pogojene s postopno povečujočim se obsegom sistemskih sredstev.

## *2.1 Vseprisotnost, poraba energije in naraščajoča kompleksnost aplikacij*

Sodobne elektronske naprave temeljijo na kompleksni elektronski zasnovi in visoki stopnji sistemske integracije. Splošen trend povečevanja funkcionalnosti, zmogljivosti in integracije večpredstavnostnih, komunikacijskih, nadzornih in ostalih naprav za obdelavo podatkov je pripeljal do izdelkov, ki so čedalje manjši, nudijo čedalje več v smislu funkcionalnosti, zmogljivosti in mobilnosti, se znajo povezati v okolje in so do neke stopnje skladni s prihodnjimi standardi. Izredno pomemben podsklop teh izdelkov tvorijo tako imenovani vgrajeni sistemi (VS). Kakor že beseda sama namiguje, so to elektronski sistemi, ki so vgrajeni v določeno napravo z namenom opravljanja/dopolnjevanja funkcij nadzora, krmiljenja in obdelave podatkov. Ti sistemi se od splošnonamenskih sistemov ločijo v tem, da imajo v sistemu, v katerem so vgrajeni, točno določeno vlogo, ki je ni mogoče spreminjati.

Načrtovanje sodobnih elektronskih sistemov vodijo kompromisi. Če pustimo ob strani vedno prisotne ekonomske dejavnike, kot so cena elektronskega sistema (celotne naprave ali pa samo dodane vrednost v primeru VS) in njegov čas do trga, so kompromisi, ki usmerjajo načrtovanje teh sistemov, naslednji: zmogljivost, poraba energije, površina integriranega vezja, odzivnost realnega časa itd [k1].

Poraba energije igra zelo pomembno vlogo pri načrtovanju elektronskega sistema. Število tranzistorjev v integriranih vezij se, skladno z Moorovim zakonom, podvoji vsakih 18 mesecev, medtem ko se na drugi strani kapaciteta baterij podvoji vsakih 5 let [a1]. Priročne in

praktično mobilne naprave bomo imeli le, če nam bo uspelo omejevati moč integriranih vezij, ki te naprave poganjajo.

Razvoj elektronskih sistemov postopoma vodi do elektronskih naprav, ki bodo svojo vlogo v našem vsakdanjem življenju igrale nevsiljivo in stopljeno z okoljem; uporabnik se bo tako lahko osredotočil na nalogo samo, ne pa na napravo [p21], [k10]. Vseprisotnost[2] VS je pojem, ki ga je vpeljal M. Weiser [p22], in predstavlja tretji val računalniških sistemov. Prvi val so predstavljali centralni računalniki, ki si jih je delilo veliko uporabnikov. Sedaj smo v dobi osebnih računalnikov, kjer vsak uporabnik uporablja (najmanj) en računalnik. Naslednji val, val vseprisotnih računalnikov, napoveduje porast števila računalnikov in njihov umik v ozadje naših življenj. To bodo elektronski VS, ki bodo vgrajeni v okolje in bodo vseprisotni. Vseprisotni VS bodo smiselni le, če bodo svojo vlogo opravljali učinkovito, če bodo prilagodljivi, če ne bodo preveliki, da bi se nevsiljivo stopili v okolje, in če bodo kar se da avtonomni. S porastom vseprisotnosti VS se je zato pojavila zahteva po tipalih, aktuatorjih in celo kompleksnejših sistemih, ki bi se ali povsem napajali z energijo iz okolja ali pa z vgrajeno baterijo delovali celotno življenjsko dobo. Kot lahko vidimo, igra poraba električne energije pri uspešni integraciji VS v okolje zelo pomembno vlogo, saj ob zahtevani funkcionalnosti vpliva tako na avtonomnost kot tudi fizično velikost sistema.

Po drugi strani pa celo sistemi, katerim mobilnost in s tem povezana nizka poraba nista na prvem mestu, občutijo, kako pomembna je vloga porabljene energije. Trend trajno povečujoče se funkcionalnosti, zmogljivosti in integracije je pripeljal do integriranih vezij z nekaj sto milijoni tranzistorjev, pri katerih je disipacija toplote v razredu 100 W ali več[3]. Tako velike disipacije toplote zahtevajo posebna draga ohišja integriranih vezij in posebej načrtovana hladilna okolja. To nas privede do povsem novih problemov, ki jih je treba nasloviti, če želimo v prihodnosti imeti izdelke, ki jih bo (še) možno narediti. Vztrajno povečevanje moči pa ni problem samo zaradi dragih ohišij in kompliciranih hladilnih sistemov, ampak vpliva tudi na zanesljivost. Minimalne dimenzije integriranih vezij se krčijo, kar po eni strani omogoča večje frekvence delovanja, po drugi strani pa to pomeni večje tranzientne tokove preklopov, kar znižuje življenjski čas vezja in istočasno povzroča dinamične napetostne padce, ki vnesejo dodatne funkcionalne probleme. Zaradi krčenja minimalnih dimenzij in

[2] V angleščini označeno s pojmoma: *pervasive computing* in *ubiquitous computing*.
[3] Intel Itanium 2 (jedro Madison), večprocesorsko jedro za strežnike, ima 130 W disipacije toplote in 592 milijonov tranzistorjev [ss8].

večanja števila tranzistorjev igra statični del izgub čedalje pomembnejšo vlogo [p23]. Enačba (1) prikazuje celotne izgube integriranega vezja, razdeljene v dinamičen del zaradi preklopov in statičen del, ki je posledica puščanja tokov (tok tranzistorja pri napetosti, nižji od napetosti praga, in tok spoja vrata-oksid). Iz enačbe (2) je razvidna eksponentna odvisnost toka prevajanja tranzistorja od temperature integriranega vezja. Iz tega je razvidno, da moramo za omejevanje statičnih izgub vezja omejiti njegovo delovno temperaturo. Še zlasti je to pomembno pri vezjih, kjer dinamični del izgub ni tako očitno prevladujoč, tj. pri vezjih, ki ciljajo področje vseprisotnih VS.

$$P = ACU^2 f + \left(I_{podprag} + I_{oks}\right)U \tag{1}$$

$$I_{podprag} = K_1 W e^{-\frac{U_{prag}}{nU_T}}\left(1 - e^{\frac{U}{U_T}}\right);\ U_T = {}^{kT}\!/_{q} \tag{2}$$

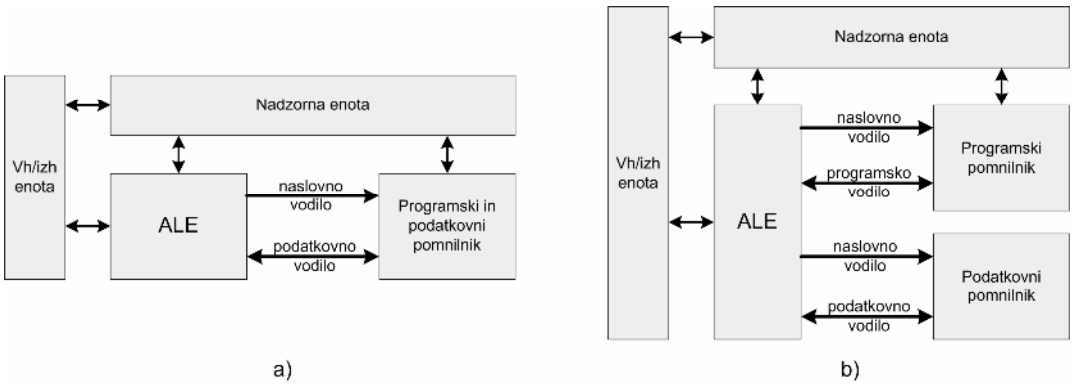
$$I_{oks} = K_2 W \left(\frac{U}{d_{oks}}\right)^2 e^{-\frac{\alpha d_{oks}}{U}} \tag{3}$$

Aplikacije, ki jih sodobni elektronski sistemi podpirajo, zahtevajo čedalje večjo pasovno širino obdelave podatkov, število in kompleksnost nadzornih funkcij, ki jih opravljajo, se veča, hkrati pa vklop v okolje zahteva tudi odzivnost realnega časa. Te zahteve se izražajo v povečanem številu operacij, ki jih mora elektronski sistem zagotoviti v časovni enoti. V primeru, da je za obdelovanje podatkov uporabljen procesor, ki podatke v osnovi obdeluje sekvenčno, je povečanje števila operacij mogoče doseči s povečanjem frekvence urinega signala. Ta pristop je tudi najlažji, saj je omogočen z napredkom v izdelavi integriranih vezij in nenehnim krčenjem minimalnih dimenzij. Kot je bilo že omenjeno, smo leta 2006 v obdobju mikroprocesorjev z nekaj 100 milijoni tranzistorjev, ki tečejo s frekvenco nekaj GHz in porabijo nekaj deset (ali več) vatov. Predvsem slednjih je veliko preveč za kakršno koli vseprisotno prihodnost VS. Čedalje večje količine obdelanih podatkov, ki se od sodobnih sistemov zahtevajo vse pogosteje, bo potrebno zagotoviti z upoštevanjem celovite učinkovitosti sistema. To pomeni, da bo potrebno korenito spremeniti sedanji pristop k višanju količine obdelanih podatkov na časovno enoto – višanje frekvence delovanja mikroprocesorjev trenutno vodi samo k večji porabi.

## 2.2 Centralno obdelovanje podatkov

### 2.2.1 Von Neumannova arhitektura

Prva integrirana vezja so vsebovala majhno število tranzistorjev, ker je bilo samo tako možno zagotoviti zanesljivo delovanje celotnega vezja pri sprejemljivi ceni. Leta 1971 je Intel izdelal prvi mikroprocesor *Intel 4004*, ki je imel vsega 2300 tranzistorjev, naslednji, zelo uspešen, je bil Intel 8080 (1974), ki je imel 4500 tranzistorjev. Zaradi omejenih sredstev strojne opreme (omejeno število tranzistorjev) je bilo potrebno poiskati način, kako postaviti arhitekturo, ki bi omogočila opravljanje zahtevnejših opravil in ki bi bila še programabilna. Rešitev za to je John von Neumann [wp2] predstavil že leta 1945. Osnovni elementi arhitekture, ki jo je predstavil, so spomin, nadzorna enota, aritmetično-logična enota (ALE) in V/I enota (slika 2a). ALE je sposobna opravljanja preprostejših aritmetičnih in logičnih operacij, zato je potrebno vsa zahtevnejša opravila predstaviti v obliki osnovnih ukazov, ki se zaporedno izvedejo nad podatki, ki so drugače shranjeni v spominu in so v ALE preneseni samo za obdelavo. Zaporedje opravil je, prav tako kot podatki, nad katerimi se izvajajo določene operacije, shranjeno v spominu, nadzorna enota pa skrbi za pravilno koordinacijo posameznih enot, tako da je zaporedje izvajanja pravilno; tako prenosa kot tudi obdelave podatkov.




Slika 2 – Shematski prikaz von Neumannove in harvardske arhitekture

Vse odtlej smo, skladno z Moorovim zakonom, priča velikemu razvoju na področju integriranih vezij in posledično tudi mikroprocesorjev, zato so danes sredstva strojne opreme na voljo praktično v neomejenem obsegu, skoraj vsi[4] mikroprocesorji pa še vedno temeljijo na

[4] Na tem mestu je mišljen samo osnovni princip zaporednosti obdelovanja. Obstajajo tudi mehanizmi pohitritve, ki še vedno temeljijo na von Neumannovi arhitekturi (cevovodne strukture, koprocesorji ...).

isti Von Neumannovi[5] arhitekturi, ki je omogočila programabilnost in s tem večnamenskost integriranih vezij z omejenimi sredstvi.

Računalniška veda – način razmišljanja, orodja, rešitve in pristopi – temelji na principu delovanja von Neumannove arhitekture [p6], [p17] . Skorajda vsi obstoječi programski jeziki so namenjeni opisu zaporedja osnovnih ukazov, ki, ustrezno združeni, tvorijo kompleksne rešitve. Način razmišljanja programskih inženirjev je sekvenčen. Pri tem fundamentalno sekvenčnem modelu je kakršna koli vzporedna obdelava samo navidezna, pa še ta največkrat vpelje povsem nove težave (semaforji, muteksi ...), če za navidezno vzporedno delovanje ni elegantno poskrbljeno na nivoju OS [p6].

### *2.2.2 Pretočno ozko grlo, interaktivni in reaktivni sistemi*

Dejstvo, da sta CPE in spomin ločena, pripelje do t. i. von Neumannovega ozkega grla[6], ker je pasovna širina prenosa podatkov med CPE in spominom zelo majhna v primerjavi s količino spomina ali pa celo hitrostjo obdelave podatkov CPE. To ozko grlo pretočnosti podatkov igra pri sodobnih sistemih čedalje pomembnejšo vlogo [p17]. Citat iz Backusovega predavanja [p20]:

*"Nedvomno mora obstajati manj primitiven način opravljanja obsežnih sprememb nad shranjenimi podatki, kot je porivanje ogromnega števila besed naprej in nazaj skozi von Neumannovo ozko grlo. Ne samo, da je ta kanal dobesedno ozko grlo prometa podatkov naloge, ampak je, kar je še pomembneje, intelektualno ozko grlo, ki nas je držalo privezane na načinu razmišljanja ena-beseda-naenkrat, namesto da bi nas opogumljala, da bi razmišljali o večjih konceptualnih enotah obravnavane naloge. V tem pogledu je programiranje samo načrtovanje in določanje ogromnega prometa besed skozi von Neumannovo ozko grlo. Velik del tega prometa zadeva, ne pomembnih podatkov samih, temveč to, kje jih najti."*

Delna rešitev za omenjen problem je uporaba izpopolnjenih predpomnilniških shem, ki na podlagi statističnih podatkov o dostopu do spomina pripomorejo k temu, da mikroprocesor, statistično gledano, ni več omejen s pasovno širino prenosa podatkov. Veliko pridobitev v zmogljivosti, ki izvirajo iz arhitekturnih sprememb, temelji ravno na statističnih pohitritvah –

[5] Pogosto uporabljena je tudi harvardska arhitektura (slika 2b), ki je v osnovi nadgradnja von Neumannove arhitekture. Osnovna izboljšava je ta, da so podatkovna in ukazna spomina ter pripadajoči vodili ločeni.
[6] V izvirniku *Von Neumann bottleneck*. Izraz, ki ga je uporabil John Backus v predavanju ob podelitvi ACM Turingove nagrade. Predavanje je objavljeno v [p20].

poleg že omenjene uporabe predpomnilnika so te npr. še: napovedano zaporedje izvajanja instrukcij, dinamično razpošiljanje podatkov in napoved vejitev [p6]. Pohitritve, osnovane na statističnih podatkih o obdelavi – kakor že beseda samo pove – delujejo statistično. To pomeni, da gledano v povprečju, k hitrosti obdelovanja sicer pripomorejo, cena, ki jo je potrebno zaradi teh pohitritev plačati, pa je nedoločljiv čas izvajanja posameznega opravila (npr. če je vsebina predpomnilnika neustrezna – *cache miss*). Iz tega sledi, da so vse te statistične arhitekturne pohitritve za VS z zahtevami sprotne obdelave podatkov in še zlasti življenjsko kritične aplikacije neuporabne. Če so naš cilj avtonomni VS, je rešitev povečati frekvenco delovanja tako, da bodo vsa opravila zaključena tudi v najslabšem možnem primeru in navkljub nepredvidljivosti neustrezna.

Glede na način obravnave podatkov ločimo interaktivne, transformacijske in reaktivne sisteme [k8], [p6]. Za interaktivne sisteme je značilen dialog z uporabnikom, ki vnese določene zahteve in/ali podatke, ki jih sistem *s svojo hitrostjo* obdela in rezultate vrne uporabniku. Transformacijski sistemi preprosto vzamejo blok vhodnih in jih, zopet *s svojo hitrostjo*, pretovorijo v blok izhodnih podatkov, ki predstavljajo že kar izhod sistema. Reaktivni sistemi pa se od predhodnih dveh ločijo po tem, da na okolje reagirajo nenehno, *s hitrostjo okolja*. Ti sistemi imajo omejitve realnega časa in pri njih je varnost pogosto ključnega pomena; celo do točke, kjer napaka lahko vpliva na izgubo življenj. V nasprotju s transformacijskimi in reaktivnimi sistemi se njihovo izvajanje tipično ne zaključi (z izjemo napake). Čeprav so sklopi lahko zgrajeni deterministično, je delovanje reaktivnih sistemov inherentno nedeterministično, saj so pogojeni s trenutnimi vhodi iz okolja; veljavnimi ali ne, od sistema se pričakuje, da jih bo obdelal.

V zvezi z reaktivnimi sistemi smo omenili zahteve realnega *časa*, v teoriji računalniške znanosti pa je bil čas sistematično odstranjen; nadomeščen je bil z diskretnim zaporedjem izvajanja ukazov. Z uporabo programskih jezikov lahko določimo zaporedje izvajanja operacij, ne moremo pa eksplicitno določiti časa. Čas je nepredvidljiv in v najboljšem primeru določljiv z območjem najkrajšega/najdaljšega časa izvajanja opravila. V operacijskih sistemih s podporo realnega čas je pogosto, da je opis komponente sisteme (procesa) skrčen zgolj na eno samo številko, njeno prioriteto. Prioriteto lahko ponovno razumemo kot statističen podatek o pomembnosti izvajanja procesa. Problemi so tudi z vzporednim delovanjem, bodisi navideznim ali resničnim (v primeru večprocesorskih sistemov), saj so obstoječi podporni mehanizmi preveč preprosti (semaforji, monitorji in muteksi) [p6].

Poudarili bi radi, da z izpostavitvijo vseh problemov sedanje računalniške znanosti le-te in vseh njenih enormnih dosežkov nikakor ne želimo razvrednotiti, radi pa bi prikazali, kako globoko je zakoreninjen von Neumannov sekvenčni pristop reševanja problemov. Z vztrajanjem na tej osnovi bomo lahko deležni samo izboljšav, ki bodo takšne ali drugačne kompenzacije določenih pomanjkljivosti omenjene arhitekture in/ali načina obdelovanja podatkov [p6], [p17]. Uporaba določenih ožje namenjenih procesorjev (koprocesorji, DSP-ji, grafični procesorji ...) še vedno temelji na sekvenčnem obdelovanju podatkov, le da sta v tem primeru procesorjeva ALE in spremljevalna arhitektura prilagojeni določenim tipom operacij. Na sekvenčnem obdelovanju so osnovani tudi procesorji, ki podpirajo različne oblike vzporedne obdelave podatkov: vektorski procesorji (ena instrukcija, več podatkov), superskalarni procesorji (SO tekom delovanja določa, kaj se bo izvajalo vzporedno) ter procesorji VLIW (ena instrukcija za več ALE, PO določi vnaprej, kaj se bo izvajalo vzporedno) [k9].

## 2.3 Porazdeljeno obdelovanje podatkov

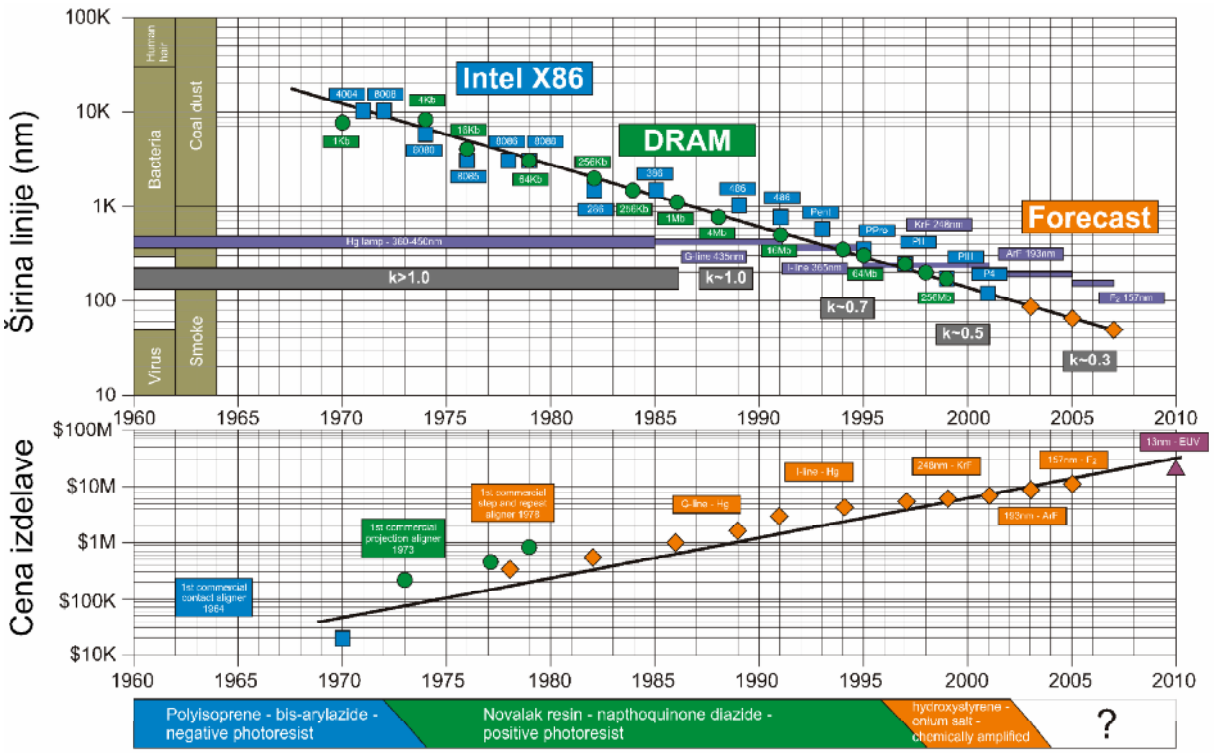
### 2.3.1 Namenska vezja

Do sedaj je bilo govora le o programirljivih vezjih, ki podatke obdelujejo centralno in zatorej sekvenčno, tj. mikroprocesorjih. Pri njih vrstni red operacij določa programska koda, ki je zapisana v pomnilnik in je lahko prirejena neskončni paleti aplikacij. V luči opisovanja problema bi lahko rekli, da so namenska vezja (ASIC) mikroprocesorjem komplementarna. To so vezja, kjer je potek obdelovanja podatkov prirejen določenemu sklopu aplikacij in se izvaja na SO, razviti s točno določenim namenom. Sredstva namenskih vezij so tako po meri prilagojena ciljni aplikaciji. Nivo vzporednega obdelovanja podatkov je poljuben, hitrost obdelave posameznih podsklopov pa je tesno prilagojena zahtevam posameznega podsklopa. Skupen rezultat vsega je, da je veliko lažje poskrbeti za odzivnost realnega časa, hkrati je tudi poraba teh vezij lahko nekaj dekad nižja od splošno namenskih programirljivih vezij.

Problem pa nastopi v ceni izdelka, saj je načrtovanje sodobnih sistemov izredno kompleksen postopek, cene priprave za proizvodnjo pa se vrtoglavo dvigajo[7]. Iz tega sledita dve pomembni dejstvi: a) vsaka napaka v načrtovanju je izredno draga, zato je potrebno veliko testiranja, preden se design preda v izdelavo v tovarno integriranih vezij in b) zaradi velikih začetnih stroškov, povezanih s pripravo za masovno izdelavo, so cenovno smotrna

[7] Izdelava seta mask za 90 nm tehnologijo je v rangu milijona dolarjev (leto 2005), [a2].

samo vezja, ki se izdelujejo v velikih količinah (Slika 3). Slednje tudi razloži popularnost/uspeh prej omenjenih mikroprocesorjev.




Slika 3 – Trend krčenja dimenzij in večanja cen izdelave integriranih vezij [a4]

### *2.3.2 Vezja FPGA*

Poleg omenjenih namenskih vezij omogočajo porazdeljeno obdelovanje podatkov tudi vezja FPGA. V osnovi so to vezja s programabilno mrežo logičnih celic (pomnilni elementi in logična vrata), ki jih lahko med seboj povežemo v poljubno kompleksne strukture [k11]. Samo za ilustracijo naj omenimo, da so lahko sodobna vezja FPGA tako zelo velika, da lahko vsebujejo celo nekaj zelo zmogljivih procesorjev, ki predstavljajo samo majhen del celotne površine integriranega vezja. Primer takšnega vezja je npr. Virtex-II Pro proizvajalca Xilinx, ki vsebuje do 4 procesorje PowerPC 405 [ss5]. Na ta vezja lahko gledamo kot vmesno točko med namenskimi vezji in mikroprocesorji, saj združujejo lastnosti obojih. Omogočajo enostavno programiranje, kot je to pri mikroprocesorjih, po drugi strani pa omogočajo tudi porazdeljeno obdelovanje podatkov s po meri prilagojeno SO. Ta vezja so cenovno ugodna iz istega razloga kakor mikroprocesorji – njihova enostavna programabilnost jih naredi vseuporabne in tako primerne za masovno proizvodnjo. Vseuporabnost je posledica predpripravljene mreže logičnih celic, ki jih je realno nemogoče v celoti izkoristiti, zato je, v primerjavi z namenskimi vezji, učinkovitost teh vezij inherentno manjša in iz istega razloga

so tudi hitrosti delovanja manjše. A to je pač kompromis, ki smo ga pripravljeni plačati za prilagodljivost in porazdeljeno obdelovanje. Proizvajalci so glede učinkovitosti naredili še korak naprej in poleg programirljivih logičnih celic v vezja vgradili še pogosto uporabljene elemente SO, kot npr. izredno hitre komunikacijske vmesnike, množilnike, procesorska jedra ipd. Vse to naredi ta vezja izredno prikladna tudi za raziskovanja SNSPO, saj omogoča enostavno vključevanje elementov SO za poljubno porazdeljeno obdelovanje podatkov in hitro ovrednotenje učinkovitosti celotnega sistema. Vezja FPGA se programira v jeziku VHDL ali Verilog in če naredimo primerjavo z mikroprocesorji, kjer se stavki programa prevedejo v binarno kodo, ki, naložena v spominu, določa zaporedje obdelovanja podatkov v centralni obdelovalni enoti, se stavki v jeziku za opis strojne opreme prevedejo v binarno kodo, ki določi tako strojno funkcionalnost logičnih celic (tj. SO) kot tudi povezavo med njimi in vgrajeno nespremenljivo SO, kar določa podatkovno pot obdelave.

## *2.4 Vmesna pot – SNSPO*

Skupaj z njihovimi prednostmi in pomanjkljivostmi smo predstavili tri glavne skupine vezij, ki omogočajo gradnjo sodobnih elektronskih sistemov (procesorje, kot predstavnike von Neumannove arhitekture, in namenska vezja ter vezja FPGA, kot predstavnike arhitekture, ki omogoča porazdeljeno obdelavo). Rezultat raziskovanj različnih načinov izvedb sistema so današnje načrtovalske metode, ki so izpopolnjene v obeh robnih primerih, tj. tako pri programiranju von Neumannove arhitekture kot tudi v načrtovanju namenskih vezij. Problem pa nastane, če želimo ti dve področji združiti, da bi pridobili prednosti obeh. S tem se ukvarja SNSPO, katerega predmet raziskovanja je vmesna pot, ki se nahaja med vseuporabnimi in enostavno programirljivimi procesorji, ki podatke obdelujejo tako, da jih iz spomina enega za drugim prenašajo v "obdelovalno" enoto in nazaj, in med namenskimi vezji, ki podatke obdelujejo porazdeljeno in po potrebi paralelno, tj. brez prenašanja sem ter tja. Na eni strani imamo vseuporabnost, prilagodljivost, sekvenčno obdelavo, visoko frekvenco delovanja, preveliko porabo in von Neumannovo pretočno ozko grlo. Na drugi strani imamo namensko SO, manjšo prilagodljivost, porazdeljeno obdelavo, nižjo frekvenco delovanja, nižjo porabo. Vmes imamo SNSPO, ki se tesno prepletajoče ukvarja z načrtovanjem po meri narejene SO ter PO in z učinkovito integracijo obeh delov že na sistemskem nivoju. Istočasno skušamo odgovoriti tudi na vprašanje učinkovite abstrakcije, saj bo edino tako mogoče obvladati vztrajno povečujočo kompleksnost sodobnih elektronskih sistemov. Šele z obvladanjem te vmesne poti bo mogoča učinkovita izraba čedalje zmogljivejših vezij FPGA.

Da bi razumeli interese raziskovanja področja SNSPO, bomo prikazali dva povsem različna poteka načrtovanja elektronskih sistemov, ki pojasnita, zakaj je potrebno hkratno obvladovanje tako programske kot tudi strojne opreme. Načrtovanje vsakega sistema se začne z neko začetno idejo, ki jo je potrebno pretvoriti v bolj ali manj formalno specifikacijo; kaj mora sistem delati (algoritem), kakšne so njegove omejitve/zahteve (cena celotne aplikacije, poraba, čas izvajanja itd.) in kakšna bo platforma (arhitektura). Glede na to, ali smo se odločili za programsko ali strojno rešitev – izbira platforme, algoritem opišemo ali v želenem programskem jeziku (npr. C) ali v jeziku za opis strojne opreme (HDL). Ko je opis algoritma zaključen, ga lahko preverimo v simulatorju ali pa na ciljni strojni opremi. Če se izkaže, da določenih omejitev/zahtev nismo izpolnili, je potrebno poiskati problematična področja izvajanja algoritma (postopek profiliranja) in preučiti možnosti, kako postopek izvajanja izboljšati. Lahko zadostuje že sprememba v algoritmu, obstaja pa tudi verjetnost, da bo potrebno poseči po bolj drastičnih ukrepih. Če se program, ki teče na procesorju, ne izvede dovolj hitro, potrebujemo hitrejši procesor, kar lahko implicira preveliko porabo. V primeru strojne rešitve pa lahko ugotovimo, da zaradi množice nastavitvenih parametrov potrebujemo veliko in kompleksno nadzorno vezje, od katerega pa se ne zahteva velika hitrost delovanja. Ugotovimo, da lahko programsko izvedbo algoritma pohitrimo, če procesor razbremenimo izvajanja računsko intenzivnih delov algoritma in te opravimo v namenski strojni opremi. Po drugi strani pa se lahko kompleksnosti strojnih nadzornih vezij izognemo tako, da uporabimo prilagodljivejšo programsko opremo, ki teče na procesorju.

### *2.4.1 Heterogenost*

Učinkovitost implementacije algoritma lahko, celovito ovrednoteno, izboljšamo, če ga razdelimo na del, ki se izvaja v PO, in del, ki se izvaja v SO [p1]–[p19]. Da bi zagotovili učinkovito in sinhrono delovanje posameznih sklopov, se mora načrtovalec spopasti z dvema velikima problemoma: a) potrebuje detajlno poznavanje obeh področij, tako PO kot tudi SO, in b) ker sodobna načrtovalska orodja še ne podpirajo enovitega povezovanja SO in PO, tako s stališča opisa, prevajanja, sinteze, razhroščevanja ipd., je odvisno predvsem od njegove iznajdljivosti, kako bo posamezne dele povezal v zaključeno enoto. Sopomenka za omenjen sklop problemov je *heterogenost* in se pri raziskovanju SNSPO kaže v nepovezanosti ter raznolikosti pristopov in potrebnih orodjih. Problem heterogenosti je velik, zato mu bomo v disertaciji namenili še veliko pozornosti. Če na načrtovanje sistema pogledamo z drugačnega zornega kota, lahko ugotovimo, da je heterogenost posledica umetno narejene delitve

postopka implementaciji aplikacije na dva ločena načina. Za prvega je značilno nenehno prenašanje ter obdelovanje podatkov, skladno z von Neumannovo arhitekturo, zaznamovan pa je z enostavnim opisom algoritma v enem izmed programskih jezikov, ki ga je potrebno prevesti v binarno kodo, razumljivo procesorju; ta se vpiše v spomin in nadzira delovanje posameznih sklopov. Za drugega je značilna vnaprej konstruirana in po meri prilagojena strojna oprema, zaznamuje ga metoda načrtovanja vezij ASIC ali FPGA v enem izmed jezikov za opis strojne opreme ali celo na nivoju fizične postavitve. Na koncu oba postopka izvedbe aplikacije ponudita rešitev iste naloge, ki pa se razlikujeta v učinkovitosti, porabi energije, hitrosti delovanja, stroškov načrtovanja ipd. Do delitve je prišlo zato, da sta možna t. i. način razvoja na platformah in uporaba abstrakcije ter s tem obvladovanje kompleksnosti načrtovanja sistema, zlasti s ponovno uporabo obstoječih rešitev. Ker do sedaj še ni bilo potrebe po tesnem sočasnem sodelovanju obeh področij, so okolja, pristopi pri načrtovanju in razpoložljivi jeziki za opisa algoritma različni. To pomeni, da je odvisno predvsem od iznajdljivosti načrtovalca, na kakšen način bo uspel povezati te nehomogene sklope v skladno delujočo celoto. Problem nastane, če kljub uspešno narejeni delitvi na SO in PO ter povezavi obeh delov sistemske zahteve niso povsem izpolnjene; npr. ugotovimo lahko, da prihaja do konfliktov pri izrabi sistemskih sredstev. Vodila in komunikacijske enote so en tak primer sistemskih sredstev, ki jih raziskovalci pri SNSPO dolgo časa niso pravilno upoštevali, zato pri predhodni analizi delitve časi komunikacije niso bili pravilno pridobljeni in posledično tudi končni rezultat ni bil želen [p24].

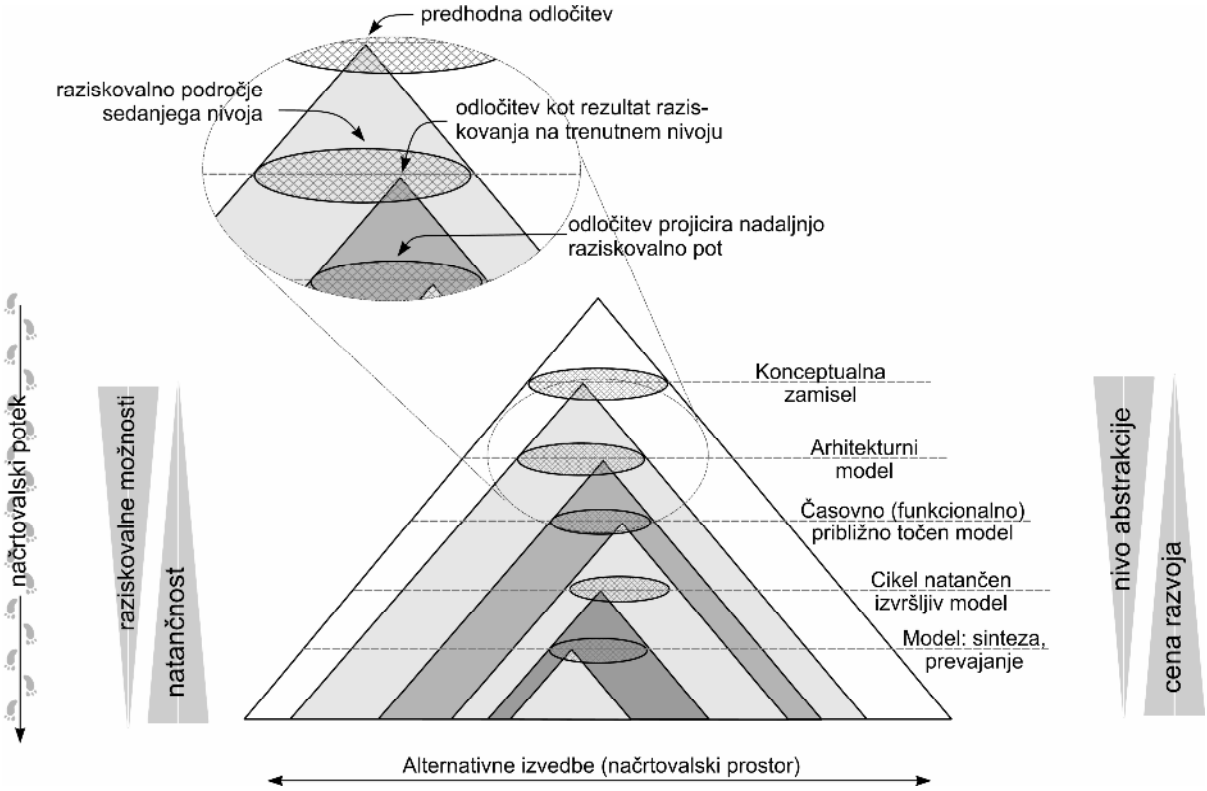
Raziskovalci SNSPO se ukvarjamo s problemom, kako premostiti načrtovalsko vrzel med SO in PO, in iščemo način, ki bi načrtovalcu omogočal učinkovito hkratno uporabo obeh področij. Težava, ki jo vidimo, je vrzel, zakoreninjena globoko v izobraževalnem sistemu, ki marsikje smeri a priori deli na strojni (elektronika) in programski (računalništvo) del. Ta delitev samo otežuje kasnejšo integracijo obeh delov zaradi povsem konceptualnih razlik v načinu razmišljanja, ki je potreben za učinkovito implementacijo določene naloge v posameznem področju [p6], [p17], [p20]. Kot bomo podrobneje predstavili v nadaljevanju, smo izbrali pot, s katero skušamo uporabiti čimveč dosedanjega znanja in orodij, istočasno pa želimo področji SO in PO čimbolj povezati, da bi tako načrtovalcu karseda zakrili vrzel.

### *2.4.2 Sistemske odločitve*

Trenutno uveljavljeni poteki načrtovanja elektronskih sistemov načrtovalcem ne nudijo učinkovitih in primernih orodij ter mehanizmov, ki bi jim pomagali ovrednotiti posamezne

odločitve na sistemskem nivoju. Obravnava tega problema je običajno v domeni večjih raziskovalnih skupin, ki imajo možnost postavitve celotnega načrtovalskega poteka, kot npr. [rs1]–[rs7]. Če se vrnemo na prej omenjena zgleda poteka načrtovanja elektronskega sistema, je lahko jasno, da nekatere odločitve niso rezultat premišljene primerjave posameznih načrtovalskih možnosti, temveč so to ad hoc odločitve, osnovane na preteklih izkušnjah z načrtovanjem podobnih sistemov. Pomembno je razumeti, da je delitev na SO ali PO *zelo pomembna sistemska odločitev*, saj usmeri vso nadaljnje raziskovanje; vse optimizacije sledečih problemov obstajajo le v smeri predhodne višjenivojske odločitve. Ob tem se moramo zavedati še dveh zelo pomembnih dejstev. Prvič, izvedba algoritma se lahko zelo razlikuje glede na to, ali je namenjen SO ali PO [p7]; če torej izberemo izvedbo v PO in ugotovimo, da rešitev ni povsem ustrezna in nato naknadno pospešimo samo kritične dele s pomočjo SO, lahko pridemo do t. i. lokalne optimizirane rešitve globalno ne najbolj ustreznega načina implementacije. Podobno se nam to lahko zgodi pri komplementarni izvedbi z začetno implementacijo v SO. Drugič, nižji kot je nivo abstrakcije, ki je uporabljen za ovrednotenje, več truda je potrebnega za pridobitev modela, ki ovrednotenje določenih sistemskih odločitev omogoča, in dražje je raziskovanje različnih možnosti.

Slika 4 prikazuje potek načrtovanja sistema in obseg raziskovanja na posameznih nivojih abstrakcije. Vrh piramide predstavlja začetek načrtovanja, kjer postavimo konceptualno zamisel o sistemu; sistem predstavimo npr. v obliki diagramov, na visokem nivoju abstrakcije, najpogosteje v neformalni obliki, izvedba je še zelo nenatančno določena, raziskovalne možnosti so še povsem odprte in porabljeno število inženirskih ur je še nizko. Možnosti raziskovanja v določenih fazah načrtovanja sistema so predstavljene z elipsami – predstavljajo obseg potencialnih odločitev glede implementacije, ki bi jo lahko izbrali. Višji ko je nivo abstrakcije, tj. višje ko smo v piramidi, večji je končni, tj. spodnji, načrtovalski prostor, na katerega lahko našo izbiro projiciramo. Načrtovalčeva naloga je, da iz množice potencialnih rešitev izbere najprimernejšo, ki nadaljnji potek raziskovanja projicira na zožen načrtovalski prostor. Tekom napredovanja v načrtovanju sistema se nivo abstrakcije opisa sistema niža, ker sistem postaja čedalje bolj določen, cena razvoja se z vloženimi človek-urami veča in raziskovalne možnosti se zaradi predhodno narejenih odločitev čedalje bolj zapirajo. Postopoma pridemo do končne točke, ko imamo pred seboj model sistema, ki predstavlja eno samo možnost in ga lahko uporabimo za dokončno fizično implementacijo sistema.




Slika 4 – Piramida abstrakcije

Postopek iskanja ustrezne (dovolj dobre) implementacije sistema lahko ponazorimo z iterativnim postopkom (Slika 4, povečan del). V vsakem koraku trenutno področje, ki je projekcija predhodne odločitve, raziščemo, tj. ovrednotimo posamezne točke tega področja, in izberemo najprimernejšo, tj. naredimo odločitev, ki bo nakazala smer nadaljnjega poteka načrtovanja. Če na določenem nivoju ugotovimo, da na podlagi predhodnih odločitev nobena razpoložljiva izvedba ni ustrezna, se moramo vrniti na predhodni nivo in se tam, sedaj z razširjenim znanjem, odločiti še enkrat (Slika 4 tega zaradi preglednosti ne prikazuje).

Trenutna načrtovalska orodja še ne podpirajo pristopa, ki bi omogočal enovit opis in ovrednotenje odločitev na vseh stopnjah načrtovalskega poteka. Posledica je daljša načrtovalska pot, preden ugotovimo ne/ustreznost določene načrtovalske odločitve, ki je v najboljšem primeru samo eno stopnjo više. Glede na to, da so človek-ure, namenjene razvoju določenega sistema, običajno omejene, je posledično raziskano načrtovalsko področje manjše.

Drug razlog, zakaj vpeljujemo abstrakcijo, je vztrajno povečujoča se kompleksnost sodobnih sistemov. Uporaba abstrakcije je po našem mnenju ključnega pomena za učinkovito obvladovanje kompleksnosti, ki nam ob primerni podpori nudi tudi povratno informacijo o zgodnjih sistemskih odločitvah. Slednje predstavlja tudi enega izmed naših prispevkov k

znanosti; metodologiji ovrednotenja abstraktnega višjenivojsko opisanega sistema bomo namenili pozornost v poglavju 4.4.

## *2.5 Od kod pride pohitritev/učinkovitost*

Do sedaj smo predstavili razkorak v tehnoloških in načrtovalskih zmožnostih, predstavili ključne razlike med centralnim in porazdeljenim obdelovanjem podatkov ter nakazali vmesno smer načrtovanja, tj. SNSPO. Pri tem smo se poleg splošne predstavitve osredotočili tudi na prednosti in pomanjkljivosti določene poti ter izpostavili težave, ki jih morajo načrtovalci ali raziskovalci premagati na posamezni načrtovalski poti. Da pa bi lahko učinkovito pristopili k načrtovanju elektronskega sistema in s tem tudi k raziskovanju SNSPO, bomo na tem mestu predstavili izsledke ovrednotenja dejavnikov, ki so ključnega pomena pri zagotavljanju pohitritve in s tem posledično učinkovitosti [p19]. V nadaljevanju se bomo omejili na primerjavo med vezji FPGA in splošno namenskimi procesorji, kar sicer predstavlja primarni interes našega raziskovanja, so pa zaključki glede pristopov načrtovanja vsesplošno uporabni.

Da bi lahko razumeli, od kod pride pohitritev, moramo najprej predstaviti nekaj osnovnih relacij. Pohitritev definiramo kot razmerje med številom urinih ciklov, potrebnih za obdelavo iste količine podatkov na procesorju in vezju FPGA[8] (enačba (4)). Predpostavimo iterativno obdelavo podatkov, tako da je število urnih ciklov procesorja določeno z enačbo (5), kjer $instr_{iter}$ predstavlja število instrukcij ene iteracije, $N_{iter}$ predstavlja število iteracij in $UCI$ predstavlja povprečno število urinih ciklov, potrebnih za izvedbo ene instrukcije. Instrukcije iteracije lahko nadalje razdelimo na podporne ($instr_{pod}$), ki služijo nadzoru poteka izračunavanja, in tiste operacije, ki direktno služijo obdelavi podatkov ($instr_{oper}$).

$$pohitr = \frac{ur.cikl_{proc}}{ur.cikl_{FPGA}} \tag{4}$$

$$ur.cikl_{proc} = instr_{iter} \cdot N_{iter} \cdot UCI = \left(instr_{pod} + instr_{oper}\right) \cdot N_{iter} \cdot UCI \tag{5}$$

$$ur.cikl_{FPGA} = \frac{N_{iter}}{P_{iter}} \tag{6}$$

[8] Najpomembnejši del tipičnih aplikacij, ki jih je mogoče zelo učinkovito implementirati v vezjih FPGA, predstavljajo predvsem operacije, ki se izvajajo nad tokom vhodnih podatkov, kot je to npr. pri avdio in video obdelavi signalov. V nadaljevanju poglavja bodo privzete takšne aplikacije. Seveda pa so faktorji pohitritve odvisni od načina izvedbe in uporabljene SO. Za podrobnosti naj si bralec pogleda [p19].

$$učink = \frac{instr_{oper}}{instr_{iter}},\ neučink = \frac{1}{učink} \tag{7}$$

V vezjih FPGA lahko urin cikel določimo s trajanjem iteracije, posledično ena iteracija traja en urin cikel, število ciklov, ki so potrebni za obdelavo vseh podatkov, pa je določeno s številom sočasno delujočih zank $P_{iter}$ (enačba (6)). Učinkovitost definirajmo s pomočjo števila instrukcij določene iteracije, ki služijo neposredni obdelavi podatkov ($instr_{oper}$), in števila vseh iteracij ($instr_{iter}$). Neučinkovitost naj bo recipročna učinkovitosti (enačba (7)).

Sedaj lahko zapišemo enačbo, ki razloži od kod pride pohitritev:

$$pohitr = P_{iter} \cdot instr_{oper} \cdot neučink \cdot CPI \tag{8}$$

Direktna prednost vezij FPGA pred procesorji je možnost *sočasnega izvajanja večjega števila iteracij*. $P_{iter}$ predstavlja razmerje števila sočasno delujočih sklopov, ki lahko izvajajo iteracije (pri vezjih FPGA je omejitev velikost programabilne mreže, procesorji z enim jedrom lahko izvajajo samo eno iteracijo naenkrat, pri procesorjih VLIW je število sočasnih iteracij večje). Drug faktor, $instr_{oper}$, predstavlja število operacij, ki služijo neposredni obdelavi podatkov. To število je lahko kvečjemu večje od števila operacij vezja FPGA, saj tu določene operacije za izvedbo ne potrebujejo časa (npr. premikanje, množenje z 2 ali izbiranje posameznih bitov je tu narejeno enostavno s povezavami). *Neučinkovitost* je posledica tega, da procesorji obdelavo podatkov časovno multipleksirajo na eno podatkovno pot, medtem ko je v vezjih FPGA podatkovna pot prilagojena nenehni obdelavi toka podatkov. Porazdeljen spomin prednost vezij FPGA samo še poveča, saj podatkov ni potrebno vmesno zapisovati v centralni spomin in kasneje zopet prebirati. V von Neumannovem modelu je nadzor poteka obdelave združen s samo obdelavo podatkov v program, ki se izvaja zaporedno, na drugi strani sta v vezjih FPGA ta dva "mehanizma" ločena; vejitve in skoki so izvedeni kot del podatkovne poti. Članek [p19] predstavi implementacijo treh tipov aplikacij, ki zahtevajo obdelavo nad tokom podatkov, na štirih vrstah SO. Avtorji zaključijo, da je najpomembnejši faktor instrukcijska učinkovitost, iz česar lahko še enkrat ugotovimo, da je problem v von Neumannovem sekvenčnem modelu in da moramo raziskovati v smeri, ki bo omogočala osvoboditev od tega modela.

# 3 Pregled obstoječih metod in sorodnega dela

Akademska sfera se s področjem SNSPO bolj ali manj aktivno ukvarja že zadnjih 10–15 let. Npr. raziskovalci projekta Polis [rs2.b], za katerega lahko napišemo, da predstavlja prvo resnejše pionirsko raziskovanje področja SNSPO, so prve raziskovalne rezultate objavili leta 1992. Na raziskanost in obvladovanje tega področja delno vpliva dejstvo, da se je industrija dolgo lahko izogibala vlaganju v razvoj obvladovanja SNSPO. Neučinkovitost, ki smo jo predstavili v prejšnjem poglavju, je uspešno kompenzirala ločeno – tj. z napredkom v tehnologiji izdelave integriranih vezij na eni strani in s programskimi orodji za učinkovitejše obvladovanje PO na drugi strani. Delno je sedanje stanje tudi posledica kompleksnosti področja, ki zahteva celovitejši pristop in zelo obsežno raziskovalno delo, za katero pa zadostna motivacija še ni prisotna dovolj dolgo. Z večanjem kompleksnosti integriranih vezij in elektronskih sistemov potreba po obvladovanja SNSPO vztrajno raste.

Rezultat omenjenega je veliko število objavljenih raziskovalnih dosežkov s področja SNSPO. Zaradi predstavljene kompleksnosti se je z raziskovanjem sprva spopadala samo akademska sfera, ki pa se je bolj osredotočila na metodologije, kot pa na celovita programska ogrodja. Celovite izvedbe, ki bi na učinkovit način podpirale SNSPO od začetne ideje do končne implementacije, so zelo redke ali pa jih skorajda ni. Po eni strani lahko najdemo zelo veliko raziskav, ki se ukvarjajo z ozko usmerjenimi podsklopi SNSPO, kot npr. [p24], in se tako osredotočijo v ciljno področje, da celotnega poteka ne uspejo zajeti. Rezultati takšnih raziskav so praviloma sámoovrednoteni[9]. Na drugi strani pa najdemo metodologije, ki celovito obravnavo SNSPO sicer podpirajo, npr. [rs5], je pa njihova uporabnost lahko močno okrnjena; uporabljeni so npr. lastni/nestandardni jeziki, metodologija je primerna samo za določen tip aplikacij ali celo SO, velikokrat je potrebnih veliko ročnih prilagajanj ipd.

[9] To pomeni, da je vhodni nabor podatkov, nad katerim se izvaja določen algoritem, ustvarjen umetno in da je rezultat obdelave ponovno ovrednoten umetno (tj. s stališča obravnavanega podproblema). Pristop, ki ga zagovarjamo, se razlikuje od sámoovrednotenega v tem, da so vhodni podatki za algoritem določenega pristopa pridobljeni iz realnega primera načrtovanja sistema in da je algoritem ovrednoten s stališča njegovega vpliva na učinkovitost v celoti izvedenega sistema.

Vložiti smo morali relativno veliko truda, da smo množico raziskovalnih objav postopoma prečistili in ugotovili temeljne pristope posameznih metodologij različnih raziskovalnih skupin ter dejansko uspešnost implementiranih orodij. Ker smo mnenja, da je za nove raziskovalce ključnega pomena celovit vpogled v dosedanje raziskovalne dosežke, bomo nadaljevanje tega poglavja namenili predstavitvi posameznih raziskovalnih skupin, na katere smo tekom našega raziskovanja naleteli največkrat in so naše raziskovanje tudi najmočneje usmerjale [rs1]–[rs7]. Pomembno je razumeti, da je večina znanstvenih objav posameznikov teh skupin izpeljana iz metodologije skupine in je zato predstavljene dosežke lažje ovrednotiti, če je postavitev v širši kontekst SNSPO jasna. Nemalokrat ima lahko namreč bralec vtis, da je glede SNSPO že vse raziskano in implementirano – po podrobni analizi pa ugotovimo, da temu še zdaleč ni tako.

Na seznamu literature in referenc smo navedli omenjene pomembnejše raziskovalne skupine (Raziskovalne skupine, str. 114). Navedeni so podatki o domači strani skupine ali projekta ter tudi vodja, ki običajno nosi zasluge za organizacijo in vodenje projekta. Vodjo običajno najdemo zapisanega kot enega izmed soavtorjev (običajno zadnjega) in služi kot zelo pomemben podatek za uvrščanje posameznega raziskovalnega članka (povezava z določeno raziskovalno skupino). Bralec lahko na spletnih straneh raziskovalnih skupin pridobi podrobnejše informacije za nadaljnjo podrobnejšo raziskavo glede metodologije, razvojnega okolja, če ga imajo, in popoln seznam objavljene literature.

Ko se bomo v nadaljevanju osredotočili na posamezne podsklope SNSPO, bomo posamezne raziskovalne skupine in njihove pristope k SNSPO še nekajkrat primerjalno uvrstili. Na tem mestu so raziskovalne skupine zbrane načrtno in predstavljene celovito, da bralec pridobi pregleden vpogled in informacije, ki so potrebne za nadaljnje raziskovanje posamezne metodologije.

## *3.1 Polis*

Polis [rs2.b] je razvojno okolje za vgrajene sisteme s prevladujočim odločitvenim potekom. Publikacije, povezane s projektom, so bile objavljane v letih od '92 do '98[10] in projekt, kot je bil v prvotni obliki, ni več aktiven. V predstavitev smo ga vključili zato, ker projekt predstavlja enega izmed prvih pomembnejših raziskovalnih aktivnosti na področju

[10] http://embedded.eecs.berkeley.edu/Respep/Research/hsc/publications.html

SNSPO. Po podrobnejšem pregledu lahko ugotovimo, da projekt Polis izhaja iz istega oddelka Berkeleyeve univerze, kot še Ptolemy [rs2.a] in tudi Metropolis [rs2.c], zato smo jih v referencah tudi združili skupaj, znotraj [rs2]. Projekta Ptolemy in Metropolis bomo podrobneje predstavili v nadaljevanju.

Raziskovalci projekta Polis so se usmerili v podporo za razdelitev sistema na podsklope in pomoč pri izbiri mikrokrmilnika ter periferije. Razvojno okolje Polis je nudilo podporo za ustvarjanje kode C programskega dela, vključno s preprostim RTOS za mikrokrmilnik, in kode HDL za sintezo SO. Čeprav z današnjega stališča mikrokrmilnik s preprosto periferijo predstavlja razmeroma nezahtevno platformo, je projekt vseeno zastavil nekaj zelo pomembnih izhodišč za raziskovanje SNSPO. Za predstavitev sistema so uporabili namenski računski model (uporabili so model CFSM (*codesign finite state machine*)), lotili so se problema opisa in simulacije heterogenih komponent, ponudili so podporo za ovrednotenje cene posameznih odločitev in kar je morda najpomembneje – postavili so temelje za delitev opisa sistema na funkcionalnost in arhitekturo, kar bomo v nadaljevanju še podrobneje obravnavali.

## *3.2 Jerraya in ostali*

Jerraya in ostali [rs1] so razvili visokonivojsko metodologijo načrtovanja sistema, ki je prvotno namenjena heterogenim večprocesorskim sistemom na čipu [p1], [p2], [p3]. S pomočjo njihove metodologije lahko sistem opišemo s pomočjo mreže virtualnih komponent, opisanih z obnašanjem in priključki, ki so med seboj povezane s kanali. Od tukaj tudi izvira ime: metodologija za načrtovanje večprocesorskih sistemov na čipu, osnovana na opisu komponent (*component-based methodology for MPSoC design*). Virtualne komponente so sestavljene iz okvirja (*wrapper*), ki komponento objame, in notranje komponente, ki lahko predstavlja kakršno koli kombinacijo programskih opravil in/ali strojnih funkcij. Naloga okvirja je prilagajanje notranjih in zunanjih dostopov. Razvoj sistema se začne z abstraktnim modelom arhitekture (virtualna arhitektura), kjer je sistem predstavljen v obliki hierarhične mreže virtualnih komponent (ali modelov po terminologiji TLM). Ta model služi samo kot abstrakcija celotnega SoC. Modela ni mogoče uporabiti za sintezo in ga ni mogoče simulirati, ker ima okvir opisane samo zahteve, ne pa tudi obnašanja. Glavni cilj njihove metodologije načrtovanja je samodejno generiranje okvirjev, da bi pridobili detajlen opis arhitekture, ki bi bil uporaben za sintezo in simulacijo. Njihova metodologija podpira tudi heterogenost;

komponente PO se izvajajo na procesorju (lahko tudi časovno rezinjene znotraj OS), na vodilo do katerega dostopa(jo) procesor(ji) pa je lahko priključena tudi namenska SO. Da bi omogočili samodejno kreiranje vmesnikov, so se metodično lotili abstrakcije okvirjev priključenih opravil PO in komponent SO. Vpeljali so več nivojev abstrakcije, ki omogoča prečkanje meje heterogenosti, ki jo ustvarjajo vmesniki. Abstrakcija in vmesniki omogočajo hitrejše raziskovanje, preglednejše načrtovanje, delitev dela in ponovno uporabo rešitev. V članku [p15] je predstavljen primer implementacije metodologije na simulacijsko ogrodje SystemC in s podporo za OS.

## *3.3 Metropolis*

Metropolis [rs2.c], [p4], [k12.a] je razvojno okolje, ki nudi infrastrukturo za načrtovanje VS znotraj enovitega programskega ogrodja, ki natančno določa potek načrtovanja. Poudarek je na modeliranju heterogenih sistemov na različnih nivojih abstrakcije. *Metamodel Metropolis* je jezik za določanje mreže sočasno delujočih objektov, pri čemer je obnašanje mreže formalno določeno. Metamodel lahko predstavi vse ključne stopnje načrtovalskega poteka. Eno izmed temeljnih področij raziskovanja projekta predstavlja premagovanje velikega načrtovalskega razkoraka med funkcionalnim opisom določene funkcije navzdol do njene implementacije. Premagovanja razkoraka so se lotili tako, da so vpeljali več nivojev platform, od katerih vsaka služi raziskovanju načrtovalskega prostora v omejeni smeri. Funkcionalnost sistema je predstavljena z naborom objektov (procesov), ki svojo nalogo opravljajo sočasno, medtem ko komunicirajo drug z drugim. Implementacija komunikacije in procesiranja objektov je po zaslugi vmesnikov opravljena ločeno; objekt, ki implementira vmesnik, se imenuje medij. Naloga vmesnikov je tudi pospeševanje ponovne uporabe rešitev. Funkcionalnost arhitekture je modelirana z naborom storitev, ki jih arhitektura na razpolago nudi funkcionalnemu modelu. Za ovrednotenje implementacijske učinkovitosti glede zmogljivosti sistema (čas izvajanje, poraba energije ipd.) metodologija uporablja pristop kvantitativnih merilnikov. Mapirna mreža zajame funkcionalno in arhitekturno mrežo ter med njima določi mapiranje. Mapirno mrežo lahko razumemo tudi kot implementacijo določenih storitev, kjer funkcionalna mreža nudi algoritem in arhitekturna učinkovitost. Nabor različnih mapirnih mrež, ki nudijo iste storitve z različnimi cenami, tvori platformo. Koncept platform se tekom načrtovalskega postopka pojavlja rekurzivno; *metamodel Metropolis* omogoča enovit opis te rekurzivne paradigme načrtovanja, osnovane na platformi, s pomočjo formalne semantike, ki natančno določa obnašanje mreže. Poleg tega razvojno okolje Metropolis nudi

tudi nabor orodij za analizo, verifikacijo, simulacijo in sintezo, ki omogočajo integracijo celotnega poteka načrtovanja znotraj enovitega programskega ogrodja.

## 3.4 AAA

Metodologija AAA (*Algorithm-Architecture Adequation*) [rs5], [p5], [p13], [p14], [p25], [p26] je osnovana na teoriji grafov, s pomočjo katerih je mogoče modelirati strojno arhitekturo in algoritem aplikacije (oba sta predstavljena s svojim grafom). Vozlišča arhitekturnega grafa predstavljajo enega izmed 4 vrst avtomata FSM: operator, komunikator, spomin in vodilo/multiplekser/demultiplekser. Usmerjene povezave med avtomati FSM so ponazorjene z vejami grafa. Komplementarno, vozlišča grafa algoritma predstavljajo operacije in veje prenos podatkov med njimi. Na ta način je algoritem aplikacije predstavljen kot končno zaporedje operacij, izvajanih na avtomatih FSM. Kompleksne operacije je mogoče zgraditi s pomočjo hierarhičnega opisa operatorjev. Metodologija nudi način, kako samodejno pridobiti implementacijo s pomočjo formalnih transformacij grafov. Naloga implementacije je določiti prostorsko in časovno dodelitev operacij algoritma (vozlišča grafa algoritma) ter prenosa podatkov med operacijami (veje grafa algoritma) na arhitekturne operatorje (vozlišča grafa arhitekture). Da bi zadovoljili potrebe realnega časa, je časovna dodelitev narejena statično (tj. tekom programiranja in ne tekom izvajanja algoritma). Optimizacija implementacije predstavlja iskanje najboljše prostorske (in kasneje tudi časovne) dodelitve vseh operacij algoritma na operatorje arhitekture (od tu izvira ime metodologije). Pri dodeljevanju operacij je mogoče izbirati med vsemi operatorji, ki so sposobni izvedbe določene operacije. Problem dodelitve tako pripelje do optimizacijskega problema teže NP, ki ga rešujejo s pomočjo hevrističnih postopkov optimizacije. Predstavljeni načrtovalski potek metodologije AAA programabilnih vezij ne podpira direktno, je pa podpora za programabilna vezje vpeljana preko razširitvene metodologije – AAA-IC [p13]. Slednja metodologija opis algoritma razširi s faktoriziranimi grafi podatkovne odvisnosti, s katerimi so učinkovito predstavljena kakršna koli ponavljanja v poteku obdelave podatkov. Kakor trdijo avtorji, to omogoča preglednejši prikaz največje mogoče stopnje vzporedne obdelave podatkov, ki posledično lahko vodi do najučinkovitejše izrabe razpoložljive vzporedne obdelave podatkov, ki jo nudi arhitektura. Na žalost pa sta programski okolji, ki omenjeni metodologiji implementirata, trenutno še ločeni (SynDEx in SynDEx-IC), posledično je potrebno ločitev na SO in PO opraviti ročno.

## 3.5 Gajski in ostali

Gajski in sodelavci [rs3], [p9], [p10] so postavili metodologijo raziskovanja, ki je osnovana na jeziku SpecC in nadgradi njegove lastnosti načrtovalskega jezika sistemskega nivoja (*SLDL*). Štirje glavni deli njihove metodologije so: modeliranje[11], prečiščevanje, raziskovanje in sinteza. Prečiščevanje je proces nadgradnje določenega modela TLM, da bi s tem pridobili podrobnejši model, na podlagi katerega lahko pridobimo natančnejše odločitve o razvoju sistema. Tehtnejše sprejemanje načrtovalskih odločitev je načrtovalcu omogočeno preko vgrajene podpore za raziskovanje. Ker mora biti raziskovanje podprto na vseh nivojih TLM, morajo biti za ovrednotenja posameznih potencialnih načrtovalskih odločitev upoštevane različne načrtovalske metrike modela sistema – različni modeli TLM zahtevajo različna ovrednotenja. V [p9] lahko bralec spozna pristop hitrega raziskovanja načrtovalskega prostora, ki z metodologijo sovpada. Samodejni prehod med zaporednimi/sledečimi modeli TLM v okviru danih omejitev in optimizacijske metrike je narejen s pomočjo zadnjega dela metodologije, tj. sinteze. Načrtovalsko okolje SCE (*SoC Environment*) implementira predstavljeno metodologijo in se osredotoči še zlasti na njen tretji del, tj. raziskovanje. Okolje SCE omogoča grafično raziskovanje hierarhije in povezovanja v načrtovalskem jeziku sistemskega nivoja, samodejno analizo (s pomočjo orodja za profiliranje [p9]), mapiranje na arhitekturo, validacijo in sintezo. Raziskovanje se prične s specifikacijo funkcionalnega modela, napisano v jeziku C. Da bi lahko izkoristili prednosti načrtovalskega jezika sistemskega nivoja, ki jih ponuja modeliranje TLM, je potrebno začetno specifikacijo prečistiti v jezik SpecC. Koda se v nadaljnjem postopku profilira, da se pridobi profil aplikacije (algoritma), s pomočjo katerega je mogoče učinkovito raziskovanje mapiranja algoritma na arhitekturo skozi različne stopnje modeliranja TLM.

## 3.6 SPACE

Metodologija SPACE[12] [rs7] je usmerjena v hkratno modeliranje SO in PO ter ponuja podporo za lastnosti, ki jih običajno najdemo v OS realnega časa, kot npr. prednostno časovno dodeljevanje in prekinitev izvajanje opravil z nižjo prioriteto. V primerjavi z ostalimi

[11] Modeliranje vidimo kot zelo pomemben del visokonivojskega pristopa SNSPO, zato ga bomo podrobneje predstavili v poglavju 4.4. V poglavju 4.4.3 bomo predstavili naš originalni prispevek k znanosti z razširjenim pristopom modeliranja. S trenutno uveljavljenimi pristopi modeliranja se lahko bralec seznani npr. v [p10] in [k5.a].

[12] Kratica, ki pomeni SystemC Partitioning Aspects of Codesign Exploration.

raziskovalnimi metodami predstavlja ta način modeliranja visok nivo podpore tudi za PO, saj je pri ostalih poudarek predvsem na modeliranju SO. Modeliranje je osnovano na jeziku SystemC in v članku [p8] je prikazan opis arhitekture, ki omogoča modeliranje PO pod okriljem izbranega OS za delovanje v realnem času. Programski del opisa sistema je izveden znotraj simulatorja instrukcijskega nabora in je povezan z ostalimi deli SO, ki so opisani v SystemC, s pomočjo strojnih kanalov. Metodologija SPACE podpira tudi možnost enostavnega prehoda med opravili, ki se izvajajo v PO, in algoritmi, ki se izvajajo v SO. To je omogočeno s pomočjo posebnih programskih vmesnikov (API) in okvirjev, katerih naloga je prilagoditev opravila za izvedbo v okviru PO ali SO. Ker je opis opravila v obeh primerih enak, je s tem omogočeno enostavno raziskovanje načrtovalskega prostora različnih izvedb (mapiranje, časovno dodeljevanje in določanje prioritete niti opravil PO).

## *3.7 Ptolemy II*

Metodologija Ptolemy II temelji na komponentnem opisu razvoja sistema in razvojnem okolju za modeliranje, razvitem v Javi. V projektu Ptolemy [rs2.a] je raziskovalno delo osredotočeno na modeliranje in simulacijo sočasnih vgrajenih sistemov za delo v realnem času [p12]. Glavni poudarek projekta je na metodologiji za definiranje in izvedbo vgrajene PO skupaj s sistemom, v katerega je vgrajena. Različni sistemi so podprti z vrsto različnih računskih modelov (MoC), ki jih projekt podpira. Skupina je na področju računskih modelov zelo pomembna; v modelirnem okolju Ptolemy II imajo podprtih preko 15 različnih modelov. Njihovo obširno razlago in opis lahko bralec najde v [p12]. Uporaba različnih računskih modelov je pomembna zaradi dveh dejstev: a) izbira računskega modela je odvisna od tipa modela sistema, ki ga želimo opisati, in b) sposobnost pretvorbe opisanega modela v implementacijo je odvisna od uporabljenega računskega modela. Raziskovalci skupine Ptolemy zagovarjajo dejstvo, da izbira računskega modela vpliva na kakovost razvoja sistema. Računski modeli so uporabljeni za gradnjo izvršljivih modelov, ker pa je nabor podprtih modelov velik (za različne domene), so raziskovalci veliko truda namenili zagotavljanju načina za zlaganje in medsebojno povezovanje različnih računskih modelov. Modeli sistema so zgrajeni na principu medsebojno vplivajočih komponent. Računski model določa semantiko medsebojnega vpliva. V nasprotju (in komplementarno) z objektno orientiranim pristopom njihova metodologija zagotavlja način razvoja sistema, ki omogoča sočasno delovanje komponent sistema in komunikacijo med njimi. Sintaksa opisuje koncept modelov

sistema, njihovih portov in parametrov ter komunikacijske kanale. Semantika, ki je na sintakso povečini ortogonalna, je določena z izbiro računskega modela.

Raziskovalna skupina ima že bogato in obsežno raziskovalno zgodovino, kjer je nemalo število raziskovalcev pripomoglo k širšemu razumevanju heterogenega modeliranja. Od ostalih metodologij se projekt Ptolemy loči v tem, da se ne ukvarja s celotnim postopkom načrtovanja heterogenega sistema, temveč se osredotoča samo na modeliranje.

## *3.8 SPARK*

SPARK [rs4], [p16] je visokonivojsko programsko ogrodje za pretvorbo programskega jezika C v jezik VHDL, primeren za sintezo. Programsko ogrodje temelji na transformacijskih tehnikah, ki omogočajo pretvorbo opisa ANSI-C (z nekaterimi omejitvami) v trinivojsko vmesno predstavitev, ki zajame tako podatkovno kot kontrolno odvisnost. Vmesna predstavitev vsebuje graf poteka nadzora, graf poteka podatkov in hierarhični graf opravil. Dodatni vhodni podatki so predstavljeni v obliki knjižnice sredstev SO, v obliki časovnih omejitev, v obliki omejitev razpoložljivih sredstev in v obliki načrtovalčevih navodil glede raznih hevristik ter transformacij. Informacije o strukturi, skupaj z informacijami o operacijah in podatkovnih odvisnostih, ki so zajete znotraj vmesne predstavitve, omogočajo transformacije na nivoju izvorne kode in globalne odločitve glede premikov kode. V koraku pred sintezo (pred razporeditvijo sredstev, časovno dodelitvijo ter dodelitvijo sredstev) se izvede nabor grobo- in finozrnatih tehnik transformacij kode, ki poskrbijo za prerazporeditev kode z namenom raziskovanja paralelizma in pogojnih odvisnosti. V koraku razporeditve sredstev in časovne dodelitve, ki sledi, načrtovalec izbere sredstva za izvedbo in module. Korak časovne dodelitve temelji na tradicionalnih transformacijah visokonivojske sinteze, ki so nadalje izboljšane s tehnikami dinamične transformacije [p16]. V koraku dodeljevanja so operacije povezane ali dodeljene določenim komponentam SO; spremenljivke so povezane z registri, prenosi podatkov so dodeljeni vodilom ipd. Sledi korak ustvarjanja nadzora, kjer je ustvarjena kontrolna enota, ki skrbi za signale za nadzor časovnega dodeljevanja in povezovanja podatkov v podatkovnih poteh. V zadnjem koraku je ustvarjena koda VHDL (opis RTL), ki je primerna za sintezo.

Primerjalno s predhodno omenjenimi metodologijami SPARK SNSPO obravnava na nižjem nivoju. Usmeritev je mnogo ožja in je osredotočena na samodejen prehod opisa PO (ANSI-C) v opis SO (RTL VHLD). Taki pristopi niso pogosti, saj temeljijo na pretvorbi kode,

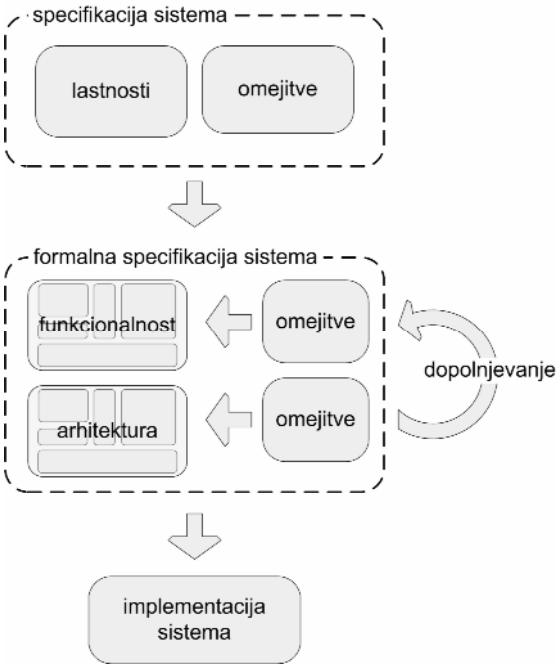
ki je bila namenjena za zaporedno izvajanje, v opis SO, ki je v osnovi namenjena sočasnemu izvajanju. Pristopi s samodejno pretvorbo ne nudijo optimalnih rezultatov saj se algoritmi za opis PO in SO razlikujejo [p18]. Podoben (in opuščen) poskus samodejne pretvorbe opisa PO v opis, primeren za SO, je predstavljen v [p7], le da so šli tu avtorji še korak dlje – tu je SO načrtovalcu povsem prikrita.

# 4 Elementi SNSPO

Za obvladovanje kompleksnosti načrtovanja heterogenih elektronskih sistemov, predstavljene v 2. poglavju, je potreben sistematičen pristop za uspešno izpeljavo postopka načrtovanja elektronskega sistema k SNSPO. Stopnje načrtovalskega poteka in prehodi med stopnjami morajo biti nedvoumno določeni, potrebni pa so tudi mehanizmi obvladovanja kompleksnosti, ki bodo predstavljeni v nadaljevanju. Za razumevanje in uvrščanje metodologij SNSPO posameznih raziskovalnih skupin je potrebno razumeti, da je različne mehanizme obvladovanja kompleksnosti mogoče uporabiti sočasno in med seboj prepletajoče. To je pri različnih metodologijah tudi običajno, zato je za njihovo razumevanje potrebno dobro poznavanje osnovnih konceptov. V nadaljevanju se bomo osredotočili na koncepte obvladovanja kompleksnosti, za katere smo ugotovili, da so še zlasti pomembni s stališča visokonivojskega pristopa k SNSPO. Najprej bomo predstavili celoten postopek SNSPO, tako da bo bralec lahko dobil celovit vtis o poteku načrtovanja, nato pa bomo posamezne visokonivojske koncepte obvladovanja kompleksnosti v okviru te slike predstavili podrobneje. Predstavitev poteka izvajanja SNSPO prikazuje slika 6, ki tudi zajema visokonivojske koncepte obvladovanja kompleksnosti, ki jih bomo predstavili v nadaljevanju.

## 4.1 Načrtovalski potek

Načrtovanje kompleksnega elektronskega sistema se prične z opisom oz. s specifikacijo sistema, ki zajame njegove visokonivojske lastnosti in omejitve. Če je začetni opis sistema predstavljen v obliki grafov, razpredelnic, v opisni obliki in v jeziku komunikacije med ljudmi, ta oblika opisa predstavlja neformalni opis sistema. Prej ko slej je potrebno ta opis pretvoriti v formalni opis [wp4], ki lastnosti in omejitve sistema zajame nedvoumno, preverljivo in konsistentno (slika 5). Nedvoumen opis predstavlja osnovo za implementacijo sistema, kjer je zahtevam sistema mogoče preverljivo slediti od njihovega izvora do končne implementacije. Pri formalnem opisu sistema je ključnega pomena, kaj mora sistem narediti, in ne (nujno), kako naj to narediti. V načrtovalskem postopku je pri transformaciji specifikacije v implementacijo mogoče uporabiti formalno verifikacijo [wp5] na osnovi dokazano pravilnih korakov dopolnjevanja lastnosti sistema.



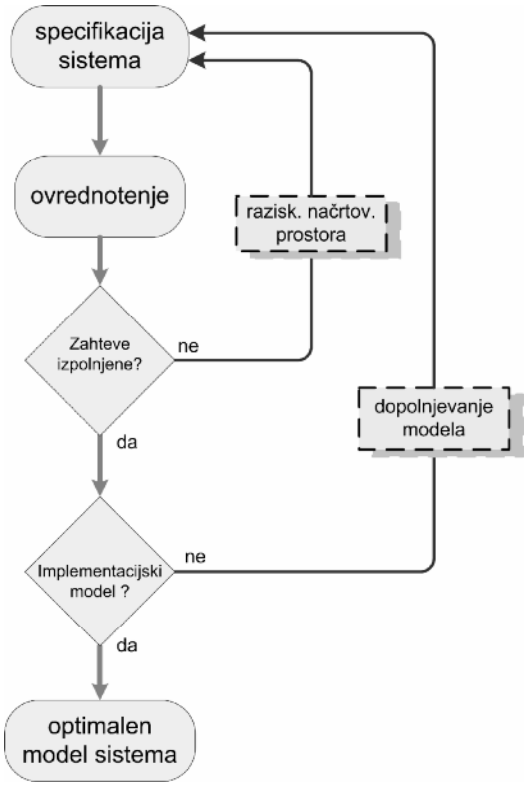

Slika 5 – Visokonivojska specifikacija sistema

V fazi zajema visokonivojskih lastnosti in omejitev se načrtovalci pogosto soočajo s težavo nezadovoljive podpore za formalno obliko opisa. Korak naprej v visokonivojskem opisu predstavlja na primer jezik UML [ss3], ki je namenjen za modeliranje raznolike palete domen in na različnih stopnjah poznavanja sistema. Semantiko modeliranja (pomen sestavnih elementov jezika) si v jeziku UML načrtovalec lahko priredi skladno z domeno opisovanega sistema. Uporaba jezika UML pa žal še ni široko prisotna, saj so orodja, ki bi ga podpirala, še v razvoju. Standard jezika je še razmeroma mlad in potrebno bo še nekaj časa, da bo na voljo ustrezen model izobraževanja načrtovalcev in bo uporaba jezika resnično koristna za povečevanje načrtovalskih zmožnosti. Zaradi tega prihaja pri načrtovanju do razkoraka med začetnim neformalnim opisom in formalno specifikacijo sistema, ki nedvoumno postavlja okvirje nadaljnjega razvoja sistema. Z opisano problematiko se ukvarja področje visokonivojskega modeliranja, ki je tesno povezano s pojmom abstrakcije. Abstrakcijo bomo poglobljeno predstavili v nadaljevanju v poglavju 4.4.

Poleg abstrakcije je dodaten mehanizem obvladovanja kompleksnosti sistema razdelitev na problemske domene[13], kjer določeno kompleksno nalogo porazdelimo na ožje, zaključene in lažje obvladljive segmente. S tem je pripravljena podlaga za ponovno uporabo sklopov rešitev

[13] Implementacijo razdelitve na domene lahko najdemo npr. v knjižnici STL (*standard template library*) jezika C. Prednosti sistematičnega pristopa so razložene v [k7].

in možnost neposrednejšega osredotočenja na samo problematiko področja. Pri opisu elektronskega sistema sta ti domeni običajno funkcionalnost (naloge, ki jih sistem opravlja) in arhitektura (platforma, ki izvaja naloge) – podroben opis arhitekturno-funkcionalne delitve bo sledil v poglavju 4.2. Opis sistema lahko znotraj zajema funkcionalnosti še nadalje razdelimo na logične podsklope. Pri tem je široko uporaben opis funkcionalnosti po komponentah, katere bistvo modeliranja sistema je opis, ki izračunavanje in komunikacijo obravnava ločeno. Podroben opis delitve sledi v poglavju 4.3.




Slika 6 – Načrtovalski potek SNSPO na sistemskem nivoju

Ko je model sistema (zahteve in predlagane rešitve) zajet, sledi njegovo ovrednotenje. Tako dobimo povratno informacijo, ali sistem, ki smo si ga zamislili, izpolnjuje naše zahteve in ali ga je možno realizirati. Če zahteve niso izpolnjene, je potrebno vhodni opis sistema popraviti (Slika 6, puščica raziskovanje načrtovalskega prostora). Če smo s trenutnim modelom sistema zadovoljni, sledi preverjanje, ali imamo dovolj informacij za implementacijo sistema. Če informacij ni dovolj, je potrebno model sistema nadgraditi z manjkajočimi detajli (Slika 6, puščica nadgrajevanje modela). Vsakič, ko model sistema izpolnjuje zahteve, ki jih je mogoče ovrednotiti, dobimo potrditev, da je zahtevano funkcionalnost – na podani arhitekturi, v okviru podanih omejitev in s podano prostorsko ter časovno dodelitvijo – mogoče implementirati. Končni cilj je model sistema, ki izpolnjuje

podane načrtovalske zahteve in ga je mogoče s pomočjo avtomatizacijskih orodij pretvoriti v končno izvedbo.

Delitev na domene, vodenje skozi različne nivoje abstrakcije, opis sistema s komponentami ter podporo za ovrednotenje modela sistema (poglavje 4.6) omogočajo okvirji, ki bodo podrobneje predstavljeni v poglavju 4.5.

## *4.2 Delitev na funkcionalnost in arhitekturo*

*Def. 1* **Funkcionalnost** določa oz. opisuje naloge sistema ali to, kaj sistem dela.

*Def. 2* **Arhitektura** predstavlja strojne gradnike za implementacijo funkcionalnosti.

*Def. 3* **Algoritem** določa način izvedbe funkcionalnosti. Predstavlja končen nabor enolično določenih pravil za rešitev določenega problema v končnem številu korakov.

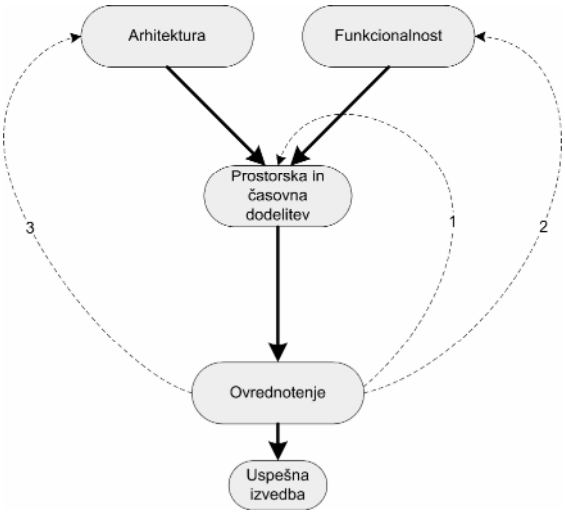
Ideja o ločitvi opisa sistema na arhitekturo in funkcionalnost se je prvič pojavila že pri raziskovalni skupini Polis [p4] in ta pristop sedaj uporabljajo skorajda vse raziskovalne skupine. Funkcionalnost določa naloge sistema, arhitektura pa predstavlja strojne gradnike, ki to omogočajo.

Izjema so metodologije, ki so osnovane na t. i. nevsiljivem načinu razvoja sistema, kjer je arhitektura načrtovalcu prikrita (podobno kot je to pri PO). Primer takšnega pristopa je npr. prikazan v [p7]. Ker pa nevsiljiv način razvoja ni obrodil uspešnih rezultatov, je postalo jasno, da lahko pridemo do optimalne zasnove kompleksnega sistema samo, če se v postopku načrtovanja arhitekturnih gradnikov jasno zavedamo in jih tudi upoštevamo. Kljub temu pa še vedno obstajajo poskusi samodejnega kreiranja manjšega dela SO za določene računsko zahtevne dele PO[14].

Prednosti pristopa delitve opisa sistema na arhitekturo in funkcionalnost lahko razložimo s pomočjo grafa Y [p38] (Slika 7). Ločen opis načrtovalcu omogoča, da arhitekturo in funkcionalnost sistema obravnava neodvisno drug od drugega, se njuni izvedbi tako podrobneje posveti in ju v kasnejši stopnji načrtovanja združi z mehanizmi prostorske ter časovne dodelitve. To istočasno predstavlja tudi osnovo za delitev dela med različne skupine načrtovalcev. Pristop podpira ponovno uporabo rešitev, saj lahko posamezne dele enega

[14] Takšen primer je na primer namenski prevajalnik podjetja Tensilica [ss9], ki določen del programske kode samodejno prevede v razpoložljiv manjši del SO.

sistema uporabimo ponovno pri gradnji modelov drugih sistemov. Določen arhitekturni opis lahko npr. uporabimo kot podlago za izvajanje funkcionalnosti različnih sistemov, ne da bi bilo za to potrebno opis arhitekture delati ponovno. Načrtovanje sistema poteka iterativno. V primeru, da rezultati ovrednotenja niso zadovoljivi, lahko najprej poizkusimo drugačno prostorsko in časovno dodelitev (1), nato poizkusimo na drugačen način opisati funkcionalnost (2) in šele nato spremenimo opis arhitekture (3), saj v okviru podanih funkcionalnih zahtev ni mogoče doseči zadovoljivih rezultatov.




Slika 7 – Graf Y [p43]

Združitev opisa arhitekture in funkcionalnosti v celovit opis sistema je mogoča, če je delitev narejena skladno z določenimi pravili. Opis arhitekture in funkcionalnosti si lahko predstavljamo kot dve ortogonalni komponenti, kateri sicer lahko obravnavamo vsako zase, celoten opis sistema pa bo smiseln šele takrat, ko ju bo mogoče učinkovito združiti. Ker ortogonalizacija obema komponentama (tako opisu arhitekture kot tudi funkcionalnosti) postavlja določene omejitve, mora biti model delitve postavljen čimbolj smotrno, da lahko predhodno predstavljene prednosti učinkovito izkoristimo. Združevanje opisa se izvede v koraku prostorske in časovne dodelitve; v tem koraku se določi, kateri arhitekturni podsklop (prostorska dodelitev) bo ob katerem času (časovna dodelitev) izvedel določen del funkcionalnosti. Za izvajanje določene funkcionalnosti so primerni vsi kvalificirani deli arhitekture.

Poleg izpolnjevanja zahteve za združevanje v enovit opis sistema z namenom končne implementacije mora biti arhitekturno-funkcionalni opis sistema odprt tudi za uporabo algoritmov, ki avtomatizirajo raziskovanje načrtovalskega prostora. Torej mora biti opis

sistema narejen na način, ki bo predstavljal izhodišče samodejnega ovrednotenja različnih točk načrtovalskega prostora (podrobneje v poglavju 4.7). Naloga načrtovalca je, da s pomočjo okolja za podporo SNSPO pripravi opis arhitekture in funkcionalnosti ter ju med seboj poveže na način, da bo pridobil model, ki bo ustrezal podanim izhodiščnim zahtevam elektronskega sistema. Pri tem postopku so mu v pomoč mehanizmi ovrednotenja posamezne rešitve in optimizacijski postopki iskanja dovolj dobre rešitve.

Ker je potrebno opis sistema enovito vklopiti v celoten načrtovalski potek, tj. z ostalimi koncepti obvladovanja kompleksnosti, mora biti delitev opisa sistema narejena skladno z določenimi pravili. V načrtovalskem poteku, ki smo ga postavili, smo za to poskrbeli tako, da smo pripravili knjižnico okvirjev, ki predstavljajo osnovo za zajemanje opisa sistema. V luči predhodno omenjene ortogonalizacije opisa sistema si lahko te okvirje predstavljamo kakor koordinatni sistem, v katerem moramo opisati sistem, ki ga razvijamo. Okvirji so "prazne" komponente, ki so prirejene opisu arhitekture in funkcionalnosti. Načrtovalec je z uporabo okvirjev voden čez postopek opisa sistema, ne da bi mu bilo npr. potrebno skrbeti za podporo za avtomatizacijo raziskovanja načrtovalskega prostora, ker je le-ta že vgrajena v okvirje same. Vlogo in izvedbo okvirjev bomo podrobneje predstavili v poglavju 4.5.

## *4.3 Komponentno načrtovanje*

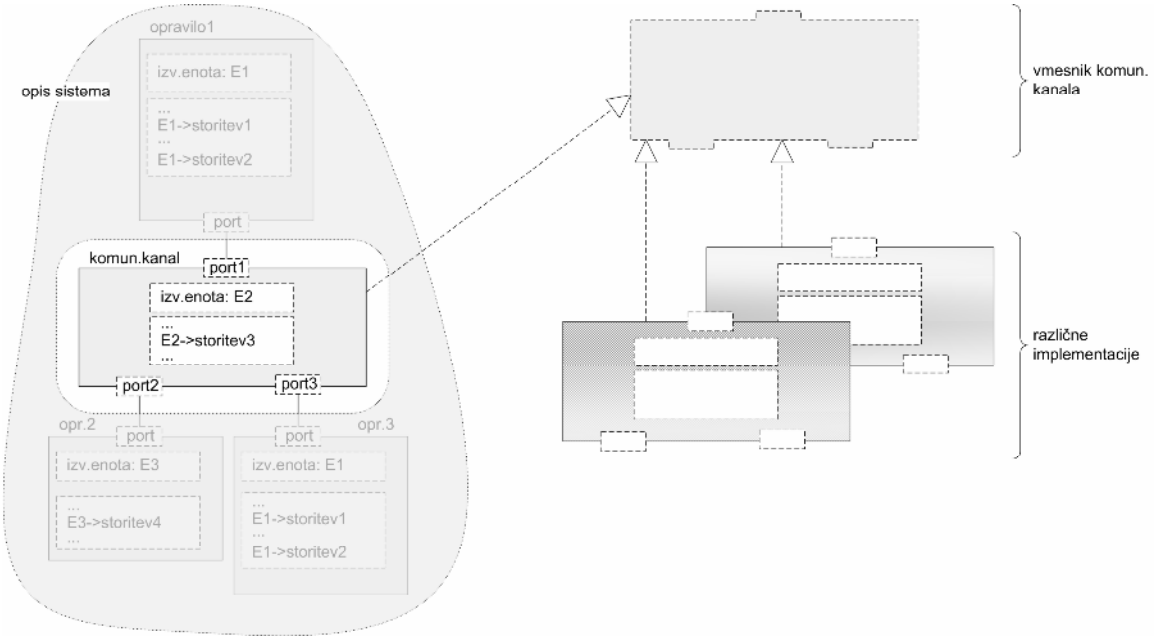
Pod izrazom komponentno načrtovanje (*component based design*) lahko najdemo v literaturi različne razlage. Definicija se od metodologije do metodologije razlikuje, saj si jo vsaka raziskovalna skupina prikroji za svoje potrebe. Ker za têrmin ni nekega splošnega pravila, se je potrebno za razumevanje povezovanja komponent in njihovo vlogo prepričati pri vsaki metodologiji posebej. Naša definicija komponentnega načrtovanja se bo zato lahko razlikovala od definicije, ki jo bralec lahko najde v povezavi s katero drugo metodologijo.

*Def. 4* **Komponenta** predstavlja zaključen skupek informacij o podsklopu določene domene (zornega kota) in strukturno omogoča hierarhičen opis sistema.

Pri naši metodologiji SNSPO uporabljamo razdelitev opisa na komponente z namenom hierarhične delitve obsežnega in kompleksnega opisa sistema na več strukturno ter pomensko zaključenih sklopov, katere je lažje obravnavati. Če je razdelitev na podsklope narejena smiselno, predstavlja to tudi osnovo za ponovno uporabo. Pri kasnejših nalogah lahko npr. ponovno uporabimo podsklope sistema, ki smo jih zgradili predhodno. Pri tem je pomembno, da komponente sistema predstavimo na smotrn način, saj vsaka delitev zahteva funkcionalni

dodatek, ki je izključno namenjen povezovanju. Preobsežne komponente lahko npr. pomenijo, da bo ob ponovni uporabi uporabljen samo določen del vse razpoložljive funkcionalnosti, preveč drobna delitev pa lahko privede do presežka funkcionalnih dodatkov.

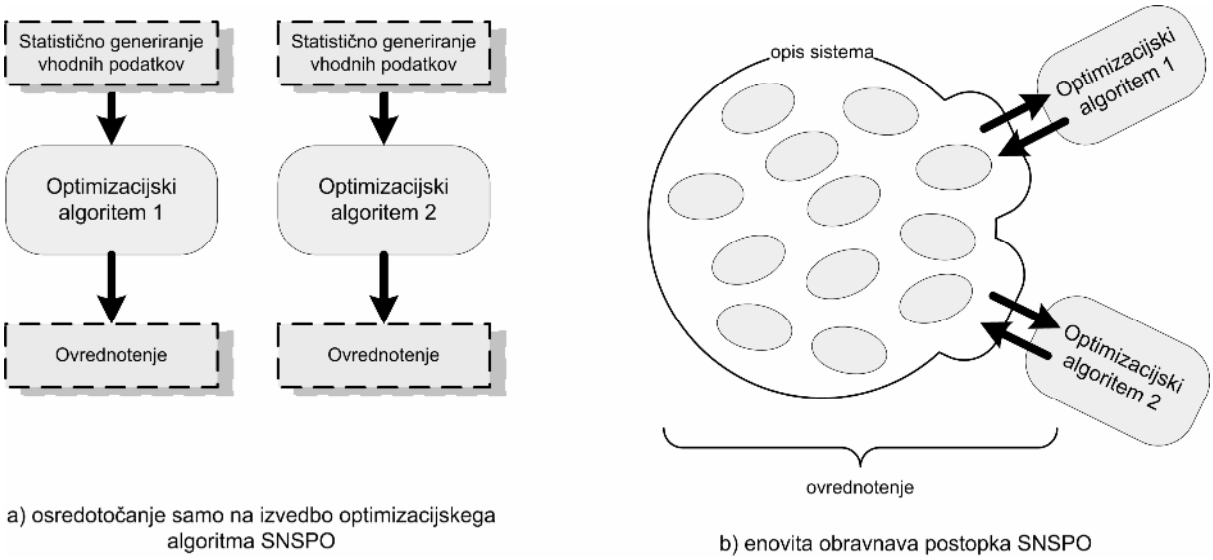
Metodologije komponentno načrtovanje pogosto uporabljajo za opis funkcionalnosti, npr. kot je to pri modeliranju TLM [p10], [k5.a]. Pri modeliranju TLM je funkcionalni opis sistema ločen na sklope, ki opišejo izračunavanje in komunikacijo, in to je tudi pristop, ki smo ga uporabili pri naši metodologiji. To pomeni, da je izvedba izračunavanja komponente (kako komponenta opravi nalogo, ki jo ima v sistemu) ločeno opisana od izvedbe komunikacije (kako komponenta komunicira s preostalimi komponentami v sistemu). To je še zlasti pomembno za raziskovanje načrtovalskega prostora, saj je tako mogoče hitreje preveriti vpliv različnih izvedb določenih podfunkcij sistema na lastnosti celotnega sistema. Načrtovalec se tako lahko ločeno osredotoči na opis funkcionalnosti in komunikacije, ki jih je ravno zaradi te iste ločitve mogoče tudi lažje ponovno uporabiti.




Slika 8 – Primer različnih implementacij istega vmesnika

Funkcionalnost sistema je opisana z mrežo medsebojno komunicirajočih komponent, ki so povezane s pomočjo komunikacijskih kanalov. V okviru naše metodologije so komunikacijski kanali tudi komponente, od ostalih pa se ločijo predvsem po primarni vlogi, ki jo imajo v sistemu (tj. služijo komunikaciji). Komponente so določene z vmesniki. Vmesnik določa, kako je komponenta videna s strani sistema, ne določa pa tega, kako je implementirana. To pomeni, da lahko posamezne komponente znotraj njih samih implementiramo različno in jih

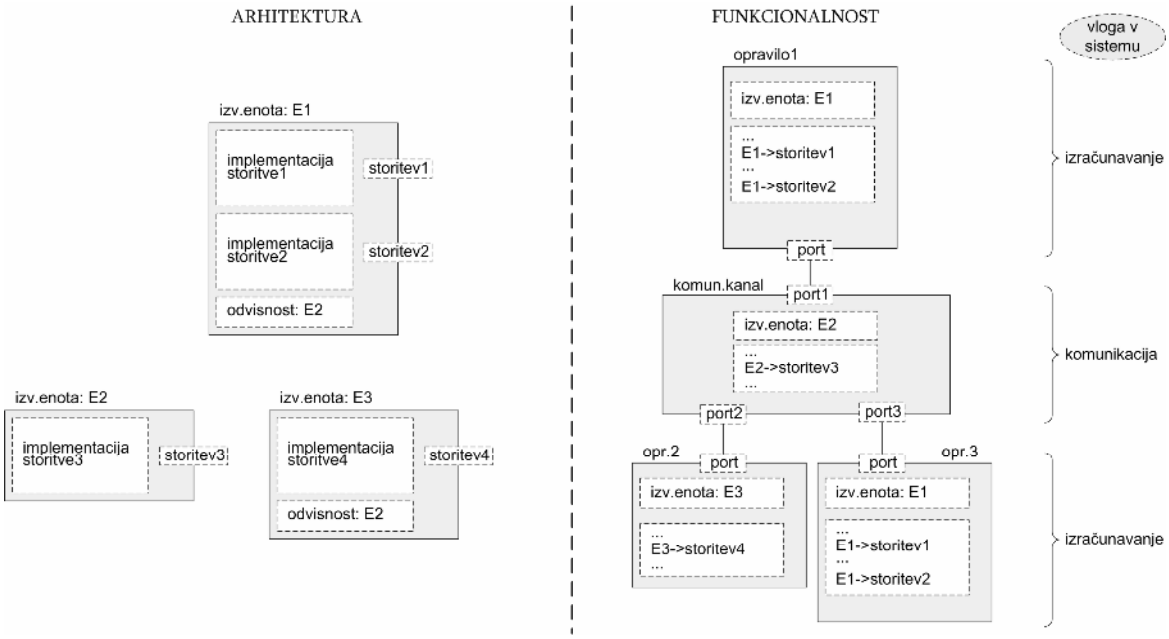
vklopimo v sistem, ne da bi bilo potrebno preostali del sistema zaradi tega kakorkoli spreminjati (slika 8). Različne izvedbe komponent se med seboj razlikujejo s stališča parametrov, ki so pomembni pri ovrednotenju celotnega sistema; zanimajo nas npr. čas izvajanja določenega algoritma, porabljena energija, zasedenost sistemskih sredstev ipd. Skladnost vmesnikov zamenljivih komponent predstavlja osnovo za samodejno raziskovanje načrtovalskega prostora. V tem primeru je ovrednotenje različnih izvedb sistema (z različnimi implementacijami komponent) samo stvar avtomatizacije postopka vklopa novega opisa v sistem in ovrednotenja celotnega sistema. Iskanje najoptimalnejše izvedbe sistema pa sodi že v domeno optimizacijskih algoritmov, ki jih bomo predstavili v poglavju 4.7.




Slika 9 – Dva različna načina razvoja optimizacijskih algoritmov SNSPO

Delitev na komponente predstavlja potreben pogoj za samodejno apliciranje optimizacijskih algoritmov SNSPO, da pa bo to tudi praktično izvedljivo, je potrebno opis komponent sistema narediti tako, da bo te algoritme nad zajetimi komponentami sistema tudi mogoče praktično uporabiti. Slika 9 prikazuje dva različna načina razvoja optimizacijskih algoritmov SNSPO. V prvem primeru (a) se vhodni podatki ustvarijo umetno in tako se ovrednotijo tudi rezultati. V drugem primeru (b) pa algoritmi pridobijo ustrezne informacije o sistemu iz enovitega opisa vseh informacij o sistemu in rezultate vrnejo v obliki, ki lahko vpliva na povezavo komponent v sistemu. Naš pristop načrtovanja SNSPO se ne osredotoča samo na izvedbo algoritma, ampak ne celoten sistem. Tako nas ne zanima, kako uspešen je posamezen algoritem npr. časovne dodelitve, ampak kako se to izraža v uspešnosti celotnega sistema.

Podobno kot pri opisu funkcionalnosti, uporabljamo v okviru naše metodologije komponentni opis tudi pri opisu arhitekture. To pomeni, da je SO predstavljena z mrežo medsebojno povezanih (arhitekturnih) komponent sistema, ki predstavljajo posamezne smiselne podsklope SO (npr. procesor, vodilo, namensko obdelovalne enote ipd.). Tudi tu je pomembno, da je razdelitev komponent smiselna, da je njihovo združevanje v celoten sistem čimbolj prilagodljivo (s podporo za ponovno uporabo) in učinkovito.




Slika 10 – Delitev na arhitekturo in funkcionalnost ter komponentni opis domen

Slika 10 prikazuje primer uporabe dveh do sedaj predstavljeni konceptov obvladovanja kompleksnosti, tj. delitev opisa sistema na arhitekturni in funkcionalni del ter komponentno načrtovanje sistema. To sliko bomo kasneje uporabili ponovno pri podrobnejšem opisu okvirjev (poglavje 4.5), kjer bodo tudi podrobneje razloženi njeni posamezni deli. Vloga okvirjev je tudi tu pomembna; ti nudijo podporo za ponavljajoče se podporne dele opisa sistema, nudijo mehanizme povezovanja na osnovi vmesnikov in omogočajo učinkovito implementacijo uporabljenih vmesnikov. V okviru naše metodologije smo raziskovali predloge okvirjev, ki bi nudili učinkovito gradnjo komponent v obeh domenah. Če strnemo najpomembnejše prednosti komponentnega načrtovanja, so to: ponovna uporaba podsklopov, obvladovanje kompleksnosti sistema in avtomatizacija raziskovanja načrtovalskega prostora.

## *4.4 Abstrakcija*

Abstrakcija predstavlja enega izmed splošno priznanih mehanizmov obvladovanja kompleksnosti [p10], [p27], [k5.a], [k12.a] in nudi osnovo za hitrejši ter učinkovitejši razvoj

elektronskih sistemov. Uporabiti jo je mogoče na različnih nivojih izvedbe sistema, od začetne sistemske specifikacije pa vse do končne implementacije. Pri postopku SNSPO je uporaba koncepta abstrakcije še zlasti pomembna, saj omogoča gradnjo (abstraktnih) modelov sistema, na podlagi katerih je mogoče pridobiti povratne informacije o ustreznosti modela, kljub temu da implementacijske podrobnosti še niso znane. Pred razmahom SNSPO je bilo v poteku načrtovanja elektronskih sistemov običajno, da so bile pomembne sistemske odločitve sprejete neosnovano, npr. po intuitivni odločitvi sistemskega načrtovalca. Običajno je bilo npr., da je bila strojna platforma (SO) pripravljena vnaprej, PO pa se je nanjo prilagodilo kasneje. Če k temu dodamo še želeno ponovno uporabo SO in z njo povezane presežke, je jasno, da tako ne moremo doseči optimalne implementacije sistema – bodisi glede izrabe sistemskih sredstev, hitrosti/zmogljivosti, porabe energije ipd. Pri tem se razume, da je slika optimalne implementacije od sistema do sistema različna in odvisna predvsem od načrtovalskih kompromisov. Razloge za optimalno implementacijo smo predstavili že v poglavju 2.

Podrobnega študija abstrakcije smo se lotili zato, ker predstavlja osnovo za ovrednotenje visokonivojskega opisa sistema, na podlagi katerega se je mogoče izogniti prezgodnjim in neosnovanim sistemskim načrtovalskim odločitvam. Na podlagi povratnih informacij ovrednotenja modela lahko ugotovimo, kateri modeli sistema izpolnjujejo načrtovalske zahteve in kateri ne. Modele sistema, ki ne izpolnjujejo zahtev, se zavrže, vloga abstrakcije pa je ta, da omogoča zanesljivo ovrednotenje modela sistema že v zgodnejših fazah načrtovalskega poteka. Tako je mogoče večji del načrtovalskega napora usmeriti v raziskovanje tistega dela načrtovalskega prostora, v katerem se nahajajo potencialni modeli sistema. Ker so modeli na različnih nivojih abstrakcije različno podrobno predstavljeni, se to kaže tudi v različni količini in tipu podatkov, ki jih je potrebno pri ovrednotenju modela upoštevati. To posledično vpliva na hitrost ovrednotenja sistema in istočasno tudi na količino truda, potrebnega za izvedbo sistemskih sprememb. Podrobnejši ko je model sistema, težje ga je ovrednotiti s stališča njegovih visokonivojskih lastnosti in zahtevnejše so spremembe na sistemskem nivoju.

Nižanje nivoja abstrakcije temelji na načrtovalskih odločitvah glede sistema in če kompleksnost teh odločitev ni prevelika ter so nivoji abstrakcije enoumno določeni, je mogoče nižanje nivoja abstrakcije avtomatizirati. Definirani nivoji abstrakcije in nedvoumno določeni prehodi med njimi neposredno pripomorejo k avtomatizaciji nižanja nivoja

abstrakcije [p27], [p28]. Npr. dandanes je samoumevno samodejno nižanje nivoja abstrakcije iz programskega jezika v binarno kodo. Postopek avtomatizacije omogočajo prevajalniki, načrtovalske odločitve pa so mogoče z uporabo nastavitev prevajalnika.

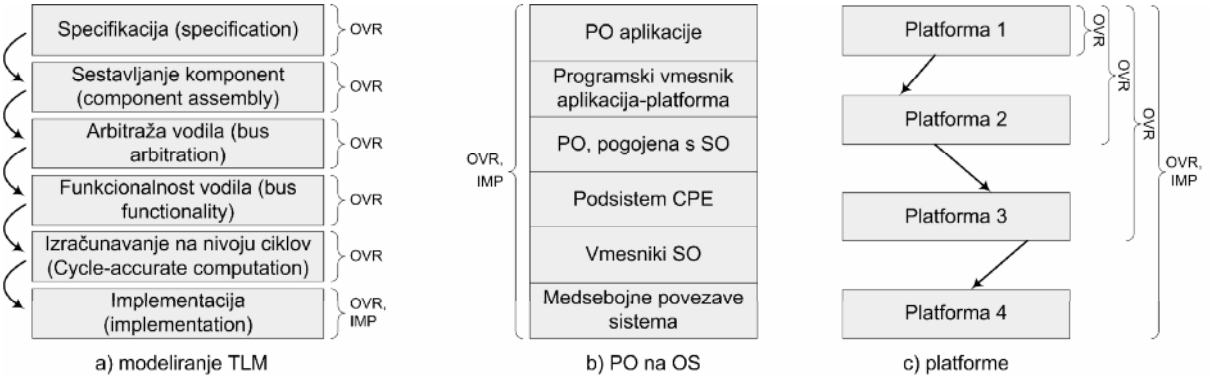
Poleg nedvoumno določenih nivojev abstrakcije z ne prevelikimi razlikami med nivoji pa je za učinkovito uporabo abstrakcije pomembno tudi to, da je najvišji nivo abstrakcije, na kateri je že možen formalni opis sistema, dovolj blizu začetni specifikaciji sistema. Začetni opis sistema, ki je narejen s pomočjo skic, razpredelnic, pogovornega opisa ipd., je namreč neformalen, saj ni enolično določen in ga tudi ni mogoče enolično ovrednotiti. Govorimo o razkoraku med začetnim neformalnim opisom sistema in prvim formalnim opisom sistema, ki je izvedljiv in ga je tako ali drugače mogoče ovrednotiti. V nadaljevanju bomo predlagali višje nivoje abstrakcije, ki bodo omogočali tako višjenivojski opis sistema kot tudi hitro in enostavno ovrednotenje na teh nivojih.

### *4.4.1 Klasifikacija abstrakcije*

V zadnjih letih je bilo objavljenih zelo veliko raziskovalnih rezultatov s področja SNSPO, ki se osredotočajo na prostorsko in časovno dodeljevanje, problem heterogenega opisa sistema ipd. [rs1]–[rs7]. Čeprav vsi ti problemi še zdaleč niso do konca razrešeni, se raziskovalni interes postopoma premika iz področja obvladovanja meje med SO in PO proti področju obvladovanja višjih nivojev abstrakcije. V kombinaciji z ostalimi koncepti obvladovanja kompleksnosti je to namreč pristop, ki bo pripomogel k zahtevani vse večji produktivnosti. V nadaljevanju bomo predstavili tri glavne tipe abstrakcije in tudi podali primere metodologij, ki jih uporabljajo. Na podlagi teh tipov abstrakcij smo klasificirali sorodne metodologije (glede na vertikalne koncepte obvladovanja kompleksnosti), s katerimi smo se srečali pri našem raziskovanju.

Prvi tip abstrakcije (slika 11 a) temelji na pristopu, kjer se opisovanja sistema lotimo z opisom samo najpomembnejših sistemskih lastnosti (specifikacija). Pri modelu na obstoječem nivoju abstrakcije so značilnosti, ki (še) niso pomembne ali pa (še) niso znane, enostavno izpuščene. Med postopnim dopolnjevanjem modela sistema se na podlagi povratnih informacij ovrednotenj načrtovalčevo znanje o sistemu dopolnjuje, zato lahko opis sistema postopoma dopolnjuje z novimi (podrobnejšimi) značilnostmi sistema (sestavljanje komponent, arbitraža in funkcionalnost vodila ter izračunavanje na nivoju ciklov). Postopek postopnega dopolnjevanja modela istočasno pomeni tudi nižaje nivoja abstrakcije (slika 11 a, puščice nakazujejo nižanje nivoja abstrakcije v smeri navzdol). Postopek dopolnjevanja je

zaključen, ko je nivo abstrakcije modela sistema dovolj nizek – torej so zajete vse potrebne informacije, na podlagi katerih je implementacija sistema mogoča z uporabo avtomatizacijskih orodij. Ovrednotenje modela sistema (OVR) je mogoče na vsakem nivoju abstrakcije, najnižji nivo abstrakcije pa poleg ovrednotenja nudi še implementacijo (IMP). Ta model predstavlja najnižji nivo abstrakcije, ki je za SNSPO še zanimiv, in vsebuje vse podatke, ki so bili zajeti v procesu postopnega dopolnjevanja sistema. Ta tip abstrakcije se danes največ uporablja pri tipu modeliranja na nivoju transakcij (*transaction level modeling,* TLM), ki so ga predstavili Gajski in ostali v [p10] (pristop je lepo razložen tudi v [k5.a]).




Slika 11 – Premeri treh tipov abstrakcij[15]

Drugi tip abstrakcije temelji na konceptu nivojev funkcionalnosti (Slika 11 b). Sosednji nivoji funkcionalnosti so povezani s pomočjo vmesnikov, ti pa poleg povezovanja skrbijo tudi za možnost učinkovitega skrivanja implementacije, vgrajene znotraj določenega nivoja. Z uporabo tega koncepta se raziskovalec lahko ožje osredotoči na problem in se mu ni potrebno obremenjevati z ostalimi nivoji implementacije sistema. Gledano od zgoraj navzdol je ključna prednost uporabe tega pristopa v tem, da ni potrebno skrbeti za spodaj ležeče implementacijske podrobnosti, ki se jih enostavno uporabi preko vmesnikov. Možnost ponovne uporabe in modularne zgradbe je večja, če so vmesniki standardizirani, kar sicer vpliva na zmanjšanje učinkovitosti [p1], istočasno pa s tem pridobimo širši nabor implementacijskih možnosti. Ta tip abstrakcije uporabljajo programski inženirji za obvladovanje kompleksnosti aplikacij, ki temeljijo na OS (PO aplikacije, gonilniki, OS ...). Od predhodno omenjenega tipa abstrakcije se razlikuje v tem, da je tu za pridobitev modela za ovrednotenje potrebno v celoti implementirati vse nivoje funkcionalnosti, saj so drug od drugega soodvisni (Slika 11 b, ovrednotenje je mogoče samo na podlagi vseh nivojev).

[15] Okrajšavi OVR in IMP pomenita ovrednotenje ter implementacija.

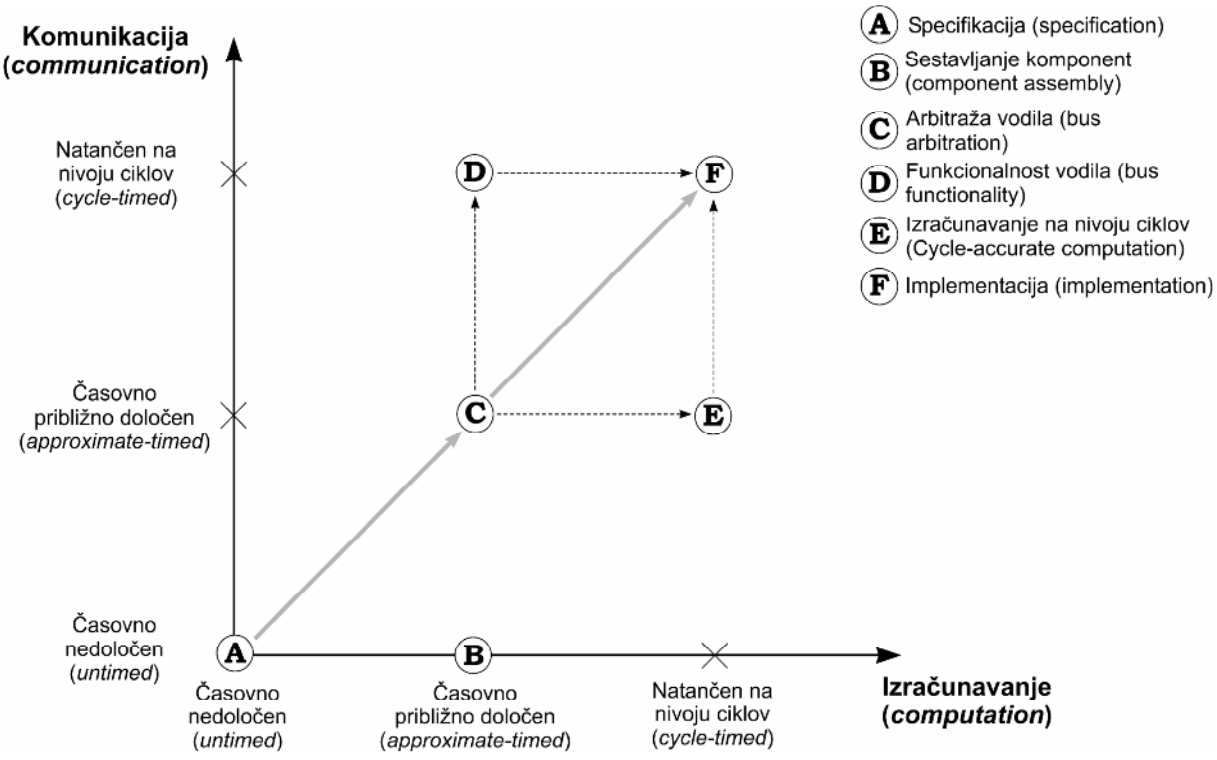
Čeprav je v literaturi mogoče zaslediti pristope, ki omogočajo profiliranje na nivoju aplikacije (npr. [p9]), je profiliranje še vedno odvisno od informacij, ki izvirajo iz nižjih nivojev. Prof. Jerraya in ostali [rs1] so postavili metodologija SNSPO, ki temelji na tem tipu abstrakcije – njihovo delo temelji na uporabi vmesnih adapterjev med nivoji funkcionalnosti [p1], [p15], [k12.b]. Metodologija SPACE [rs7] temelji na konceptu nivojev funkcionalnosti z implementacijo podpore za funkcije OS znotraj simulacijskega ogrodja SystemC [p8].

Tretji tip abstrakcije temelji na konceptu platform, ki predstavljajo nivoje abstrakcije (Slika 11 c). Načrtovalčeva naloga je, da na vsakem nivoju abstrakcije čim bolje poveže podane zahteve z abstraktno predstavitvijo potencialne implementacije. Izbira potencialne implementacije trenutnega nivoja abstrakcije vpelje specifikacijo zahtev za naslednji, spodaj ležeči, nivo abstrakcije. Podobno kot pri prvem tipu abstrakcije, se opis sistema prične z zajemom lastnosti na sistemskem nivoju, razlikuje pa se v tem, da se tu ne izboljšuje opisa posameznega nivoja abstrakcije, temveč se opisi na različnih nivojih abstrakcije med seboj dopolnjujejo (z različnimi nivoji implementacijske podrobnosti). V primerjavi s tipom abstrakcije, ki temelji na konceptu nivojev funkcionalnosti (Slika 11 b), je pri temu tipu abstrakcije ovrednotenje modela sistema (OVR) mogoče na vsakem nivoju abstrakcije, nivoji abstrakcije pa tu niso tako natančno določeni. Primer načrtovalske metodologije, ki uporablja ta konceptu abstrakcije, je Metropolis [rs2.c], ki temelji na rekurzivni paradigmi načrtovanja na osnovi platform [p4]. Več informacij o načrtovanja na osnovi platform lahko bralec najde v [p29].

Podporo za zgodnje raziskovanje na sistemskem nivoju, ko implementacijske podrobnosti še niso znane/zajete, nudita samo prvi in tretji tip abstrakcije. Pri drugemu tipu abstrakcije so za ovrednotenje modela potrebni vsi nivoji funkcionalnosti in posledično povratnih informacij samo na podlagi visokonivojskega opisa ni mogoče pridobiti. Glede na priljubljenost uporabe abstrakcije, ki temelji na modeliranju TLM, in široki razpolagi podpornih modelirnih ogrodij (npr. SystemC [ss1] in SpecC [ss2]) smo se v našem raziskovanju osredotočili na prvi tip abstrakcije (tj. modeliranje TLM). V naslednjem poglavju (4.4.2) bomo podrobneje predstavili modeliranje TLM in naš predlog za izboljšanje mehanizmov modeliranja TLM. Razloge, zakaj je izboljšava potrebna, bomo podprli z razlago modeliranja TLM na osnovi metamodela Rugby (poglavje 4.4.2.1), iz te razlage pa bo tudi jasno, na katerih področjih modeliranja je potrebna izboljšava. Sledila bo predstavitev predloga razširitve z višjimi nivoji abstrakcije, ki bodo postavili temelje za učinkovito raziskovanje na sistemskem nivoju.

### *4.4.2 Modeliranje TLM*

V postopku SNSPO stremimo za tem, da se izognemo obvezujočim ad hoc odločitvam na sistemskem nivoju, saj neosnovano ožijo razpoložljiv načrtovalski prostor in izključujejo potencialno boljše načrtovalske odločitve. Čeprav pri modeliranju TLM začetni model sistema (Slika 12, točka A – specifikacija, v literaturi [k5.a] označen tudi kot *system architecture model, SAM*) zajame komunikacijo na abstrakten in nepopoln način ter ima komunikacijo in izračunavanje časovno nedoločeno, je ta model že na začetku predstavljen v funkcionalno popolni obliki. Funkcionalno popolni zato, ker je uveljavljena načrtovalska praksa, da se začetni model sistema opiše s programskim jezikom (npr. C ali C++) na način, ki zajame polno funkcionalnost modela. Ovrednotenje visokonivojskih načrtovalskih odločitev in profiliranje, na podlagi katerih so vodene nadaljnje načrtovalske odločitve, je mogoče šele od opisa tega modela naprej. S sistemskega stališča predstavlja takšen začetni opis algoritma pomanjkljivost tega pristopa, saj ni dovolj abstrakten. Raziskovanje na nivoju sistema je tako oteženo, saj je za kakršno koli ovrednotenje, tudi sistemsko, potrebna vključitev nizkonivojskega opisa.




Slika 12 – Modeliranje TLM [p10]

Rezultat popolnega funkcionalnega opisa je izvršljivi model sistema, ki ima značilnosti programske implementacije, tj. ima zaporedno zasnovo izvajanja operacij nad podatki, ki se

nahajajo v osrednjem spominu. To pa je ravno v nasprotju s SO, katere prednost je sočasno obdelava porazdeljenih podatkov. Algoritmi z zaporednim potekom izvajanja, ki veljajo za učinkovite v izvedbi PO, so redko enako učinkoviti v izvedbi SO. Kot je že bilo izpostavljeno v literaturi [p7], se algoritmi, namenjeni za SO, bistveno razlikujejo od algoritmov, namenjenih za PO predvsem zaradi izrabe različnih tipov implementacijskih sredstev. Nadaljnje načrtovalske odločitve, ki so osnovane na profiliranju programskega opisa [p9], [p11], ne morejo nuditi zadostnih informacij za optimalen potek SNSPO. V postopku profiliranja se odkrijejo ozka grla algoritma (z zaporednim potekom izvajanja) in ustaljena načrtovalska praksa je, da se te računsko intenzivne dele premesti v SO. Tako se sredstva SO, ki so temeljno sočasna, uporabijo skladno z modelom PO, ki je temeljno zaporeden. Na podlagi te ideje skoraj zagotovo ne moremo doseči optimalne izvedbe sistema. Profiliranje kode PO in odkrivanje implementacijskih ozkih grl si namreč lahko razlagamo kot hitro in poceni kompenzacijo začetnih ad hoc načrtovalskih odločitev.

#### *4.4.2.1 Modeliranje TLM, razloženo na podlagi metamodela Rugby*

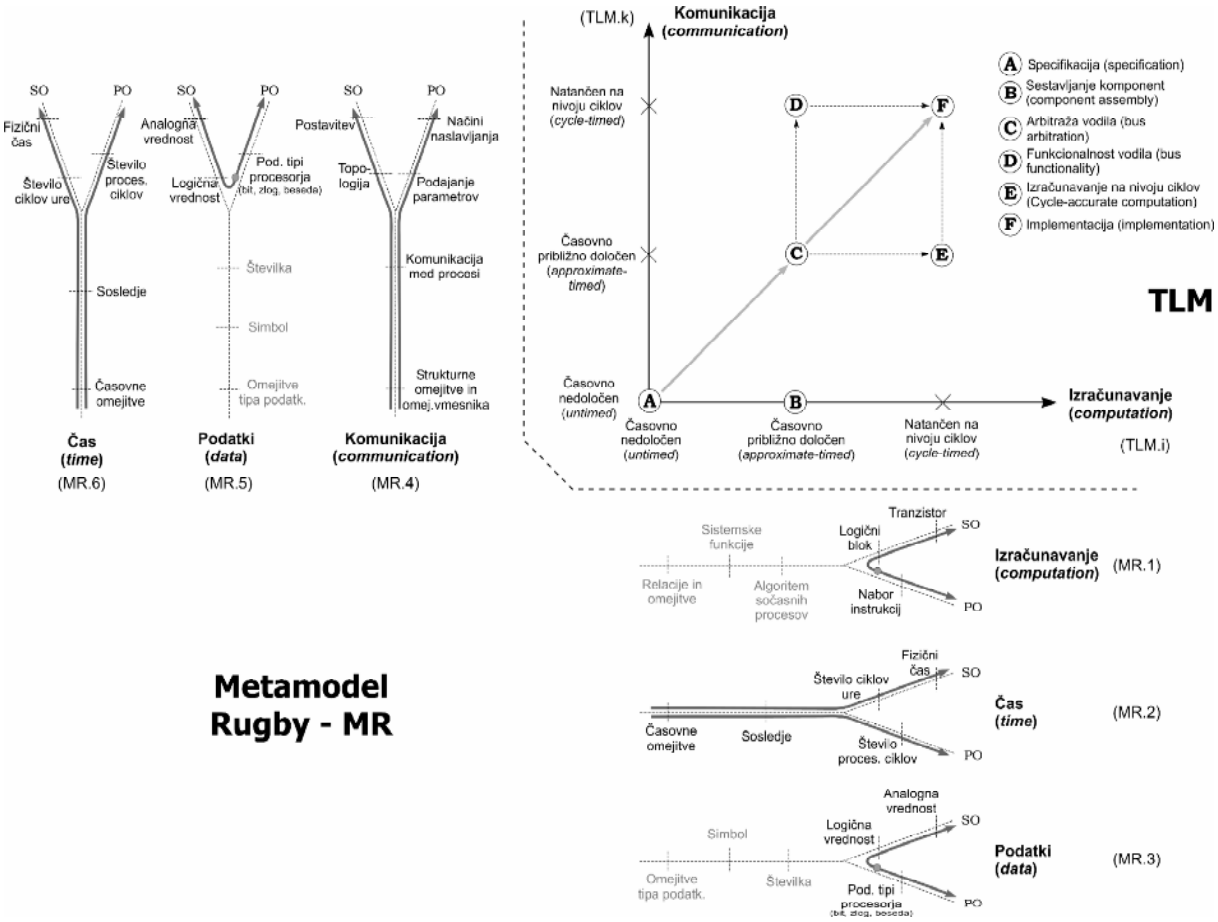
Kot smo že omenili, se pri modeliranju TLM najprej zgradi funkcionalno popoln model sistema (Slika 13, točka A). Na tem nivoju abstrakcije model služi za potrditev funkcionalne pravilnosti opisanega algoritma. V nadaljevanju načrtovanja sistema je model nadgrajen z opisom časovnih podrobnosti in opisom arhitekturno izvedljive komunikacije. Namen postopnega dopolnjevanja modela sistema je vodenje načrtovalca od začetne specifikacije sistema (točka A) preko jasno določenih vmesnih korakov načrtovanja (nivojev abstrakcije, točke B–E) do končnega implementacijskega modela (točka F). S sprejemanjem načrtovalskih odločitev in dopolnjevanjem opisa sistema tako načrtovalec zamenjuje bolj abstrakten model sistema z manj abstraktnim vse do točke, ko opis lahko uporabi za samodejno izvedbo sistema. V zadnjem koraku modeliranja je zgrajen funkcionalno, časovno in komunikacijsko v celoti določen model (model RTL, točka F). Ker ta model vsebuje dovolj informacij za dokončno izvedbo sistema s pomočjo avtomatizacijskih orodij, predstavlja implementacijski model (*implementation model*).

Da bi razložili mehanizme nižanja nivoja abstrakcije, ki se skrivajo za postopnim dopolnjevanjem, bomo modeliranje TLM predstavili na osnovi metamodela Rugby (MR). MR so predstavili Jantsch in ostali v [p27] ter v razširjeni izvedbi še v [p28] za namene raziskovanja modeliranja in pristopa k obravnavi abstrakcije na bolj osnoven način. Pri MR je

modeliranje sistema razdeljeno v štiri domene (aspekte): komunikacijo (*communication*), izračunavanje (*computation*), podatke (*data*) in čas (*time*).

*Def. 5* **Domena** je določen pogleda na model, ki ga je mogoče logično analizirati ločeno od ostalih pogledov.[p27]

V vsaki od teh domen obstaja več nivojev abstrakcije, ki obsegajo nivoje med začetno idejo (Slika 13, skrajne leve točke vodoravnih domen (MR.1–3) in najnižje točke navpičnih domen (MR.4–6)) in končnim opisom modela SO ali PO, ki nudi dovolj informacij za implementacijo elektronskega sistema (skrajno desne točke vodoravnih domene, vrhnje točke navpičnih domen).




Slika 13 – Modeliranje TLM, razloženo na osnovi metamodela Rugby

Slika 13 prikazuje povezavo med obema modeloma, ki smo jo naredili na osnovi pregleda postopkov dopolnjevanja in principov modeliranja TLM. Izhodišče primerjave je naslednje: modeliranje TLM loči dve domeni (izračunavanje in komunikacijo), MR pa štiri domene (izračunavanje, komunikacijo, čas in podatke), ki so osnovnejše. Nivoji abstrakcije v modeliranju TLM so določeni s kombinacijam podanih dveh domen (Slika 13, zgoraj desno).

Ugotovili smo, da v modeliranju TLM domena izračunavanja zajame v MR domene izračunavanja, časa in podatkov ter da v modeliranju TLM domena komunikacije zajame v MR domene komunikacije, časa in podatkov. Termina *izračunavanje* (TLM.i) in *komunikacija* (TLM.k), uporabljena pri modeliranju TLM, tako nista enaka istoimenskima terminoma, uporabljenima v MR (MR.1 in MR.4). Za lažje razumevanje in v izogib zmešnjavi smo pri opisih dodali oznake v oklepajih.

#### *4.4.2.1.1 Modeliranje TLM in domena izračunavanja*

Domeno *izračunavanja* modeliranja TLM (TLM.i) lahko predstavimo na osnovi domen izračunavanja, časa in podatkov MR (MR.1–3). Nižanje nivoja abstrakcije domene *izračunavanja* (TLM.i) od začetne specifikacije do končne implementacije pomeni dopolnjevanje modela izračunavanja. Pri tem raziskani nivoji abstrakcije, ovrednoteni s stališča MR, so po posameznih domenah (MR.1–3) prikazani pod osjo *izračunavanja* (TLM.i) – puščice prikazujejo nižanje nivoja abstrakcije, prikazano s stališča MR. Pri primerjavi nivojev abstrakcije, ugotovimo, da so razpoložljivi nivoji abstrakcije izkoriščeni samo v časovni domeni (MR.2). Razpoložljivi nivoji abstrakcije v preostalih dveh domenah, tj. domeni izračunavanja (MR.1) in podatkov (MR.3), pri modeliranju TLM niso v celoti izkoriščeni. Metodologije, ki ponujajo formalno podporo od začetnega modela TLM (*specification model*) naprej, omogočajo, gledano s perspektive MR, širšo uporabo nivojev abstrakcije torej samo v časovni domeni (MR.2). Začetni opis sistema namreč ne vsebuje nobenih informacij o trajanju izračunavanj, zanj pa je pomembno samo pravilno sosledje dogodkov – to implicira povsem abstraktno časovno domeno MR (MR.2). Tekom postopka dopolnjevanja modela se časi izračunavanja postopoma dopolnjujejo in na koncu tvorijo časovno povsem natančen model sistema – tako so nivoji abstrakcije v časovni domeni raziskani.

V nasprotju s pravkar razloženo časovno domeno pa je (pri TLM.i) uporabljen nivo abstrakcije v domenah izračunavanja (MR.1) in podatkov (MR.3) na razmeroma nizki stopnji. Razložili smo že, da je začetni model modeliranja TLM funkcionalno kompleten, saj so oblike podatkov (MR.3) in algoritem za njihovo obdelavo (MR.1) v končni obliki. Omogočeno je takojšnje funkcionalno preverjanje s pomočjo testnih vektorjev, vsa nadaljnja dopolnjevanja modelov pa se ukvarjajo s časovno določitvijo in komunikacijo. Na sliki 13 je začetni nivo abstrakcije v domenah izračunavanja in podatkov MR prikazan s piko na programski strani abstrakcije, to ustreza začetnemu programskemu modelu pri modeliranju TLM. V primeru, da

je potrebna premestitev izračunavanja dela algoritma v SO, je to problem transformacije na istem nivoju abstrakcije, le da preko heterogene meje. Za izvedbo preslikave se uporabljajo različni pristopi: s pomočjo ročne pretvorbe kode, z uporabo podsklopa enega izmed jezikov za raziskovanju na sistemskem nivoju, ki ga je mogoče prevesti v obliko, primerno za opis SO, in s pomočjo samodejne transformacije programskega opisa v strojni opis, kot je to pri metodologiji SPARK [p16], [rs4] ali v primeru nevsiljive obravnave na SO in PO [p7].

#### *4.4.2.1.2 Modeliranje TLM in domena komunikacije*

Podobno kakor domeno izračunavanja lahko tudi domeno *komunikacije* modeliranja TLM (TLM.k) predstavimo na osnovi domen komunikacije, podatkov in časa (MR.4–6). Tu so razpoložljivi nivoji abstrakcije v celoti izkoriščeni v domenah komunikacije (MR.4) in časa (MR.6). Podroben pregled mehanizmov komunikacije razkrije, da je v začetnem modelu sistema pri modeliranju TLM (specifikacija) komunikacija implementirana enostavno z mehanizmi skupnih spremenljivk in za ustrezno sosledje dodatno razširjena z *dogodki* [p10], [k5.a]. Tako predstavljena komunikacija očitno predstavlja visok nivo abstrakcije. Tekom postopnega dopolnjevanja se skupne spremenljivke zamenjajo s kanali in definirajo do stopnje, ko je poznan vsak krmilni signal povezave na komunikacijskem vodilu. Istočasno je v procesu dopolnjevanja poskrbljeno tudi za nižanje nivoja abstrakcije časovne domene. Ko je komunikacija predstavljena abstraktno z mehanizmom skupnih spremenljivk, je komunikacija sprva časovno nedoločena, tekom postopnega dopolnjevanja pa se zgradi implementacijski model RTL, ki je natančen na nivoju ciklov. Z upoštevanjem podatkovne domene MR (MR.3 in MR.5) in dejstva, da model sistema pri komuniciranju (TLM.k) uporablja podatke, ki so ustvarjeni v delu modela za izračunavanje (TLM.i), je izkoristek nivojev abstrakcije obeh podatkovnih domen enak (MR.3 in MR.5). Na sliki 13 je to prikazano z enakim izkoristkom nivojev abstrakcije podatkovnih domen (MR.3 in MR.5) pri obeh oseh izračunavanja (TLM.i) in komunikacije (TLM.k).

### *4.4.3 Uporaba višjih nivojev abstrakcije*

V prejšnjih poglavjih smo raziskali modeliranje TLM na podlagi uporabljenih nivojev abstrakcije in ugotovili, da je v posameznih domenah podpora za abstrakcijo, zlasti na višjih nivojih, nezadostna. Ugotovitve smo strnili v tabelo 1, ki prikazuje uporabo nivojev abstrakcije modeliranja TLM na osnovi MR. Nivoja abstrakcije izračunavanja in podatkov (gledano iz stališča MR) nista polno izkoriščena, zato bomo v nadaljevanju predstavili

predlog razširjenih nivojev abstrakcije in tudi ponudili programsko podporo za njihovo uporabo.

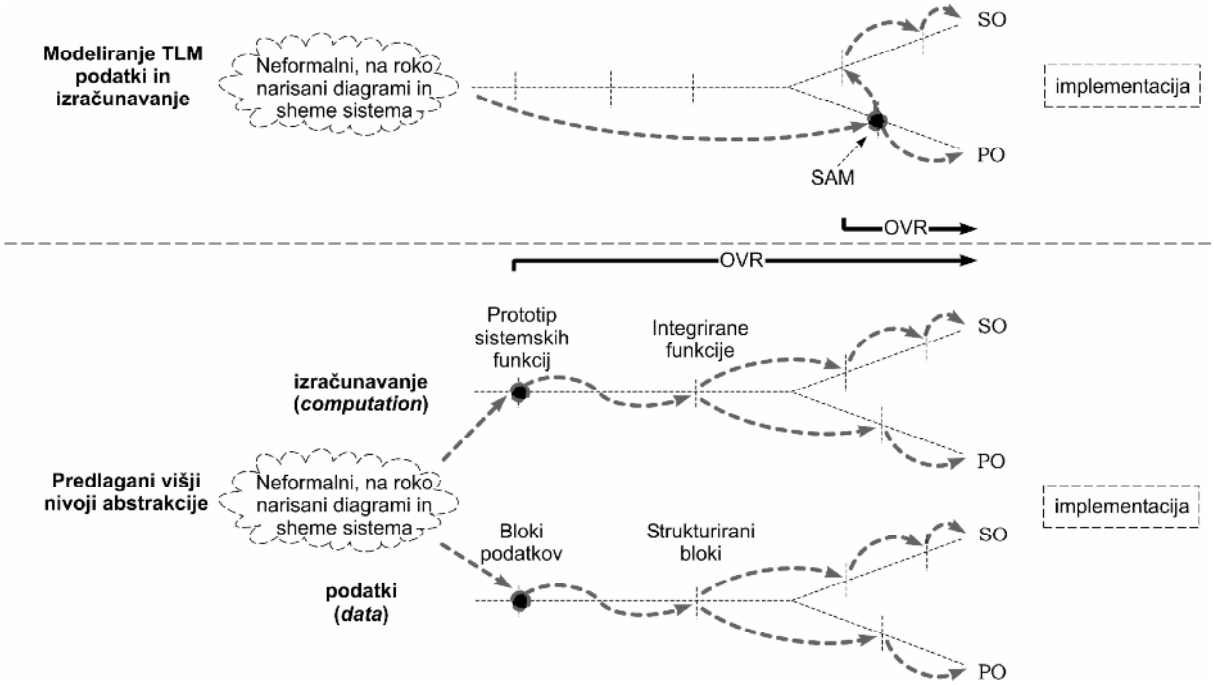
| **Modeliranje TLM** | **Metamodel Rugby** | | | |
|---|---|---|---|---|
| | Izračunavanje *(computation)* | Komunikacija *(communication)* | Podatki *(data)* | Čas *(time)* |
| Izračunavanje *(computation)* | delno | x | delno | popolno |
| Komunikacija *(communication)* | x | popolno | delno | popolno |

Tabela 1 – Raziskovanje nivojev abstrakcije modeliranja TLM, razloženo na osnovi metamodela Rugby

Tudi če pustimo inherentno podoptimalne implementacije sistema ob strani, je potreba po višjih nivojih abstrakcije še vedno prisotna, saj predstavlja način zmanjševanja konceptualnega razkoraka med (neformalnim) visokonivojskim opisom in prvim funkcionalnim opisom s programskim jezikom. Razkorak, ki nastane zaradi nezadostne podpore za višje nivoje abstrakcije v modeliranju TLM, je prikazan na zgornjem delu slike 14. Pri tem smo kot osnovo za prikaz nižanja nivojev abstrakcije uporabili način, ki je bralcu znan že iz MR. Razkorak je prisoten v domenah podatkov in izračunavanja (MR), v časovni domeni pa se višji nivoji abstrakcije učinkovito uporabljajo. Prehod od prve konceptualne zasnove modela do modela SAM, od koder je naprej mogoče model ovrednotiti, je v smislu vloženega truda, pomembnih sistemskih načrtovalskih odločitev in zajema zahtevanih podatkov zelo zahteven. S področjem nižanja nivoja abstrakcije in z visokonivojskim opisom sistema se ukvarja modelirni jezik UML [ss3], ki pa se v osnovi ukvarja samo s splošno paradigmo opisovanja modelov in sam po sebi ne nudi določenega načrtovalskega mehanizma. Pripomore lahko kvečjemu k učinkoviti sintaksi zajema parametrov sistema, ki ga želimo opisati. Trenutno v povezavi s SNSPO za jezik UML še ni dovolj orodij, ki bi pri nižanju nivoja abstrakcije omogočala učinkovito samodejno pretvorbo že zajetih informacij v obliko, primerno za nižji nivo abstrakcije. Manjka pa tudi še podpora za ovrednotenje zgrajenih modelov.

*Modeliranje sistema smo v okviru naše metodologije razširili na višje nivoje abstrakcije. Razširjen koncept abstrakcije sicer temelji na modeliranju TLM, a je nadgrajen z določenimi lastnostmi visokonivojskega modeliranja, ki ima značilnosti jezika UML. Spodnji del slike 14 prikazuje dodatne nivoje abstrakcije, ki jih predlagamo v domenah izračunavanja in podatkov. Z uporabo teh dodatnih nivojev je mogoče k razvoju sistema pristopiti sistematično in intuitivno, sitem je mogoče formalno obravnavati na višjih nivojih abstrakcije (ne samo od*

*funkcionalno kompletnega modela SAM naprej), razkorak med začetnim neformalnim opisom sistema in prvim formalnim modelom sistema je manjši, ovrednotenje načrtovalskih odločitev, ki so pripeljale do trenutnega modela sistema, pa je mogoče narediti v zgodnejši stopnji načrtovalskega poteka.*




Slika 14 – Nivoji abstrakcije modeliranja TLM in predlagani višji nivoji abstrakcije

Dodatni nivoji abstrakcije prispevajo k temu, da so koraki nižanja nivojev abstrakcije enostavnejši, saj prispevajo k zmanjševanju potrebnega obsega odločitev in vnosa informacij. Verjamemo, da je dolgoročni cilj raziskovanja SNSPO takšno definiranje nivojev abstrakcije, da bodo nižanja nivojev omogočena samodejno. Na nižjih nivojih abstrakcije nam je taka avtomatizacija povsem samoumevna, pomislimo samo na C, zbirni jezik in binarno kodo ali pa opis VHDL in konfiguracijski bitni niz.

Poleg definiranja višjih nivojev abstrakcije naša metodologija nudi tudi podporo za gradnjo modela, ki sistem opiše na visokem nivoju in ki ga je mogoče izvršiti in ovrednotiti. Opis sistema se začne z opisom abstraktnih operacij izračunavanja, ki obdelujejo abstraktne podatke, ki se prenašajo preko abstraktnih komunikacijskih kanalov, tako izračunavanje kot komunikacija pa sta lahko časovno povsem nedoločena ali pa so njune časovne značilnosti določene s pomočjo hitrega ovrednotenja. Prvi korak v formalnem opisu sistema predstavlja določitev pomembnih delov algoritma in njihovih vlog v sistemu. V tem koraku se določijo *prototipi sistemskih funkcij*, ki praktično predstavljajo okvirje najpomembnejših sestavnih

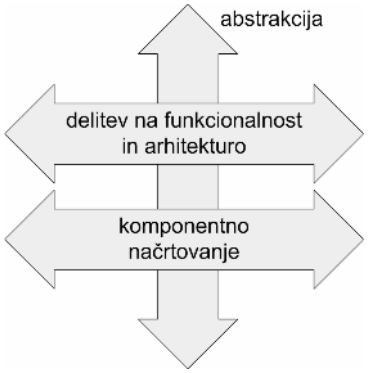
sklopov načrtovanega sistema. Skladno s predlogom naše metodologije predstavlja ta opis najvišji nivo abstrakcije v domeni izračunavanja in se nahaja veliko bližje neformalnemu začetnemu opisu sistema, kot je to pri modeliranju TLM. Nedvoumno določene podatkovne odvisnosti med visokonivojskimi sklopi sistema predstavljajo temelje za določanje osnovne visokonivojske komunikacije. Predlagamo pristop, ki ne samo da uporablja abstraktne komunikacijske mehanizme, izpeljane iz modeliranja TLM (npr. skupne spremenljivke, uporaba dogodkov, časovno nedoločena komunikacija ipd.), ampak tudi komunicira abstraktne podatke, ki obdelujejo prototipi sistemskih funkcij. Kar se tiče pravilnega sosledja obdelovanja podatkov smo ugotovili, da na tem nivoju abstrakcije za opis sistema zadostuje, če so podatki predstavljeni enostavno kot bloki informacij (spodnji del slike 14, domena podatkov, nivo abstrakcije *bloki podatkov*). Da bi bilo raziskovanje implementacijskih možnosti karseda neomejeno, smo zgradili modelirno shemo, ki omogoča sočasno izvajanje vseh funkcionalnih sklopov sistema. Pravilno zaporedje izvajanja operacij (obdelovanja podatkov) je zagotovljeno z uporabo mehanizma dogodkov; vsakič ko je blok s podatki pripravljen za predajo naprej, so prejemniki o tem obveščeni z dogodkom.

Vse skupaj je očitno zelo abstrakten nivo predstavitve sistema, saj so zajete samo najpomembnejše lastnosti sistema. Vseeno pa trdimo, da je opis sistema formalen, saj nudimo mehanizme za pretvorbo tega visokonivojskega opisa v model sistema, ki ga je mogoče izvršiti in ki zagotavlja povratno informacijo glede sistemskih odločitev, ki so vodile do podanega modela sistema. Na podlagi prototipov sistemskih funkcij, ki komunicirajo bloke informacij, je postavljen trden okvir nadaljnjega iskanja optimalne implementacije sistema. Ko so lastnosti algoritma in sistema znane podrobneje, je mogoče prototipe sistemskih funkcije dopolniti s podfunkcijami na nižjem nivoju abstrakcije. V domeni izračunavanja smo ta nivo poimenovali *integrirane funkcije*, saj so v prototipih sistemskih funkcij integrirane funkcije, ki jim natančneje določajo vlogo v sistemu. Z nadgrajevanjem znanja o sistemu postopoma pridobimo tudi več informacij o obliki podatkov, ki predstavljajo komunikacijo med podsklopi sistema. Nivo abstrakcije smo poimenovali *strukturiran blok*; zaznamujejo ga namreč podatkovni bloki, katerih struktura je čedalje natančneje določena.

Predstavljen koncept vpelje nivoje abstrakcije nad nivoji, določenimi v modeliranju TLM (v domenah izračunavanja in podatkov). Trdimo, da lahko z uporabo tega koncepta načrtovalski prostor raziščemo učinkoviteje. Primer uporabe koncepta bomo na praktičnem primeru prikazali v poglavju 6.

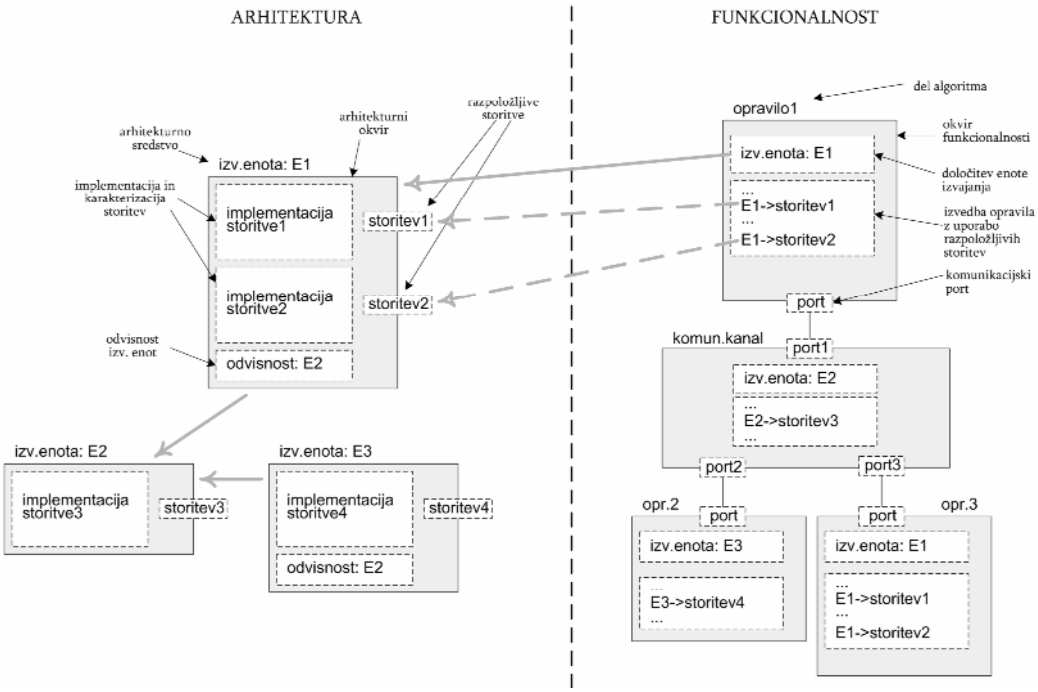
## 4.5 Okvirji

Do sedaj smo predstavili potek SNSPO in pristope, ki jih lahko uporabimo za lažje obvladovanje kompleksnosti. Slika 15 prikazuje prepletajočo uporabo konceptov – opis sistema je razdeljen na funkcionalnost in arhitekturo, posamezni sklopi so zajeti s komponentnim načinom načrtovanja, oboje pa je uporabljeno na vseh nivojih abstrakcije. Prva dva koncepta predstavljata horizontalno usmerjen pristop, abstrakcija pa predstavlja vertikalno usmerjen koncept obvladovanja kompleksnosti.




Slika 15 – Sočasna uporaba vertikalnih in horizontalnih konceptov obvladovanja kompleksnosti

Podporo za jasen, zgoščen in po domenah ločen zajem specifikacije sistema smo omogočili s pomočjo knjižnice okvirjev. Uporaba okvirjev omogoča enostavno in intuitivno apliciranje predstavljenih vertikalnih ter horizontalnih konceptov obvladovanja kompleksnosti. Poleg ločevanja domen, zajema in ovrednotenja informacij sistema okvirji nudijo tudi podlago za hitro in enostavno konstruiranje modelov, ki jih je mogoče izvršiti in ovrednotiti. Naša metodologija nudi podporo za sprejemanje trdno osnovanih načrtovalskih odločitev, ki so utemeljene na povratnih rezultatih ovrednotenja modela, npr. glede časa izvajanja, zasedenosti sredstev, prostega časa. Da bi načrtovalca razbremenili ponavljajoče implementacije podpore za običajno zahtevane lastnosti sistema, smo podporo za zajem teh informacij vgradili v okvirje in predstavljajo njihov integralni del. V postopku izvajanja modela so funkcije zajema odgovorne za samodejno zbiranje informacij o sistemu, ki se zapisujejo v obliki simulacijskih sledi. Te je mogoče po zaključku izvajanja simulacije pregledati in na njihovi podlagi priti do želenih informacij glede delovanja sistema.




Slika 16 – Opis sistema s pomočjo okvirjev in koncepti obvladovanja kompleksnosti

Primer opisa sistema z uporabo okvirjev je prikazan na sliki 16. Načrtovalec opiše arhitekturo in funkcionalnost s pomočjo uporabe primernih okvirjev, vanje zajame z njimi povezane podrobnosti in jih med seboj poveže skladno z zahtevami sistema. Za opis različnih nivojev abstrakcije in arhitekturne ali funkcionalne domene se uporabijo različni okvirji iz knjižnice. Z uporabo okvirjev je za opis funkcionalnosti potrebna implementacija samo tistih delov funkcij, ki neposredno predstavljajo delovanja algoritma (npr. opis dela algoritma, ki je predstavljen na sliki 40, str. 96). Arhitekturno-funkcionalna delitev je narejena s pomočjo okvirjev in njihovih vmesnikov. Poleg naklonjenosti ponovni uporabi so vmesniki namenjeni tudi ortogonalni dekompoziciji opisa sistema in raziskovanju različnih implementacij, ki nudijo iste storitve. Takšno eksperimentiranje z različnimi implementacijami omogoča učinkovito raziskovanje načrtovalskega prostora. Implementacija storitev je skrita znotraj okvirja, pomembne pa so samo izpostavljene lastnosti modula. Vmesnik arhitekturnega okvirja določa storitve, ki morajo biti implementirane (znotraj okvirja), obenem pa predstavlja tudi storitve, ki so na voljo za uporabo ostalim delom sistema. V naši metodologiji izvršilnih enot ne razlikujemo med seboj in označujejo vse vrste arhitekturnih sredstev, ki izvajajo zahteve funkcionalnega dela opisa. Njihova vloga v sistemu je tista, ki jih določi kot računsko, spominsko, komunikacijsko ali drugo enoto.

Funkcionalnost sistema je intuitivno opisana z mrežo opravil, ki komunicirajo preko portov in tako nudijo osnovo za komunikacijsko-računsko delitev. Za namene simulacije in ovrednotenja uporabljamo modelirno shemo, v kateri se lahko vsi uporabljeni okvirji funkcionalnosti (ki predstavljajo opravila) izvajajo sočasno. Gledano s stališča funkcionalnosti je izvajanje opravil omejeno samo z njihovo podatkovno odvisnostjo. V postopku raziskovanja se najprej razišče največji možni nivo vzporednega delovanja, ki ga določen algoritem omogoča, tega pa se kasneje zmanjša z omejitvami izvršilnih enot. Vsakemu opravilu se določi ustrezna izvršilna enota, ki je odgovorna za karakterizacijo cene izvajanja določene storitve (npr. čas, energija in velikost v smislu logičnih bokov ali števila tranzistorjev) in določanje omejitev izvajanja (npr. ne/razpoložljivost sredstev). Ustrezne izvršilne enote so tiste, ki nudijo implementacije storitev, ki jih uporabljajo opravila pri izvedbi določenega dela funkcionalnosti. Če več kot eno opravilo določi isto izvršilno enoto, je potrebno implementirati algoritem dodeljevanja izvršilne enote, ki določi rezultat takšne zahteve. Opis sistema z mrežo opravil, ki se izvajajo na ustreznih izvršilnih enotah, predstavlja osnovo za uporabo algoritmov za samodejno prostorsko in časovno dodeljevanje.

Kot prikazuje slika 16, je opis arhitekture tudi narejen na modularen način. Slika prikazuje izvršilni enoti E1 in E3, obe odvisni od izvršilne enote E2 (na tak način lahko npr. modeliramo dva procesorja, povezana na skupno vodilo). Zahteve storitev funkcionalnega dela opisa sistema so razširjene do vseh ustreznih izvršilnih enot, kar omogoča učinkovit način raziskovanja načrtovalskega prostora.

### *4.5.1 Računski modeli*

*Def. 6* **Računski model** določa način opisa funkcionalnosti s semantiko (kaj stavki opisa pomenijo) in sintakso (pravila tvorbe in povezave stavkov).

Računski model (*model of computation, MoC*) določa način opisa sistema. Ta je v tesni povezavi s semantiko (kaj sestavni bloki predstavljajo) in sintakso (kako je potrebno sestavne bloke uporabiti) gradnikov za opis sistema. V našo dosedanjo zgodbo opisa sistemov in še posebno okvirjev se vključujejo, ker izvedba okvirjev (ki je v tesni povezavi s spodaj ležečim simulacijskim ogrodjem) določa, kako lahko model opišemo, kako poteka simulacija in kakšne lastnosti bo tako opisan sistem imel po implementaciji.

Računski modeli imajo za seboj že dolgo zgodovino raziskovanja, saj njihovi začetki segajo že v 60. leta [k12.c]. Njihovo pomembnost so jim priznavali npr. že v okviru

raziskovalnega projekta Polis [rs2.b], kjer so aplikacijo predstavili z računskim modelom avtomata s končnim številom stanj za sočasno načrtovanje (*codesign finite state machines, CFSM*). Lee [p6] računske modele opisuje kot "zakone fizike" sočasno delujočih komponent, vključno s tem, kaj predstavljajo, kako komunicirajo, kako so povezani njihovi poteki nadzora in katere informacije si delijo:

- Računski modeli določajo komponente. Komponente so npr. lahko procedure, procesi, funkcije, avtomati s končnim številom stanj ipd.
- Računski modeli določajo komunikacijske protokole. Ti protokoli postavljajo omejitve mehanizmom, s katerimi lahko komponente vplivajo druga na drugo. Primera komunikacijskega protokola sta npr. asinhron prenos sporočil in komunikacija osnovana na principu rendez-vous.
- Računski modeli določajo tudi to, kaj komponente vedo druga o drugi (informacije, ki si jih delijo). Npr. delijo si lahko informacije v obliki globalnih spremenljivk.

Omenjenim lastnostim pa lahko dodamo še lastnosti, ki so pomembne za simulacijo [p30]:

- Uporabljen časovni model (realne vrednosti, cela števila, časovno nedoločen) in omejitve zaporedja dogodkov v sistemu (globalno urejeni, delno urejeni ipd.).
- Pravila za aktiviranje procesov.

Z računskimi modeli so postavljeni mehanizmi interakcije komponent, določajo pa tudi načrtovalski vzorec njihovega povezovanja. Še posebno so zanimivi tisti računski modeli, ki na različne načine obravnavajo sočasnost in vlogo časa. Izbira računskega modela ima velik vpliv na opis sistema, saj uporabljene omejitve neposredno vplivajo na lastnosti zgrajenega sistema. Lastnosti, izpeljane iz izbire računskega modela, so pomembne na različnih stopnjah poteka SNSPO, saj vplivajo na samo implementacijo (enostavnost, primernost in celo zmožnost implementacije z razpoložljivo SO), simulacijo (učinkovitost, hitrost) in tudi specifične lastnosti implementacije (brez možnosti obtičanja, strogo sprotni odziv). Model sistema je celo mogoče zgraditi na način, ki zagotavlja, da so določene lastnosti ene komponente ohranjene v celotnem sistemu. Obširno razlago računskih modelov lahko bralec najde v [p12], [k3.a] in [k12.c].

Kot bomo predstavili v nadaljevanju, v okviru naše metodologije za simulacijo in ovrednotenje modela sistema uporabljamo simulacijsko ogrodje SystemC, zato bomo na tem

mestu razložili njegove računske modele. Od različice 2.0 naprej je v jeziku SystemC mogoče uporabiti enostavna in močno prilagodljiva mehanizma dogodkov (*events*) in klica funkcije (*wait*), na podlagi katerih je mogoča implementacija različnih tipov kanalov, ne da bi bilo za to potrebno spreminjati spodaj ležeče simulacijsko jedro. V simulacijskem jedru je prisotna vsa potrebna funkcionalnost, ki predstavlja osnovo za zelo zmogljiv splošen računski model. Globalni model časa je resda vezan na celoštevilčni model[16], si pa načrtovalci lahko zgradijo določene kanale, ki natančno odražajo pravila komunikacije med procesi, aktiviranja procesov in zaporedje dogodkov v celotnem sistemu. Čeprav modeli zveznega časa, ki se npr. uporabljajo v modeliranju analognih sistemov, trenutno še niso podprti znotraj simulacijskega jedra jezika SystemC, je podprto modeliranje praktično katerega koli modela diskretnega časa [p30].

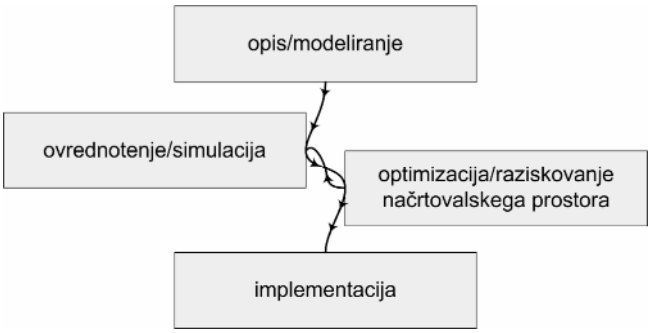
Čeprav se v okviru našega raziskovanja nismo eksplicitno ukvarjali z računskimi modeli (seveda pa smo jih uporabljali), nam je zelo pomembno dejstvo, da simulacijsko jedro SystemC nudi splošen računski model in tako v tem kontekstu praktično ne postavlja nobenih omejitev. V bodočih različicah jezika SystemC lahko pričakujemo veliko razširitev, kot npr. podporo za mehanizme, običajno prisotne v OS, in tudi modeliranje analognih sistemov [ss1]. Že sedaj pa je modeliranje takšnih sistemov mogoče preko razširitev simulacijskega jedra, npr. podpora lastnosti OS [p8] ali pa simulacija analognih sistemov [p31].

## *4.6 Podpora za ovrednotenje in simulacijo*

Na potek načrtovanja sistema lahko pogledamo tudi drugače, kot je to prikazano na sliki 6 (str. 35). Če se bolj kot na potek samega načrtovanja osredotočimo na nivoje načrtovanja, pridemo do simbolne sheme, ki je prikazana na sliki 17. Bralcu bi na osnovi do sedaj prebranega slika morala biti verjetno že poznana; načrtovanje se prične z opisom sistema, ki ga je potrebno ovrednotiti, poiskati najboljšo rešitev in to na koncu implementirati. Postopek iskanja in ovrednotenja najboljše rešitve se lahko iterativno ponavlja. Vse do sedaj obravnavane mehanizme obvladovanja kompleksnosti lahko po tej delitvi uvrstimo v zgornji nivo načrtovanja sistema. Delitev na funkcionalnost in arhitekturo, komponentno načrtovanje in abstrakcija se neposredno ukvarjajo z opisom sistema. Povedano drugače, do sedaj smo se

[16] Osnovni model za simulacijo v jeziku SystemC predstavlja model z diskretnimi dogodki (*discrete event model*). Ta je npr. uporabljen tudi pri VHDL-u, Verilogu in Simulinku. Simulacijsko jedro je razloženo v [k5] v poglavju *3.4 SystemC Simulation Kernel*.

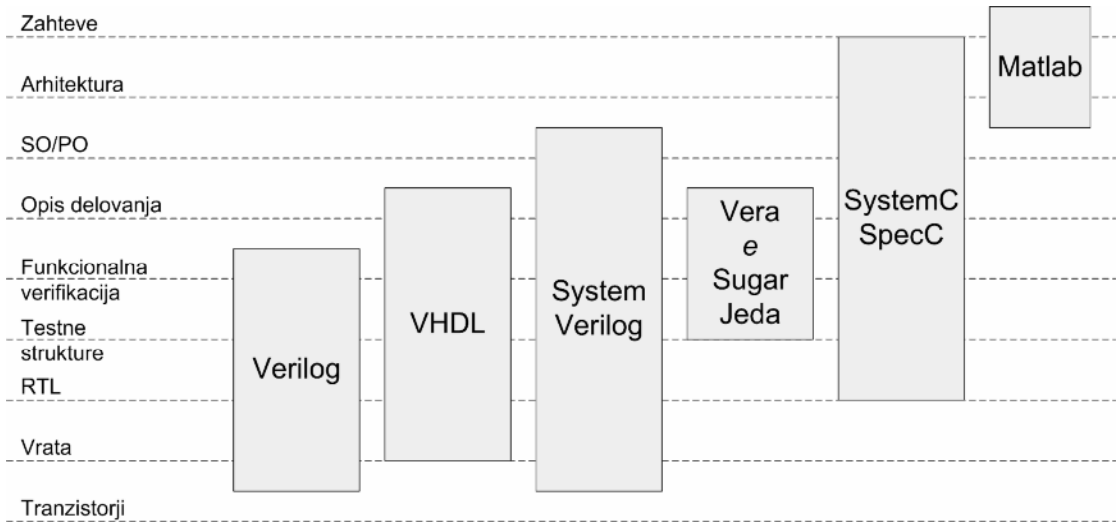
ukvarjali s koncepti visokonivojskega načrtovanja, ki so s sistemskega stališča sicer zelo pomembni, da pa bi ti koncepti lahko zaživeli in bili v postopku SNSPO praktično uporabni, pa jih je pravzaprav potrebno implementirati. Če smo do sedaj od visokonivojskega formalnega opisa zahtevali enoličen pomen in podporo za koncepte obvladovanja načrtovalske kompleksnosti, bomo sedaj temu dodali še možnost izvrševanja in ovrednotenja. V nadaljevanju poglavja se bomo osredotočili na načrtovalska orodja/jezike, ki predstavljajo osnovno ogrodje za ovrednotenje zajetih modelov, omogočajo optimizacije in tudi kasnejše implementacije. V literaturi je takšen jezik označen kot jezik za načrtovanje na sistemskem nivoju (*system level design language*, *SLDL*).




Slika 17 – Nivoji načrtovanja sistema

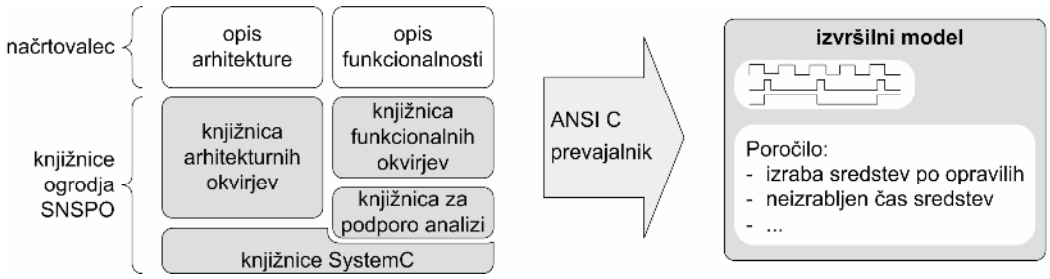
Tekom našega raziskovanja nas je ves čas vodila odločitev, da bi raje kot raziskovali v smeri povsem novih jezikov načrtovanja in njihovih podpornih orodij, uporabili že obstoječa orodja in naš raziskovalni trud usmerili v tista področja, za katera obstoječa orodja ne nudijo zadovoljive podpore. Za to smo imeli najmanj dva tehtna razloga. Prvi razlog je ta, da obstaja že množica jezikov, v katere je bilo vloženih ogromno človek-let dela. Najbolj nazoren primer je npr. programski jezik C z množico prevajalnikov, razhroščevalnikov, križnih prevajalnikov in nenazadnje z obilico dokumentacije ter dobre programerske paradigme, iz katerih je nastalo že nešteto knjig. S tega stališča je za programski del razvoja elektronskega sistema dobro poskrbljeno. Obstoječi jeziki za opis strojne opreme, npr. VHDL, zelo dobro pokrijejo strojni del razvoja sistema in nudijo načrtovalsko podporo od začetnega opisa, preko simulacije, do končne implementacije v SO. Seveda je tudi tu na voljo obširna dokumentacija in množica napotkov v obliki dobre načrtovalske prakse. Če samo pomislimo, da bi se v okviru našega raziskovanja osredotočili na gradnjo vsaj približno enakovrednega celovitega pristopa (namenski jezik in podporna orodja), ki bi pokril obe področji, lahko zelo hitro ugotovimo, da to ne bi imelo smisla. Za to bi potrebovali vsaj zelo veliko raziskovalno ekipo, kar pa je ravno

v nasprotju z dejanskim stanjem (to je pa tudi drugi tehten razlog za pametno uporabo obstoječih orodij).




Slika 18 – Primerjava področij uporabe nekaterih jezikov za načrtovanje na sistemskem nivoju [k5]

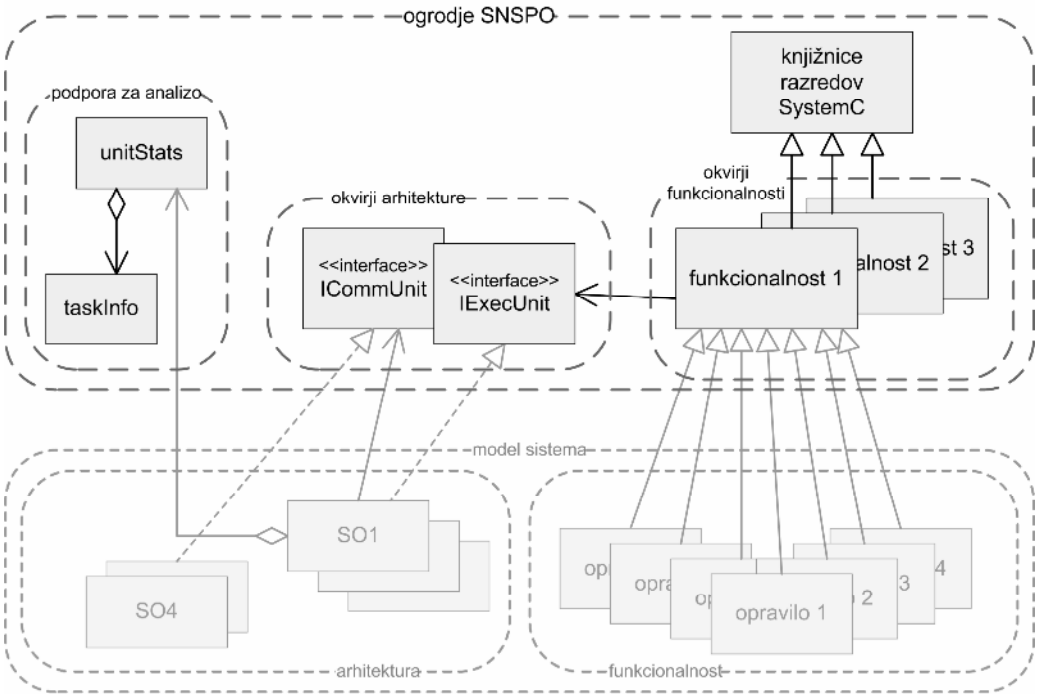
Kot je bilo predstavljeno v poglavju 2, obstajajo tako mehanizmi kot orodja, ki zelo učinkovito obravnavajo področje SO in PO ločeno vsakega zase. Zato smo se v okviru našega raziskovanja osredotočili na učinkovito uporabo že obstoječih orodij, ki smo jih nadgradili z mehanizmi za učinkovito združevanje obeh področij. Pri tem so nam bili v veliko pomoč jeziki za načrtovanje na sistemskem nivoju, na področju katerih lahko najdemo zelo bogat in raznolik nabor jezikov; slika 18 jih prikazuje samo nekaj. Med seboj se razlikujejo predvsem po nivojih, na katerih so uporabni, in po ponujenih podpornih orodjih, kar je odvisno od razširjenosti uporabe. Za naše raziskovanje so nam bili zanimivi predvsem tisti, ki podpirajo čim širši razpon nivojev in omogočajo opis PO na način, ki ni preveč omejujoč (načrtovalcu omogoča dovolj načrtovalske svobode).




Slika 19 – Podpora za simulacijo in ovrednotenje

Zaradi njegovih prednosti smo izbrali jezik SystemC (podroben opis jezika bo sledil v naslednjem poglavju), ki smo ga uporabili kot osnovo za implementacijo predstavljenih konceptov obvladovanja kompleksnosti. Knjižnice jezika SystemC smo nadgradili z lastnimi

knjižnicami okvirjev, s podporo za visokonivojske koncepte obvladovanja kompleksnosti. Načrtovalec lahko z uporabo ustreznih okvirjev knjižnice hitro in učinkovito zajame arhitekturo ter funkcionalnost opisovanega sistema. Podpora za analizo je vgrajena v knjižnico funkcionalnih okvirjev in omogoča samodejno zajemanje ter shranjevanje simulacijskih sledi, na podlagi katerih lahko ugotovimo pomembne lastnosti delovanja sistema. Simulacijsko ogrodje jezika SystemC omogoča, da model sistema, opisanega z okvirji, prevedemo in v okviru njegovega jedra tudi simuliramo njegovo delovanje (slika 19). Primer uporabe arhitekturnih in funkcionalnih okvirjev si bralec lahko pogleda v praktičnem primeru visokonivojskega razvoja kodirnika JPEG (v poglavju 6).




Slika 20 – Razredi okvirjev s shematsko prikazanim statičnim diagramom UML

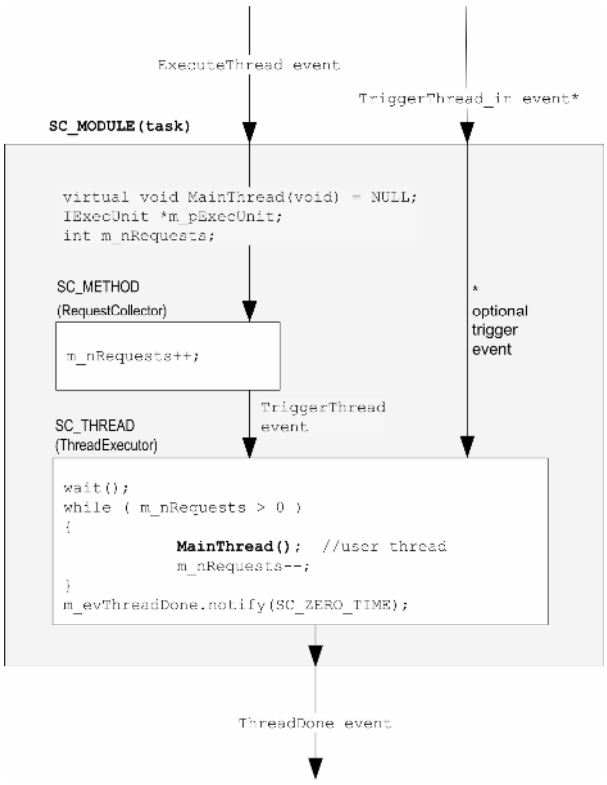
Praktična izvedba okvirjev je narejena s pomočjo razredov v jeziku C++ in na objektno orientiran način. Izhodiščno točko pri programiranju so predstavljali razredi jezika SystemC, ki smo jih nadgradili s predstavljenimi mehanizmi obvladovanja kompleksnosti. Slika 20 shematsko prikazuje razrede ogrodja SNSPO v obliki statičnega diagrama UML. Zgornji del slike predstavlja programsko podporo metodologije SNSPO in zajema knjižnice, ki jih načrtovalec uporabi za gradnjo modela. Spodnji del slike prikazuje primer uporabe knjižnic razredov in je prikazan osvetljeno, saj na tem mestu ni bistvenega pomena in samo dodatno razloži vlogo razredov ogrodja. Razredi okvirjev funkcionalnosti so izpeljani neposredno iz razredov SystemC (notacija UML, polna črta s prazno trikotno puščico), ti pa predstavljajo

osnovne razrede za izpeljavo opravil. Načrtovalec opiše funkcionalnost sistema s pomočjo mreže opravil, povezane na podlagi medsebojnih podatkovnih odvisnosti. Pri tem smo uporabili shemo modeliranja, pri kateri lahko vsa opravila svojo vlogo v sistemu igrajo sočasno (temeljni mehanizem jezika SystemC). Dokler se k opisu sistema ne dodajo omejitve s strani arhitekturnega opisa, je izvajanje opravil omejeno samo z njihovo podatkovno odvisnostjo. Okvirji funkcionalnosti ponujajo vse mehanizme opisovanja SO, ki jih nudi jezik SystemC, poleg tega pa nudijo podporo za koncepte naše metodologije. Eden od takih konceptov je na primer ločen opis arhitekture in funkcionalnosti.

Podpora za ločen opis funkcionalnosti je narejena s pomočjo vmesnikov (slika 20, *interface*). Tu smo uporabili mehanizem vmesnikov (*interfaces*) jezika C++, ki praktično predstavljajo razrede brez izvedb svojih metod (samo z deklaracijami). Arhitekturni vmesniki npr. predstavljajo funkcije, ki jih lahko uporabimo pri opisu funkcionalnosti opravil. V jeziku programiranja to pomeni, da vmesniki določajo metode (lastne funkcije), ki jih okvirji funkcionalnosti lahko uporabljajo, ne da bi bilo pomembno, kako so te metode dejansko izvedene. Da bi pohitrili opis novega modela, smo pričeli graditi različne vmesnike, ki so običajno potrebni tako pri opisu arhitekture kot funkcionalnosti. Naloga načrtovalca se zato zoži na izbiro primernih vmesnikov in implementacijo njihovih metod. Na spodnjem delu slike 20 bralec lahko opazi, da arhitekturni del modela sistema vsebuje več modulov SO, ki implementirajo določen vmesnik (notacija UML, črtkana črta s prazno trikotno puščico). Tako smo omogočili modularni opis arhitekture za lažjo obvladljivost in možnost ponovne uporabe.

Slika 21 prikazuje shematski diagram primera okvirja za zajem funkcionalnosti. Predstavljeni so najpomembnejši mehanizmi, s katerimi lahko razložimo koncept funkcionalnega opisa na podlagi opravil. Vsako novo opravilo (ki se bo izvajalo sočasno z ostalimi opravili) je potrebno izpeljati iz osnovnega razreda `sc_module` (makro `SC_MODULE`). Vlogo opravila v sistemu (za kaj je ta del opisa funkcionalnosti zadolžen) se zajame v funkciji `MainThread()`, ki je čisto virtualna (`=NULL`) in jo je potrebno obvezno določiti. Bralec lahko opazi tudi kazalec na arhitekturni element (`IExecUnit *`), ki je zadolžen za karakterizacijo izvajanja. Enote imajo privzeto možnost sočasnega izvajanja, zato smo dodali mehanizem, ki poskrbi za to, kdaj je modul aktiven in kdaj z izvajanjem čaka. To smo naredili s pomočjo mehanizma dogodkov (`event`). Kadarkoli pride do zahteve za izvajanje opravila (`ExecuteThread`), se ta zabeleži (`m_nRequests++`), in če opravilo pred tem ni bilo aktivno (je čakalo, `wait()`), prične z izvajanjem (`MainThread()`). Če je opravilo v času novega zahtevka

že bilo aktivno, je nadaljnji potek izvajanja odvisen od uporabljenega načina sprožanja ponovnega izvajanja. Zaenkrat smo vgradili dva načina. V prvem načinu, kjer `TriggerThread_in` ni določen (programsko to pomeni, da kazalec, ki bi kazal na dogodek, ni določen), se izvajanje opravila ponavlja tako dolgo, dokler čakalna vrsta zahtevkov ni izvedena (`while(m_nRequests>0)`). To pomeni, da izvršilne arhitekturne enote med ponavljanji ni mogoče prevzeti. V drugem načinu je `TriggerThread_in` določen in se izvajanje opravil kljub čakajočim zahtevkom ne izvede, dokler ne pride do dogodka `TriggerThread_in`. Z uporabo tega preprostega mehanizma smo omogočili učinkovito raziskovanje možnosti časovnega dodeljevanja. Obveščanje ostalih modulov poteka preko dogodka `ThreadDone` (zgodi se vsakič, ko je končano eno izvajanje zahteve).




Slika 21 – Shematski diagram okvirja za predstavitev funkcionalnosti

Načrtovalčev postopek opisa arhitekture zajema izbiro primernih vmesnikov ter njihovo medsebojno povezavo in karakterizacijo. Karakterizacija arhitekturnih okvirjev postavi omejitve in ceno izvajanja funkcionalnosti, povezava pa je narejena tako, da v postopku načrtovanja (natančneje v stopnji mapiranja) opravilom določimo njihovo enoto izvajanja (slika 20, notacija UML, puščica od razreda okvirja funkcionalnosti do okvirjev arhitekture – okvir funkcionalnosti ima podatke o okvirju arhitekture). S ceno izvajanja mislimo čas izvajanja, potrebna sistemska sredstva, porabo energije ipd., na podlagi katerih lahko

ovrednotimo delovanje sistema. Za samodejno beleženje dnevnika dogajanja v sistemu skrbi podpora za analizo, ki jo je mogoče pripeti vsaki implementaciji arhitekturnega vmesnika (tj. delu SO). Načrtovalec ima nadzor nad tem, kaj se beleži in kako. V knjižnici za podporo analizi smo vgradili osnovne mehanizme, načrtovalec pa lahko sam izbere, kateri sistemski parametri se beležijo, na kakšen način in v kakšni obliki naj bo izpis. Slika 22 prikazuje primer izpisa dnevnika ene arhitekturne enote (`bus`). Tu nas je zanimalo, koliko časa posamezna opravila (`entrCod`, `quant`, `frwdDCT2`, `frwdDCT1` in `rgb2yuv`) enoto aktivno uporabljajo (`active`), koliko časa opravila čakajo, da bo enota na voljo (`wait`), in za koliko časa enota pri izvajanju določenega opravila preda nalogo naprej (`forward.control`).

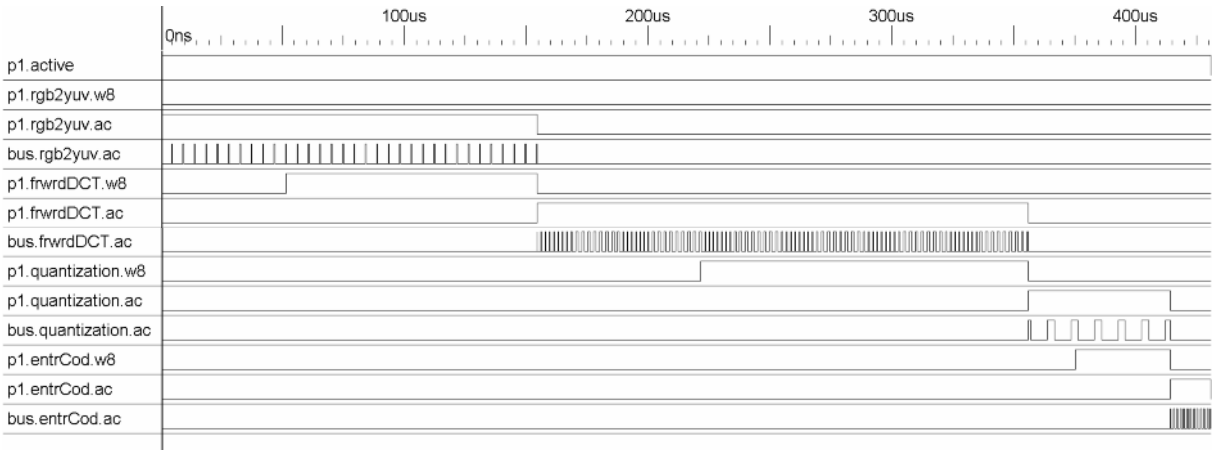
```
Unit: bus
 Unit active time (UAT): 93240 ns
 entrCod
 active: 6480 ns (6.94981% UAT)
 waiting: 3060 ns (3.28185% UAT)
 forward.control: 0 s (0% active)
 quant
 active: 17280 ns (18.5328% UAT)
 waiting: 0 s (0% UAT)
 forward.control: 0 s (0% active)
 frwdDCT2
 active: 19200 ns (20.592% UAT)
 waiting: 7900 ns (8.47276% UAT)
 forward.control: 0 s (0% active)
 frwdDCT1
 active: 38400 ns (41.184% UAT)
 waiting: 14480 ns (15.5298% UAT)
 forward.control: 0 s (0% active)
 rgb2yuv
 active: 11880 ns (12.7413% UAT)
 waiting: 1320 ns (1.4157% UAT)
 forward.control: 0 s (0% active)
```

Slika 22 – Primer izpisa dnevnika arhitekturne enote

Poleg povzetih oz. statističnih izpisov je v okviru jezika SystemC vgrajena tudi podpora za časovne sledi posameznih spremenljivk sistema. Tu je odvisno predvsem od iznajdljivosti načrtovalca, katere parametre sistema bo želel časovno zabeležiti. Slika 23 prikazuje časovno aktivnost v sistemu, kjer dve arhitekturni enoti (`p1` in `bus`) opravljata različna opravila opisa funkcionalnosti (`rgb2yuv`, `frwrdDCT`, `quantization` in `entrCod`). Vidimo lahko npr., kdaj je posamezna enota aktivna (`active` / `ac`) in kdaj določeno opravilo enoto čaka (`w8`).

V tem poglavju smo predstavili najpomembnejše mehanizme implementacije okvirjev v programskem jeziku C++ na osnovi jezika SystemC. Detajlna razlaga celotne implementacije bi bila preobsežna in bi lahko celo zameglila prvoten namen predstavljene doktorske disertacije. Postaviti smo želeli enovito ogrodje visokonivojskega raziskovanja SNSPO, na podlagi katere bo mogoče zgraditi okolje s celovito podporo, ki bo dejansko uporabna vsem

načrtovalcem kompleksnih elektronskih sistemov (in ne zgolj v raziskovalni domeni). Bralcu, ki želi obravnavano tematiko podrobneje raziskati, pa je vsa koda in vsa uporabljena PO prosto dostopna pri avtorju ali v Laboratoriju za načrtovanje integriranih vezij na Fakulteti za elektrotehniko.




Slika 23 – Primer časovnih sledi aktivnosti v sistemu[17]

### *4.6.1 SystemC*

V poglavju 2 smo predstavili izzive načrtovanja sodobnih elektronskih sistemov, s katerimi se pri raziskovanju SNSPO soočamo na vseh nivojih načrtovanja. Ob povečujoči se načrtovalski kompleksnosti se za potrditev sistemskih konceptov zahteva hitro izvršljivo specifikacijo sistema. Potreben je jezik, ki nudi zadostne nivoje abstrakcije, integracijo SO in PO ter nenazadnje tudi zadostno zmogljivost. Da bi bila možna enovita uporaba orodij za načrtovanje sistemov, storitev in blokov intelektualne lastnine, se nadalje zahtevata tudi skupen načrtovalski jezik in skupna osnova za modeliranje. Kot odgovor na vsesplošno potrebo po jeziku, ki bi izboljšal celotno produktivnost načrtovanja elektronskih sistemov, je bil razvit jezik SystemC [ss1]. Namenjen je za načrtovanje na nivoju sistema, z njegovo uporabo pa je na različnih nivojih abstrakcije mogoče predstaviti funkcionalnost, komunikacijo, SO in PO, vse zajeto skupaj, tako kot bo tudi uporabljeno v končnem sistemu. Predstavlja standardiziran[18] jezik za modeliranje, namenjen podpori za načrtovanje na sistemskem nivoju in izmenjavi intelektualne lastnine vseh nivojev abstrakcije, za sisteme, ki vsebujejo tako strojne kot programske komponente.

[17] Časovne sledi so pridobljene z mehanizmom beleženja spremenjenih vrednosti (*VCD, value change dump*), ki ga podpira SystemC že v osnovi. Uporabljen je pregledovalnik *Wave VCD Viewer* (www.iss-us.com/wavevcd/index.htm).
[18] Decembra 2005 so bile knjižnice jedra jezika SystemC 2.1 standardizirane pod okriljem standarda IEEE [ss1].

Strogo gledano, SystemC ni pravi programski jezik, ampak je narejen kot knjižnica razredov znotraj zelo uveljavljenega jezika C++, ki v svojem jedru nudi podporo za različne računske modele in načrtovalske metodologije. Načrtovalske knjižnice in modeli, ki nudijo podporo za te metodologije, so obravnavane ločeno od standarda jedra jezika. SystemC je v celoti zgrajen na jeziku C++, izvorna koda referenčnega simulatorja pa je prosta dostopna, skladno z dogovorom o odprtem programiranju na [ss1]. Model sistema, opisanega v SystemC, je mogoče prevesti in izvršiti na kateri koli platformi, za katero je na voljo ANSI skladen prevajalnik C++.

Podobno kot pri jezikih za opis SO, jezik SystemC omogoča strukturirano zgrajen opis sistema z uporabo modulov, priključkov in signalov. Module je mogoče uporabiti znotraj drugih modulov, s čimer je omogočen hierarhični opis. Priključki in signali omogočajo izmenjavo podatkov med moduli, oboje pa si lahko načrtovalec prilagodi po meri. Sočasno delujoči sklopi sistema so modelirani s procesi. Te si lahko predstavljamo kot neodvisne niti, ki z izvajanjem nadaljujejo vsakič, ko se spremenijo določeni signali, in izvajanje začasno ustavijo, ko določena opravila zaključijo. Ker se procesi izvajajo sočasno in uporabnik njihovo izvajanje lahko prekine ter nadaljujejo kadarkoli (v SystemC je tak tip označen kot `SC_THREAD`), uporaba procesov v splošnem zahteva svoj neodvisen sklad izvajanja. Določeni tipi procesov, katerih izvajanje se prekine v določenih točkah, pa za svoje izvajanje ne potrebujejo neodvisnega sklada (v SystemC je tak tip označen kot `SC_METOD`). Uporaba tega drugega tipa procesov izredno izboljša učinkovitost simulacijskega jedra.

Signali SO imajo več lastnosti, ki zahtevajo posebno obravnavo, če jih modeliramo programsko. Nedoločeno stanje signala, možnost več krmilnih vezij na isti liniji in zakasnitev izhodne vrednosti so lastnosti, ki zahtevajo posebno pozornost. Jezik SystemC ima vgrajeno podporo za vse troje. Eden izmed izzivov pri razvoju jezika za načrtovanje na sistemskem nivoju je podpora za širok nabor računskih modelov, nivojev abstrakcije in načrtovalskih metodologij, uporabljenih v razvoju sistema. V ta namen je v SystemC vgrajeno majhno jedro, ki predstavlja osnovo za splošno namensko modeliranje, na katero se dodajo računski modeli, načrtovalske knjižnice in koraki načrtovanja, ki se zahtevajo pri specifični metodologiji. Majhno, splošnonamensko jedro jezika SystemC je označeno z izrazom jedro jezika (*core language*) in predstavlja osrednji del standarda SystemC.

Splošno modeliranje komunikacije in sinhronizacije je v jeziku SystemC podprto z uporabo kanalov (*channels*), vmesnikov (*interfaces*) in dogodkov (*events*). Kanali so objekti,

ki služijo kot vsebniki za komunikacijo in sinhronizacijo. Kanali implementirajo enega ali več vmesnikov. Vmesnik definira nabor metod kanala, za katere pa sam ne nudi implementacije. Implementacijo kanala je tako mogoče skriti znotraj njega, kar je ključnega pomena za zagotavljanje prilagodljivega modeliranja komunikacije. Dogodek je prilagodljiv, nizkonivojski sinhronizacijski mehanizem, ki predstavlja temelje za gradnjo ostalih oblik sinhronizacije. Z uporabo kanalov, vmesnikov in dogodkov lahko načrtovalec modelira široko področje komunikacijskih in sinhronizacijskih mehanizmov, uporabljenih v načrtovanju sistema. Mogoče je npr. modeliranje signalov SO, skladov (FIFO, LIFO ipd.), semaforjev, spominov in vodil ipd.

Z razliko od PO je pri obravnavi SO čas pomemben. Pri PO se je zavedanje časa skrčilo na število instrukcij, ki so potrebne za izvedbo določene funkcije. Nadalje tudi podpora za realni čas ne nudi neposredne obravnave časa, ampak preko prioritete izvajanja opravil, ki jo lahko razumemo kot nekakšno statistično obravnavo odziva sistema. Tekom simulacije modela, zgrajenega iz knjižnic SystemC, se čas obravnava kot celoštevilska vrednost, ki se doda vsem dogodkom v sistemu. Na podlagi tega časa lahko pridemo do sledi dogodkov v sistemu in tekstovnih ter slikovnih izpisov, kot smo jih predstavili v prejšnjem poglavju. Pravilno modeliranje signalov SO je izvedeno preko mehanizma delta korakov, ki ga lahko razumemo kot zelo majhno povečanje časa znotraj simulacije, ki pa ne vpliva na povečanje simulacijskega časa, ki je viden od zunaj. V določenem časovnem koraku se tako lahko zgodi več delta korakov. Ko se signalu dodeli določena vrednost, ostali procesi te vrednosti ne vidijo, dokler ne nastopi nov delta korak. Procesi, občutljivi na spremenjen signal, lahko v tem primeru nadaljujejo z izvajanjem. Podrobnejšo razlago lahko bralec najde v [k5.b]. Prednost, ki jo ima jezik SystemC v primerjavi z ostalimi jeziki za opis SO, ki delujejo na podobnem principu simulacije, je hitrost izvajanja, saj je količina informacij na višjih nivojih abstrakcije lahko znatno manjša.

Slika 24 povzema najpomembnejše lastnosti arhitekture jezika SystemC, ki smo jih predstavili v tem poglavju. Bralcu, ki se želi podrobneje spoznati z jezikom SystemC, priporočamo delo [k5], ki zelo dobro predstavi tako samo modeliranje kot tudi sestavne dele jezika SystemC. To delo [k6] oriše uporabo jezika SystemC iz tehničnega vidiki na naboru več nezahtevnih, a povsem praktično uporabnih primerih. Najnovejša koda je prosto dostopna na [ss1], kjer je na voljo tudi ažurna spletna podpora za tehnične težave v obliki foruma. Bralca bi na tem mestu opozorili še na to, da v povezavi z jezikom SystemC poteka zelo

veliko aktivnosti tako akademske kot tudi industrijske narave. Zelo pomembno nadgradnjo predstavlja verifikacijska knjižnica (*SCV library, SystemC verification library*), ki omogoča združevanje jezika SystemC z mehanizmi, ki omogočajo učinkovito verifikacijo sistema. Knjižnica SCV predstavlja nadgradnjo programskega orodja *TestBuilder* proizvajalca Cadence in je prosto dostopna [ss1], [ss7].


Slika 24 – Arhitektura jezika SystemC

### *4.6.2 Ostali jeziki sistemskega načrtovanja*

Na področju jezikov za načrtovanje na sistemskem nivoju je mogoče najti zelo veliko jezikov, tako komercialnih kot tudi prosto dostopnih ali povsem raziskovalno-akademskih. Pri raziskovanju tega področja smo se soočali s problemom njihove dejanske uporabnosti in korektno narejenih primerjav med posameznimi jeziki. Sicer je motivacija snovalcev ali prodajalcev teh jezikov jasna, se pa od uporabnika zahteva že razmeroma dobro poznavanje SNSPO, da lahko izbere pravo smer raziskovanja. Prav zato bomo to poglavje namenili temu, da bralca opozorimo na določene lastnosti, za katere smo ugotovili, da so pomembne za samo prilagodljivost in učinkovitost izpeljave postopka SNSPO. Razložili bomo, zakaj smo izbrali jezik SystemC in ne katerega drugega.

Podobno modeliranje sistema, kot to omogoča jezik SystemC, omogoča tudi jezik SpecC [ss2], [rs3]. Z razliko od jezika SystemC jezik SpecC temelji na jeziku C in ne C++, oba pa izkoriščata njune prednosti (dobro poznan in razširjen jezik), nudita koncepte za gradnjo vgrajenih sistemov (vedenjsko in strukturno hierarhijo, sočasno delovanje, sinhronizacijo, zavedanje časa, podatkovni tipi SO ipd.), vsebujeta simulacijsko jedro (posamezne njune poddele je mogoče neposredno implementirati), predstavljata osnovo za skupen opisni jezik in osnovo za izmenjavo opisa. Kot smo že omenili, lahko model, opisan v jeziku SystemC,

enostavno prevedemo s standardnim prevajalnikom, v primeru jezika SpecC pa je potreben poseben prevajalnik (ki ga je sicer mogoče dobiti prosto). To nam v trenutku pisanja še ni predstavljalo resnejše omejitve. Poseben prevajalnik je potreben zato, ker so standardnemu jeziku C dodatne razširitve jezika C. Ob pregledu dogajanj, povezanih z jezikoma, lahko ugotovimo, da so aktivnosti, povezane z jezikom SystemC, mnogo večje kot pri SpecC (spletna stran, seminarji, konference, podpora, orodja, razširitve ipd.), poleg tega pa je bil jezik SystemC sprejet kot standard v okviru standarda IEEE. Na podlagi aktivnosti, standardizacije in vpletenosti svetovno pomembnih podjetij lahko upravičeno sklepamo, da bo razvoj jezika SystemC potekal še naprej in da se ne bo ustavil na mrtvi točki, kot to lahko dobimo vtis pri jeziku SpecC.

Na voljo so tudi jeziki, ki pokrivajo isto področje kot jezika SystemC in SpecC, za katera pa po podrobnem pregledu ugotovimo, da je njihov namen nekoliko drugačen. Primer takšnega jezika je Handel-C [ss10], ki ni zasnovan kot jezik za opis SO, ampak kot visokonivojski programski jezik, ki pa kot izhod nudi SO. Nizkonivojski problemi načrtovanja so načrtovalcu povsem zakriti in za njih poskrbi prevajalnik. Tako se programer lahko posveti implementaciji sami. Posledično je razvoj SO s pomočjo jezika Handel-C bolj podoben programiranju PO kot razvoju SO in kot tak namenjen programerjem s pomanjkljivim poznavanjem SO.

Nadaljnjemu obširnemu opisovanju jezikov, primernih za visokonivojsko načrtovanje heterogenih elektronskih sistemov, se bomo izognili, ker bi bila podrobna raziskava obstoječih jezikov preobsežna. Naš namen je bralca le opozoriti na določene značilnosti jezikov, ki sprva morda niso tako očitne, od njih pa je odvisna kakovost implementacije metodologije SNSPO. Zavedati se je potrebno, da je potrebno vložiti sorazmerno veliko truda, da se spoznamo s strukturo in z načinom uporabe posameznega jezika. Še veliko več truda pa je potrebno vložiti za implementacijo načrtovalske metodologije na osnovi določenega, spodaj ležečega načrtovalskega jezika.

## *4.7 Načrtovalski prostor, algoritmi in optimizacija*

Če se pri poteku SNSPO ponovno osredotočimo na nivoje načrtovanja, predstavljene na sliki 17 (str. 59), ugotovimo, da smo vse do sedaj obravnavali samo zgornja dva nivoja. Bralec je iz obravnavanega lahko dobil vtis, kako pomembno vlogo v poteku SNSPO imajo visokonivojski mehanizmi obravnave načrtovanja. Predstavili smo potek modeliranja sistema

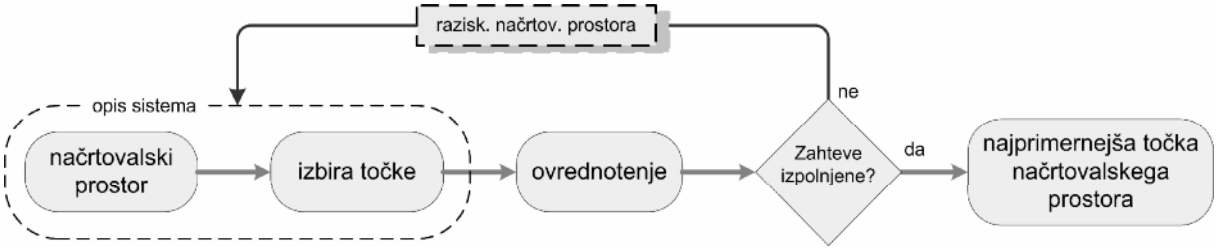
in mehanizme njegovega ovrednotenja. Med opisovanjem sistema običajno naletimo na situacijo, v kateri bi lahko npr. za implementacijo iste funkcionalnosti uporabili različne algoritme, isti algoritem bi lahko opisali z različnimi stopnjami vzporedne obdelave, od raziskovalca do raziskovalca je tudi odvisno, kako bo naredil delitev podsklopov iste funkcionalnosti ter kakšno arhitekturo bo izbral. To je le nekaj možnih primerov načrtovalskih odločitev, ki so v postopku SNSPO nekaj povsem običajnega. Še več, v okviru naše metodologije so takšni primeri različne implementacije dobrodošli, saj predstavljajo potencialne kandidate za implementacijo sistema. Pri SNSPO množico vseh kandidatov označimo s terminom *načrtovalski prostor*, ki zajema vsa tista vprašanja: "Kaj pa če bi poizkusili tako ...?", na katera naletimo na vseh stopnjah opisa sistema. Bralcu je bilo razloženo, zakaj je pomembno preveriti čim več načrtovalskih možnosti in že tudi to, kako lahko s pomočjo *abstrakcije* hitro in učinkovito ovrednotimo *ustreznost* posamezne odločitve. Pri opisu realno kompleksnega elektronskega sistema zelo hitro pridemo do točke, ko je načrtovalskih odločitev lahko zelo veliko. Ker pa s sprejemanjem posameznih načrtovalskih odločitev tvorimo kombinacije različnih implementacij sistema, je hitro jasno, da kaj hitro pridemo do ročno neobvladljivega števila različnih implementacij[19].

Z uporabo do sedaj predstavljenih konceptov obravnave kompleksnosti, lahko tvorbo različnih implementacij sistema in njihovo ovrednotenje avtomatiziramo. Pridemo do algoritmov SNSPO kot so funkcionalna delitev (*partitioning*), mapiranje (*mapping*) in časovno dodeljevanje (*scheduling*). Pomembno je razumeti, da ti algoritmi vplivajo na načrtovanje sistema na vseh nivoji abstrakcije. Postopek nižanja nivoja abstrakcije in iskanje najprimernejše točke trenutnega nivoja abstrakcije si lahko predstavljamo kakor dodajanje novih dimenzij načrtovalskemu prostoru, ki pa se istočasno krči. Optimizacijski algoritmi, ki se ukvarjajo z iskanjem najprimernejše točke načrtovalskega prostora, sodijo v razred NP kompletnih problemov [wp3], ki jih ni mogoče analitično razrešiti. Posamezni algoritmi SNSPO predstavljajo povsem svoje raziskovalno področje, ki je optimizacijsko zelo kompleksno in zato se raziskovalci lotevajo hevrističnih metod iskanja najustreznejše rešitve. Algoritmi SNSPO so vodeni z načrtovalskimi zahtevami in praviloma velja, da določena lokalna optimizacija ne predstavlja globalno najboljše rešitve. Upoštevati je namreč potrebno globalno metriko ovrednotenja sistema – v okviru celotne slike SNSPO je naš cilj na podlagi

[19] Tu lahko najdemo razlog, zakaj se pri ročni izvedbi že povprečno kompleksnega sistema izbere premočrtno pot od ideje do končne izvedbe sistema in je posledično pridobljena rešitev (najverjetneje) suboptimalna.

lokalnih optimizacij algoritmov SNSPO poiskati globalno optimalno točko načrtovalskega prostora.

*V tem se naše raziskovanje tudi loči od veliko ostalih raziskovanj. K SNSPO smo namreč pristopili celovito in se nismo osredotočili samo na iskanje določenega algoritma, ki nudi nekakšne samoovrednotene rezultate posameznega podsklopa. Priznamo sicer, da so te raziskave prav tako pomembne kot naše, je pa njihove zaključke težko resnično ovrednotiti, če jih ne ovrednotimo v sklopu celovitega načrtovanja določenega realnega primera. To je bil tudi poglavitni razlog, ki nas je vodil v gradnjo enovitega ogrodja SNSPO.*




Slika 25 – Iskanje najprimernejše implementacije sistema

Slika 25 prikazuje iterativni postopek iskanja najprimernejše implementacije obravnavanega sistema. Opis sistema, ki ga pripravi načrtovalec, je sestavljen iz množice vseh potencialnih kandidatov izvedbe sistema in natančneje določen z izbiro ene same izvedbe. Šele nato je sistem mogoče ovrednotiti. Za izbiro te določene izvedbe poskrbijo prej omenjeni algoritmi, ki so vodeni s povratno informacijo glede izpolnjevanja zahtev. S tem delom SNSPO smo se v okviru našega raziskovanja ukvarjali samo do te mere, da smo pripravili mehanizme za vklop zunanjih algoritmov (raziskovalnih dosežkov drugih raziskovalnih skupin). Podrobna razlaga uporabe in izvedbe optimizacijskih algoritmov močno presega okvirje te disertacije ter jo bralec lahko najde v literaturi, npr. [p24], [p36].

# 5 Modelirno okolje

Bralca smo seznanili z načrtovalskim potekom SNSPO, z visokonivojskimi mehanizmi obvladovanja kompleksnosti, pokazali njihovo programsko implementacijo s pomočjo okvirjev, predstavili, kako lahko takšen opis sistema ovrednotimo s pomočjo modelirnih jezikov za opis visokonivojskih lastnosti sistema, ter se dotaknili optimizacijskih metod, ki pomagajo hitro in učinkovito poiskati ustrezno implementacijo sistema. Če malce razmislimo, smo v imenu obvladovanja kompleksnosti načrtovanja prišli do orodij/mehanizmov, katerih uporaba je nič drugega kot kompleksna. Razlika v kompleksnosti pa je ta, da so orodja in mehanizmi v tem primeru določeni do te stopnje, da je mogoče postopke načrtovanja avtomatizirati.

Prišli smo do zaključka, da potrebujemo krovno ogrodje, ki bi vse te koncepte in rešitve zajelo ter omogočilo povezavo vseh korakov – od zajema ideje do predaje zadostnih in ustreznih podatkov orodjem za implementacijo. Z avtomatizacijo načrtovalskega postopka bi našo metodologijo celostno ovrednotili in dodali ključni element obvladovanja kompleksnosti SNSPO ter tako načrtovanje naredili zares učinkovito, dobili bi pa tudi povratno informacijo o tem, kateri deli potrebujejo dodelavo oz. nadgradnjo. Poleg enostavnega vklopa raziskovalnih dosežkov ostalih raziskovalcev bi imeli tudi možnost učinkovitega (realnejšega) ovrednotenja njihovih ožje usmerjenih algoritmov.

Lastnosti, ki bi jih pri takšnem krovnem ogrodju pričakovali, lahko strnemo v naslednje točke:

- skupen opisni model vseh lastnosti sistema, tako glede mehanizma zajema kot tudi oblike zajetih podatkov,
- pregledno hierarhično predstavitev sistema,
- podporo za gradnjo knjižnic pogosto uporabljenih elementov z enostavno in učinkovito ponovno uporabo,
- podporo za avtomatizirano kreiranje načrtovalskega prostora kot tudi njegovo obvladovanje (dodajanje in ovrednotenje možnih načrtovalskih odločitev),

- centralen nadzor nižje ležečih orodij, njihovo klicanje in uporaba njihovih rezultatov,
- podporo za modularen vklop algoritmov podsklopov SNSPO.

Zadnja leta smo priča porastu grafičnih modelirnih orodij za razvoj sistemov, ki pravzaprav obsegajo nabor orodij za modeliranje, analizo, simulacijo in generiranje kode. Ta nabor orodij igra pomembno vlogo pri razvoju sistemov v specifičnem in dobro določenem inženirskem področju. Služi zajemu specifikacije s pomočjo modelov domene, nudi podporo načrtovalskemu postopku s pomočjo avtomatizirane analize in simulacije sistema ter omogoča avtomatično kreiranje, prilagajanje in integriranje komponent ciljne aplikacije. Primeri takšnih, za domeno specifičnih okolij, so npr. Rational Rose [ss11] (objektno orientiran razvoj PO), Matlab/Simulink [ss12] (obdelava signalov) in LabView [ss13] (meritve). Prednosti takšnih modelirnih okolij izvirajo iz njihove specifične usmerjenosti v določeno domeno. Za domeno specifične metodologije modeliranja namreč omogočajo zgoščeno oz. jasno predstavitev ključnih načrtovalskih zornih kotov, formalen opis in samodejno preverjanje omejitev sestavljanja sistema ter gradnjo modela, ki je skladna s potekom načrtovalske metodologije določene problemske domene.

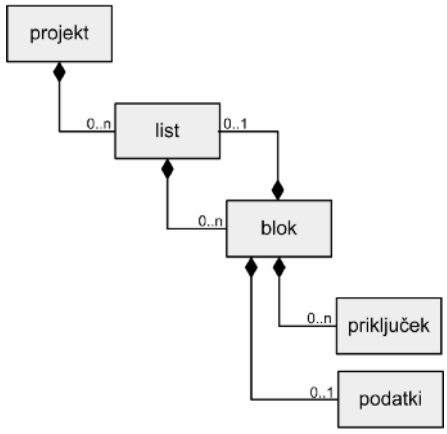
## *5.1 Lastno modelirno okolje*

Iskali smo modelirno okolje, ki bi nudilo zajem in predstavitev informacij v različnih domenah in imelo vgrajene mehanizme samodejnega generiranja datotek. Slednje bi moralo biti prilagodljivo, tako da bi generiranje datotek lahko naredili skladno s predstavljenim konceptom okvirjev ter podporo za simulacijo in ovrednotenje. Natančneje, iskali smo okolje, v katerem bi lahko gradili oz. povezovali bloke, katerih pomen bi določili sami, imeli bi podporo za koncept domen in hierarhično predstavitev sistema, na podlagi zgrajene blokovne sheme (hierarhije sistema in domen) pa bi bilo možno zahtevane datoteke jezika SystemC ustvariti samodejno. Pri iskanju potencialnih kandidatov takšnega modelirnega okolja nismo imeli uspeha, saj smo naleteli samo na okolja, ki so bila ozko namenska in zaprta ter torej povsem nekaj drugega, kar smo iskali. Odločili smo se, da takšno okolje zgradimo sami, žal pa si v začetku nismo niti predstavljali, kako zahtevne naloge smo se lotili.

Okolje smo gradili v programskem okolju MS Visual Studio 7.1 na osnovi knjižnic MFC (*Microsoft Foundation Class*) s konceptualno zasnovo MDI (*multiple document interface*). Potek gradnje okolja lahko razdelimo v približno tri glavne sklope: organizacijsko hierarhijo

(organizacija podatkov), grafično predstavitev in manipulacijo s podatki (shranjevanje, branje in iskanje – za opis sistema in knjižnice komponent).

Želeli smo zgraditi odprt in uporabniku prilagodljiv sistem, ki bi postavljal čim manj omejitev glede modelnega načina opisa. Za opis celotnega sistema smo uporabili *projekt*. Domene opisa sistema, so predstavljene z *listi*, kjer je zaradi preglednosti ena domena lahko razdeljena na več listov, toda en list lahko predstavlja samo eno domeno. Komponente v domeni so predstavljene z *bloki*. Relacije med komponentami posamezne domene so implementirane s povezavami med *priključki* blokov. Bloku so pripeti *podatki* v obliki tekstovne datoteke, ki določajo vlogo in pomen komponent posamezne domene. Blok lahko vsebuje tudi list, s čimer je podprta (poljubno globoka rekurzivna) hierarhija opisa. Shematski prikaz povedanega prikazuje v notaciji UML slika 26.

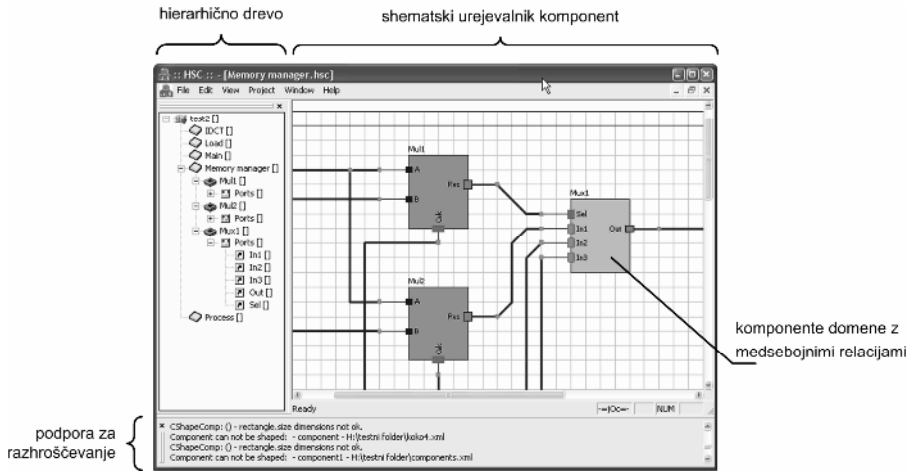



Slika 26 – Shematski prikaz organizacije podatkov opisa sistema

Grafično predstavitev bloka (komponente sistema) smo zasnovali modularno in na način, ki uporabniku omogoča enostavno in prilagodljivo uporabo. Modularno pomeni to, da smo pripravili načine, ki omogočajo, da si uporabnik vsak blok (komponento) sestavi iz elementarnih gradnikov. Ti gradniki so okvir, tekst in priključek, njihove grafične parametre pa je mogoče enostavno določiti v tekstovni obliki v datoteki komponent. Realizacija povezav (relacije med komponentami) nam je vzela še zlasti veliko časa, saj je bilo potrebno poskrbeti za veliko podrobnosti, kot npr. spojne točke med posameznimi vejami iste povezave, spojne točke povezav in priključkov komponent ipd.

Ker smo želeli odprt sistem, ki bi ga bilo mogoče enostavno uporabiti v različne namene, smo želeli opis sistema (domeni specifični podatki in podatki specifični opisu sistema) in posamezne komponente sistema shraniti na način, ki bo omogočal splošno uporabo. Tu mislimo na uporabljene komponente, njihovo postavitev, povezave, grafične lastnosti,

podatke specifične opisovani domeni ipd. Sprva smo to implementirali v obliki tekstovne datoteke in smo hitro prišli do točke, ko je zaradi množice vseh podatkov nismo več učinkovito obravnavali. Sledila je nadgradnja v obliko XML [ss14], ki pa nam je postala z globino hierarhije programsko težko obvladljiva in se nismo mogli več izmikati programiranju dostopa do podatkovne baze. Uporabili smo implementacijo Microsoft Jet DB preko vmesnika ADO[20].




Slika 27 – Uporabniški vmesnik lastnega modelirnega okolja

Slika 27 prikazuje primer uporabe modelirnega okolja po nekaj mesecih razvoja, učenja tehnik programiranja in spoznavanja programskih knjižnic C, ko je program obsegal približno 24 tisoč vrstic kode. Razvoj smo pripeljali tako daleč, da je bilo okolje že praktično uporabno za vnos lastnih komponent, njihovo povezovanje, obvladovanje celotnega projekta, izvoz poročila komponent ter vseh povezav ipd. Prav nad vsemi želenimi lastnostmi okolja smo imeli popoln nadzor, kar je seveda posledica lastne zasnove sistema.

Naslednji cilj je bil vgraditev podpore za uporabniško določen vnos pravil sestavljanja sistema. Želeli smo vgraditi mehanizme, ki bi uporabniku omogočali, da bi pri opisu sistema lahko enostavno določil, kaj je veljaven način modeliranja in kaj ne. Npr. pri opisu funkcionalnosti sistema lahko dodajamo v list (domena funkcionalnosti) komponente, ki predstavljajo njegove sklope oz. opravila. Opravila morajo biti med seboj povezana po določenih pravilih smeri podatkovnega pretoka. Postavitev arhitekturne komponente v opis te domene ne bi imela pomena in tudi povezava z ostalimi komponentami v domeni

[20] Knjižnico C++ za obravnavo vmesnika smo dobili na strani [ss15]. Stran je v celoti namenjena programerskim projektom in je med programerji zelo priljubljena. Na njej je moč najti odgovore na praktično vsa vprašanja, zelo veliko programskih knjižnic je tudi prosto objavljenih. Stran zato priporočamo.

funkcionalnosti ni smiselna. Po drugi strani pa je povezava med funkcionalnim opravilom in arhitekturnim sredstvom povsem smiselna v domeni mapiranja – ta povezava namreč predstavlja, da bo določeno arhitekturno sredstvo zadolženo za izvajanje določenega funkcionalnega opravila in je povsem običajen del SNSPO. Primerov takšnih omejitev je še veliko več in ker smo to želeli združiti z idejo odprtega sistema modeliranja, se nekaj časa nismo premaknili z mrtve točke. Ustrezne rešitve ni bilo na dlani. Če bi se odločili omejitve vgraditi v programsko kodo okolja, bi to pomenilo, da bi morali predvideti vse vrste omejitev, ki nastopijo pri SNSPO (teh je veliko), okolje pa ne bi več predstavljalo odprtega modelirnega okolja. Tako kot smo poskrbeli za grafično predstavitev komponent, njihov pomen in zajem podatkov (vse to uporabnik določi sam, skladno z zahtevami modeliranja), tako smo želi tudi zajem omejitev implementirati na način, ki bo ves nadzor prepuščal uporabniku. Raziskovanje nas je pripeljalo do jezika OCL, ki je določen znotraj jezika UML [ss3]. Obema se bomo posvetili v nadaljevanju.

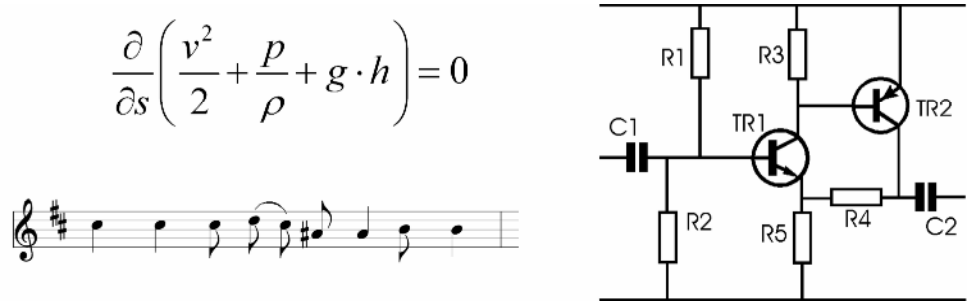
Dokaj specifične potrebe implementacije in dejstvo, da smo si že relativno dobro oblikovali idejo odprtega okolja za modeliranje, so nas preko jezika OCL in UML pripeljale do projekta GME [ss16]. Potrebovali smo malo časa, da smo ugotovili, da je projekt GME točno to, kar potrebujemo, in da v modeliranju implementira še vse tiste aspekte, na katere pri našem okolju še nismo razmišljali. Postalo je jasno, da smo z razvojem lastnega okolja zapeljali v slepo ulico in da je brez pomena vztrajati na tej poti samo zato, ker smo si jo tekom našega raziskovanja izbrali v trenutku, ko nismo poznali boljše poti. Lasten razvoj okolja smo kljub delnemu neuspehu predstavili zaradi več razlogov. Prvič, takšen je bil naš potek raziskovanja, za katerega smo porabili veliko truda in časa, in drugič, koristen je bil do te mere, da nas je seznanil s specifičnimi zahtevami okolja in na koncu pripeljal do okolja projekta GME. Bralca pa smo obenem opozorili na dejstvo, da lasten razvoj sistema zahteva zelo veliko vloženega truda.

Da bi bralec lahko razumel namembnost oz. pomembnost projekta GME, je potrebno osnovno poznavanje jezika UML in naknadno še jezika OCL, katera bomo pregledno predstavili v nadaljevanju. Zatem bomo predstavili projekt GME, kjer bo bralec lahko uvidel, s kakšno eleganco so raziskovalci implementirali vse to, kar smo mi predstavili v tem poglavju, in nadalje še prilagoditev oz. uporabo projekta GME za postopke naše metodologije SNSPO.

## 5.2 Jezika UML in OCL

Modeliranje je koncept, ki je v našem vsakdanjiku prisoten na vsakem koraku in ga čisto nezavedno ves čas s pridom izkoriščamo. Zelo preprost primer modeliranja, s katerim je bralec zelo dobro seznanjen, prikazuje slika 28. Z drugimi besedami bi lahko to isto sliko označili kot diagram napovedanega obnašanja vremena po posameznih dnevih. Čeprav je prikaz razmeroma nezahteven, je mogoče iz njega razbrati veliko informacij, kar je posledica nedvoumnih dogovorov in smiselno uporabljenih relacij. Še nekaj primerov modeliranja prikazuje slika 29, s primeri ustaljenih načinov vizualizacije. Ti diagrami bralcu, ki je seznanjen s predstavljeno domeno, vsebovano informacijo podajajo jasno in razumljivo ter na nedvoumen način. V primerjavi s predhodnim diagramom vremena lahko opazimo, da slednji diagrami obravnavajo druga področja in da je za razumevanje le-teh potrebno razumevanje tako domene same, kot tudi pomen posameznih gradnikov diagrama domene, torej modelov, ki predstavljajo nekaj točno določenega. Jasnost diagrama je odvisna od količine informacij, ki jih posamezni modeli sporočajo. To predstavlja še en zelo pomemben koncept modeliranja – za nedvoumen, jasen in razumljiv prikaz določenega aspekta opisovanega področja lahko izberemo samo tisti nivo detajlov, ki jasno sporoča želeno količino informacije. S pomočjo uporabe strukturiranih nivojev abstrakcije lahko na pregleden način poskrbimo za učinkovit zajem vseh detajlov opisovanega področja.


Slika 28 – Primer modeliranja, ki ga nezavedno uporabljamo ves čas




Slika 29 – Primeri ustaljenih načinov vizualizacije (matematičen in glasbeni zapis ter električna shema)

Na prvi pogled se nam lahko diagram in model zdita enaka, toda se razlikujeta.

*Def. 7* **Model** je uporabljen za abstrakcijo sistema, ki ga predstavlja, in vsebuje elemente, ki so potrebni za opis namembnosti modeliranega sistema. Elementi

določajo posamezne zorne kote namembnosti predmeta, sistema, medsebojnih relacij ipd.

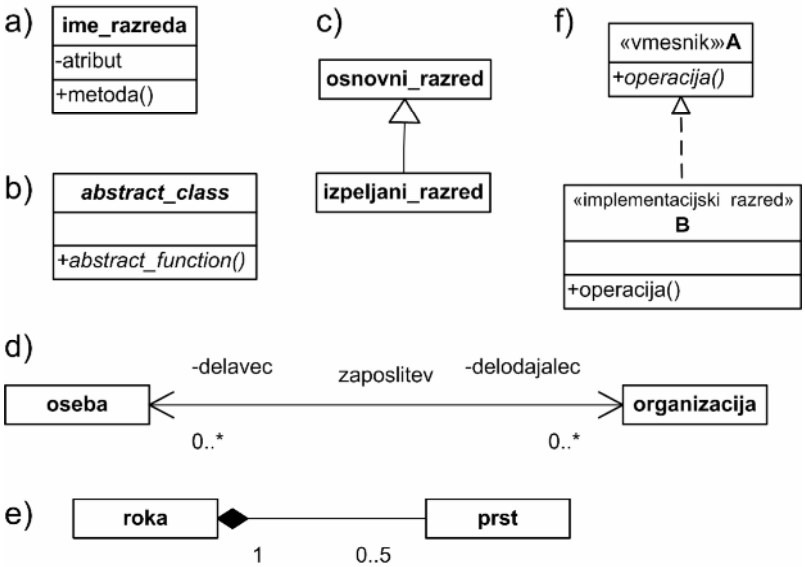
*Def. 8* **Diagram** predstavlja samo specifičen pogled trenutno obravnavane problematike. Predstavlja določen pogled na vse ali samo nekatere dele modela.

Iz tega povsem splošnega uvoda je bralec lahko ugotovil, da je modeliranje pravzaprav zelo zmogljiv koncept zgoščene predstavitve informacij določene domene na način, ki je nedvoumno določen. Model lahko razumemo kot poenostavljeno predstavitev realnega sistema ali dogodka, ki služi njegovi razlagi, ali kot strukturni načrt kompleksnega sistema (ki ga šele želimo zgraditi). Prizadevanje za standardizacijo področja modeliranja predstavlja grafični modelirni jezik UML (*Unified Modeling Language*). Določa pravila in koncepte grafičnega modeliranja na način, ki omogoča enotnejši način komuniciranja podatkov o sistemu. Osnovna ideja jezika se skriva v besedi "poenoten" (*unified*), ki predstavlja idejo, da bi kdor koli (ki je seznanjen z jezikom UML) lahko hitro in enolično razumel relacije med posameznimi sestavnimi deli sistema. Omogoča izmenjavo informacij o sistemu, hitro razjasnitev kompleksnih problemov in scenarijev ter nudi realnejšo predstavitev sistema še pred implementacijo

Jezik UML je standardiziran pod okriljem organizacije OMG [ss3] in predstavlja odgovor na množico načinov modeliranja, ki so v času nastajanja jezika UML z divergenco na področju modeliranju povzročala pravo zmedo. Jezik predstavlja odprt standard grafičnega modeliranja in ga podpirajo institucije, ki ga potrebujejo in se nanj zanašajo. Kot vsak drug jezik tudi UML določa pravila sintakse in uporabe ter – za razliko od metodologije – ne določa postopkov ali potekov načrtovanja. Za obravnavo določenega področja je skupen jezik sicer potreben, a sam zase ne predstavlja metodologije in to velja tudi za jezik UML. Čeprav ga je mogoče uporabiti skupaj z različnimi metodologijami, jezik UML sam zase še ne predstavlja metodologije. Vsebuje dva osnovna tipa diagramov: strukturne in odzivne diagrame. Statična struktura elementov sistema je prikazana s strukturnimi diagrami, medtem ko je njihovo dinamično obnašanje določeno z odzivnimi diagrami. Za modeliranje sistemov, ki so predmet SNSPO, in uporabo modelirnega okolja projekta GME je najpomembnejše razumevanje statičnega strukturnega diagrama, katerega osnovne pojme bomo predstavili v nadaljevanju. Več informacij o samih diagramih in jeziku UML lahko bralec najde na spletni

strani [ss3] v obliki specifikacije standarda in v množici knjig, ki razlagajo njegovo uporabo, npr. [k13] ali [k14].

Razredi predstavljajo jedro objektno orientiranega (OO) opisa sistema, zato statičen diagram razredov predstavlja najpopularnejšo obliko diagrama UML.[21] Na sliki 30 so prikazani samo najosnovnejši koncepti statičnega diagrama razredov, na podlagi katerega bomo bralca uvedli v modeliranje z jezikom UML. Podrobnejše informacije bralec najde v že omenjeni literaturi. Struktura sistema je zgrajena iz posameznih sestavnih delov in ti predstavljajo objekte. Objekti v sebi nosijo informacije in funkcije. Razredi opisujejo različne tipe objektov sistema, diagram razredov pa je uporabljen za prikaz razredov in njihovih medsebojnih relacij. Lastnosti razreda so zajete s pomočjo atributov, njegove funkcije pa z metodami (slika 30a). Simbol pred atributom ali funkcijo predstavlja dostop, ki je lahko javen, privaten ali zaščiten (+-#). Uporabimo lahko tudi abstraktne razrede, ki se od prejšnjih ločijo po tem, da imajo funkcije samo deklarirane, ne pa tudi definirane (b). Iz takšnih razredov ni mogoče neposredno tvoriti objektov, služijo pa definiranju določene funkcionalnosti izpeljanih razredov. Primer izpeljave razreda prikazuje primer c), ki tudi predstavlja eno izmed možnih relacij med razredi (izpeljava). Skladno z OO-pristopom načrtovanja, izpeljani razredi dedujejo vse lastnosti osnovnega razreda, katerim lahko dodajajo še svoje.




Slika 30 – Osnovni koncepti statičnega diagrama razredov jezika UML

[21] Predpostavljamo, da ima bralec osnovno znanje jezika C++ in objektno orientiranega pristopa razvoja sistema, zato se ne bomo spuščali v podrobnosti njune razlage. Podrobne informacije lahko bralec najde npr. v [k15].

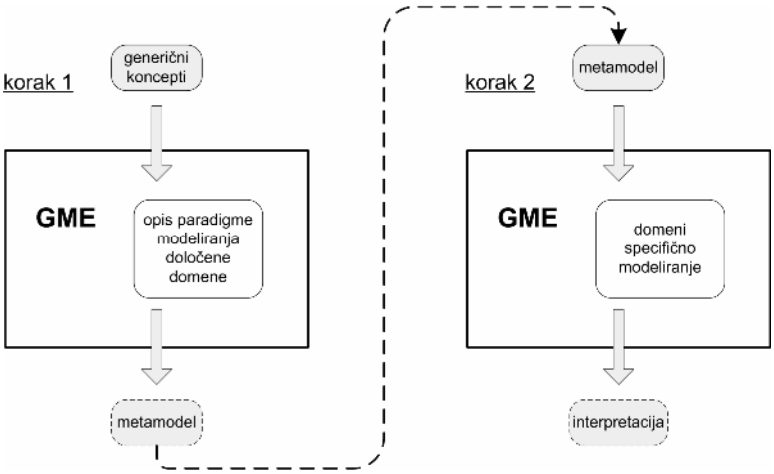
Asociacija je naslednji primer relacije in je prikazana v primeru d). Preberemo jo na naslednji način. Organizacija zaposluje poljubno število (0..*) delavcev, o katerih hrani privatne informacije. Oseba je zaposlena pri poljubnemu številu delodajalcev, o katerih tudi hrani privatne informacije. Bralec, seznanjen s programiranjem, si lahko predstavlja veliko število možnih implementacij podanega scenarija. Sestavljenost je tudi primer relacije in je prikazan v primeru e). Pomeni: objekt roka je sestavljena iz do petih prstov, vsak prst pa pripada samo eni roki. Omenili smo že abstraktne osnovne razrede, ki lahko predstavljajo tudi vmesnike. Vmesnike zato, ker dejanske implementacije funkcionalnosti razreda na vsebujejo, vsebujejo pa njihove deklaracije. Objekt je skladen z določenim vmesnikom, če je razred, ki objekt implementira, izpeljan iz abstraktnega razreda, katerega naloga je samo določanje prototipov funkcij. Implementacijo prikazuje primer f), ki ga beremo tako: razred B implementira vmesnik A (pri tem mora razred B poskrbeti za definicijo metode, ki je deklarirana v razredu A). S tem bomo zaključili predstavitev osnovnih pojmov jezika UML, saj je njegova podrobna razlaga izven konteksta predstavitve našega dela. Uporaba predstavljenih konceptov bo zadostovala za razumevanje mehanizmov modelirnega okolja GME.

Preden se lotimo razlage, kako je uporaba konceptov jezika UML neposredno koristna naši metodologiji SNSPO in modeliranja, moramo razložiti še koncept omejitev. V sam standard jezika UML je vgrajena razširitev, ki omogoča, da je poleg vnosa podatkov o modelu sistema mogoč tudi uporabniško določene vnos pravil gradnje modela. Tako lahko npr. preprečimo nesmiselne relacije med objekti, poskrbimo, da so vnesene vse zahtevane lastnosti objektov ipd. Bralec se verjetno še spomni, da je ravno ta problem tisti, ki nas je pripeljal do opustitve gradnje lastnega modelirnega okolja. Razširitev, ki to podpira, je jezik OCL (*object constraint language*) in ga lahko prevedemo kot jezik za opis omejitev objektov. Jezik OCL je prav tako standardiziran in obravnavan znotraj organizacije OMG [ss3]. Za nas je zelo pomemben zaradi dveh stvari. Prvič, standardizirano se dopolnjuje z jezikom UML in drugič, sledi njegovi namembnosti splošne in prilagodljive uporabe. Če smo do sedaj pokrili modeliranje objektov s stališča predstavitve nazornejšega opisa sistema, lahko z uporabo jezika OCL poskrbimo še za vnos pravil gradnje sistema. V podrobnosti opisa jezika UML se ne bomo spuščali, bralec lahko najde specifikacijo standarda na spletni strani organizacije OMG ter razlago uporabe v že omenjenih knjigah, ki obravnavajo jezik UML.

## 5.3 GME

V prejšnjem poglavju smo predstavili, kako kompleksna je gradnja namenskega modelirnega okolja. Rešitev tega problema predstavlja načrtovalsko okolje, ki je konfigurabilno za širok nabor domen. Takšno načrtovalsko okolje mora nuditi nabor generičnih konceptov, ki so dovolj abstraktni, da so skupni večini domen. Okolje je nato mogoče prilagoditi vsaki novi domeni za neposredno podporo njenega jezika. Razvoj takšnega okolja je sicer zelo zahteven, ga je pa potrebno opraviti samo enkrat in je zato mogoče začetno investicijo razdeliti med več domen.

Generično modelirno okolje (*Generic Modeling Environment, GME*) [p44], [k16], [ss16] predstavlja prilagodljiv nabor orodij za ustvarjanje domeni specifičnega okolja in okolja za programsko sintezo. Prilagoditev okolja je narejena s pomočjo metamodela, ki določa paradigmo modeliranja (modelirni jezik) aplikacijske domene (slika 31, korak 1). Paradigma modeliranja je sestavljena iz informacije o sintaksi, semantiki in predstavitvi domene. Določa koncepte gradnje modelov, relacije, ki lahko obstajajo med temi koncepti, načine povezave in prikaza teh konceptov ter pravila, ki usmerjajo gradnjo modelov. Paradigma modeliranja določa družino modelov, ki jih je mogoče ustvariti s pomočjo nastalega modelirnega okolja. Metamodel, ki določa paradigmo modeliranja, se uporabi za samodejno kreiranje ciljnega, domeni specifičnega okolja (slika 31, korak 2). Pri tem pristopu je zanimivo to, da se metamodel zgradi v okolju GME samem. Kreirano domeni specifično okolje se nato uporabi za gradnjo modelov domene, ki se jih shrani v podatkovno bazo modelov.




Slika 31 – Metamodel, zgrajen v okolju GME, določa domeni specifično modeliranje

Domeni specifični modeli predstavljajo osnovo za samodejno obdelavo zajetih podatkov. Za obdelavo zajetih modelov služi proces interpretacije. Ta predstavlja razširitev modelirnega

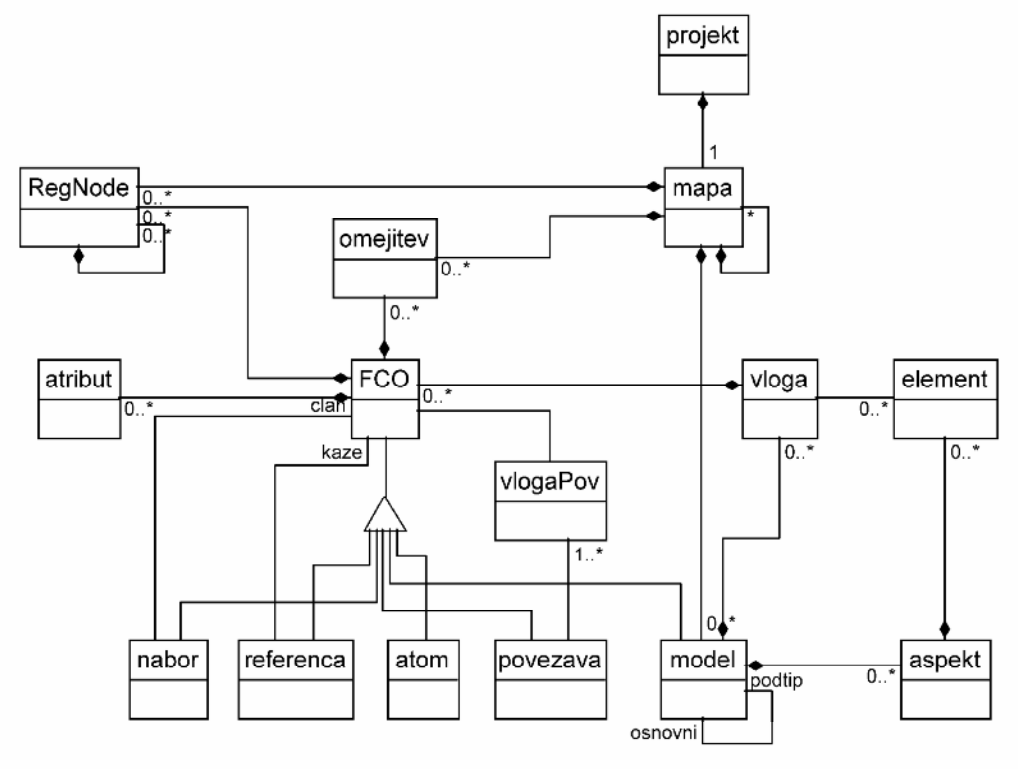
okolja, ki je domeni specifična in nudi mehanizme samodejne obdelave zajetih podatkov, skladno z metodologijo domene. Podporo za interpretacijo je mogoče dograditi v modelirno okolje GME s pomočjo mehanizmov modularne zasnove (Windows moduli COM), kar omogoča tesno integracijo dodatnih (uporabniško določenih) programskih modulov. Ta mehanizem omogoča nadgradnjo obstoječega okolja z lastnimi in domeni specifičnimi razširitvami, kamor sodi tudi interpretacija modela. Ker je programiranje modulov COM zahtevno programersko opravilo, so snovalci okolja GME pripravili programske okvirje C++ in Java, ki dodajanje funkcionalnosti v okolje močno olajšajo. Čeprav smo interpretacijo modulov implementirali s pomočjo okvirjev C++, bi podrobna razlaga razširjanja funkcionalnosti in gradnje domeni specifičnih interpretorjev presegla okvirje te disertacije. Razlago si bralec lahko najde na spletni strani [ss16] in v [k16].

Samo določanje paradigme modeliranja lahko razumemo kot še en modelirni problem in ga je mogoče narediti znotraj okolja GME. Modelirna paradigma, katere namen je konfiguracija okolja GME na način, ki omogoča gradnjo metamodelov, predstavlja sestavni del okolja GME. Modele, zgrajene na osnovi te modelirne paradigme, se nato preko interpretacije samodejno prevede v informacijo, ki služi vnovični konfiguraciji okolja GME. Prvotna paradigma metamodela je bila narejena ročno. Ko pa je bil interpretor metamodela enkrat ustvarjen, je bilo mogoče iz metamodela njegovo paradigmo ustvariti samodejno. Na to lahko gledamo podobno kot na pisanje prevajalnikov C v jeziku C. Paradigma metamodeliranja temelji na jeziku UML. Definicija sintakse je modelirana z diagramom razredov jezika UML, statična semantika pa je določena s pomočjo jezika OCL.

Za modeliranje določene domene je potrebno najprej zgraditi njen metamodel (slika 31). Povedali smo že, da to naredimo z naborom generičnih konceptov, ki so dovolj abstraktni, da predstavljajo osnovo večini domenam. Slika 32 prikazuje model, ki je osnovan na že zgrajenem metamodelu – predstavlja torej model izbrane domene (v tem primeru jezika UML) – in ne predstavlja slike, kot bi jo videli pri gradnji metamodela (primer metamodela je prikazan na slikah 33 in 34). Na osnovi metamodela statičnega diagrama razredov jezika UML je nazorno prikazan model generičnih konceptov, uporabnih za gradnjo lastnega metamodela, istočasno pa prikazuje idejo splošno rekurzivne uporabe okolja GME – tako metamodel kot tudi domeni prilagojeno modelirno okolje (ki predstavlja rezultat metamodela) lahko v okviru okolja GME obravnavamo enovito.

*Projekt* vsebuje eno ali več *map*. *Mape* predstavljajo vsebnike, ki pomagajo organizirati *modele*, podobno kot mape na trdem disku služijo organizaciji datotek. *Mape* vsebujejo *modele* in nove *mape*. *Model*, *povezava*, *atom*, *referenca* in *nabor* predstavljajo objekte *FCO* (*first class objects*) – ti predstavljajo najosnovnejše gradnike okolja GME. *Atomi* so elementarni objekti in ne morejo vsebovati *elementov*. Vsaka vrsta *atoma* ima pripadajočo ikono in vnaprej določen nabor atributov. Njihove vrednosti lahko uporabnik določa sam.

*Modeli* so sestavljeni objekti in lahko vsebujejo *elemente* ter imajo notranjo strukturo. *Element* znotraj modela-vsebnika ima vedno določeno *vlogo*. Paradigma modeliranja določa, katera vrsta *elementov* je dovoljena znotraj posameznega *modela* in *vlogo*, ki jo ima *element*. Uporabnik dobljenega metamodela pa ima nadzor nad tem, katere in koliko *elementov* bo določen *model* vseboval (to je seveda lahko omejeno s specifičnimi omejitvami). Ta relacija vsebovanja ustvari hierarhično razdelitev *modelov*. V primeru da *model* lahko vsebuje isto vrsto *modela* kot vsebovan element, je hierarhija lahko (teoretično) neomejena. Vsak objekt ima lahko največ enega prednika, ki mora biti model. Vsaj en *model* nima svojega prednika; ta predstavlja izhodiščni *model*.




Slika 32 – Modelirni koncepti okolja GME [k16]

*Aspekti* pri celotni predstavitvi predstavljajo predvsem vizualni nadzor. Vsak *model* ima vnaprej določen nabor *aspektov*. Vsak *element* je lahko viden ali skrit znotraj *aspekta*. Vsak *element* ima nabor primarnih *aspektov*, kjer ga je mogoče ustvariti ali izbrisati. Pri naboru *aspektov*, ki jih lahko imajo *modeli* in njihovi *elementi*, ni omejitev. Določiti je mogoče mapiranje, ki določi, kateri *aspekti* od *elementa* so prikazani v katerih *aspektih* od *modela prednika*.

*Povezava* predstavlja najpreprostejšo relacijo med dvema objektoma. Ta je lahko usmerjena ali neusmerjena, lahko vsebuje *atribute* in ima vlogo (*vlogaPov*). Povezava med dvema objektoma je mogoča, če imata objekta v hierarhiji vsebovanja istega prednika in sta vidna v okviru istega *aspekta* (v enem izmed primarnih aspektov povezave). Specifikacija paradigme lahko določa več vrst *povezav*. Določeno je tudi, katere vrste *elementov* lahko sodelujejo v posamezni vrsti *povezave*. *Povezave* so lahko še nadalje omejene z izrecnimi omejitvami, npr. glede njihove številčnosti.

S pomočjo *povezav* so lahko izražene samo relacije med objekti, ki so vsebovani v istem *modelu*. *Izhodiščnega modela* npr. sploh ni mogoče vključiti v *povezavo*. Za druge vrste relacij so uporabljene *reference*. Z njihovo pomočjo je mogoče povezati različne vrste objektov *modelov* iz različnih delov istega nivoja hierarhije *modela* ali celo z različnih nivojev hierarhije *modela*. *Reference* so podobne kazalcem iz objektno orientiranih programskih jezikov. *Referenca* ne predstavlja dejanskega objekta, ampak nanj samo kaže. Znotraj GME mora tudi *referenca* nastopiti kot *element modela*. Tako je osnovana relacija med *modelom*, ki *referenco* vsebuje, in objektom, na katerega *referenca* kaže. *Referenca* lahko kaže na vsak objekt *FCO*, tudi na referenco samo (izjema je *povezava*, na katero ne more kazati). Povezati jih je mogoče enako kot ostale običajne objekte *modelov*. Določena *referenca* lahko kaže na natančno en objekt, medtem ko na en objekt lahko kaže več *referenc*. Če *referenca* ne kaže na noben objekt, je potem to *ničelna referenca*. Namenjena je lahko npr. kasnejši uporabi (ko bo objekt, na katerega kaže, določen).

*Povezave* in *reference* so binarne relacije. Relacije med skupino objektov je mogoče predstaviti z *naborom*. Edina omejitev je ta, da se morajo vsi *elementi nabora* nahajati znotraj istega vsebnika (prednika) in so vidni znotraj istega *aspekta*. Vsa informacija pa ni primerna za grafično predstavitev, zato okolje GME nudi tudi možnost nadgradnje grafičnih objektov s tekstnimi *atributi*. Vsi objekti *FCO* lahko vsebujejo različne *atribute*. Ti so lahko tekst, celo število, realno število, binarna vrednost in oštevilčenje. *Mape* in objekti *FCO* lahko vsebujejo

tudi omejitve, ki omogočajo tekstovni vnos specifičnih pravil in omejitev v jeziku OCL, in parameter *Regnode*, ki je programerske narave in predstavlja način dostopa do objekta med interpretacijo modela.

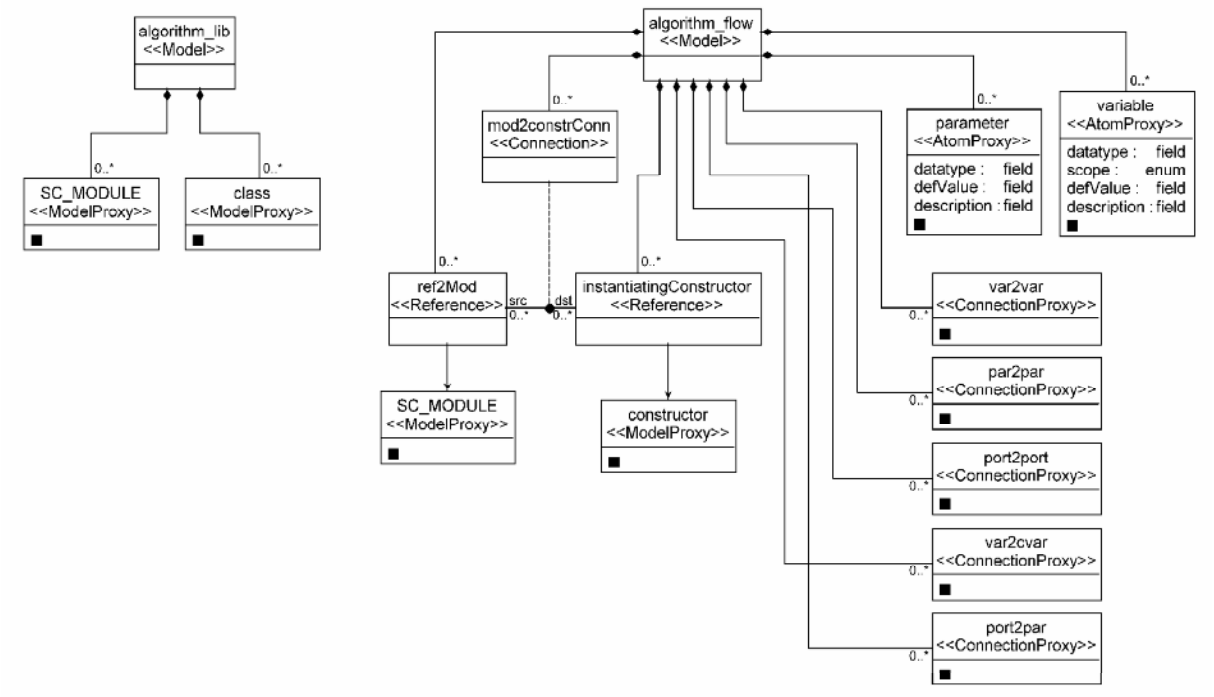
*Mape, objekti FCO (modeli, atomi, nabori, reference* in *povezave), vloge, omejitve* in *aspekti* predstavljajo glavne koncepte definiranja paradigme modeliranja. Z drugimi besedami – modelirni jezik je narejen s tvorbo objektov, ki temeljijo na teh konceptih. Ustrezne konceptualne vzporednice pri programiranju v objektno orientiranem programskem jeziku (npr. Javi) predstavljajo razred, vmesnik, vgrajeni tipi ipd. *Modeli* v okolju GME so podobni razredom v Javi; iz njih je mogoče tvoriti objekte. Ko je okolju GME ustvarjen določen *model*, ta postane tip (razred). Iz njega je mogoče ustvariti poljubno število podtipov ali objektov. Ta koncept omogoča ponovno uporabo in učinkovito obvladovanje *modelov*, saj se vsaka sprememba v tipu samodejno razširi navzdol po hierarhiji. Omogoča tudi tvorbo knjižnic tipov *modelov*, ki jih je mogoče uporabiti v več aplikacijah določene domene.

### *5.3.1 Primer metamodela modeliranja algoritma*

Bralca bomo podrobneje seznanili s konceptom gradnje metamodela na primeru, ki zajame samo določen vidik modeliranja, s katerim se ukvarja naša metodologija SNSPO. Da bi bila razlaga jasna in ne predolga, bomo predstavili samo najpomembnejše koncepte modeliranja algoritma. Zaradi istega razloga v primeru ne bomo uporabili naprednih funkcij okolja GME, kot npr. lastnih dekoratorjev (programskih gradnikov, ki skrbijo za uporabniško določen izris posamezne komponente) ali pa naprednih oblik dedovanja. Oboji resda lahko poenostavijo nekatere kompleksne relacije, zato pa zahtevajo že podrobno poznavanje okolja GME. Podobno kot je to pri programiranju je tudi tu isto nalogo mogoče narediti na več načinov. Predstavili bomo en način izvedbe.

Algoritem želimo predstaviti s smiselnimi pomenskimi podsklopi, ki bodo povezani skladno z njihovo podatkovno odvisnostjo. Podsklope želimo zgraditi in opisati ločeno od celotne predstavitve algoritma, kar nam bo omogočalo zgraditi knjižnico gradnikov algoritma. Na sliki 33 se na vrhu nahajata dva modela `algoritem_lib` (opis podsklopov) in `algorithm_flow` (predstavitev algoritma). Model `algoritem_lib` lahko vsebuje modela `SC_MODULE` in `class`, ki sta pripravljena za opis podsklopov algoritma. Oznaka `ModelProxy` in črn kvadrat spodaj desno označujeta, da gre za ekvivalent modela, katerega definicija se nahaja drugje. Ekvivalent ni enak referenci; ekvivalent zamenjuje določen objekt, medtem ko referenca služi kot kazalec na določen tip objekta (točno na katerega, določi uporabnik

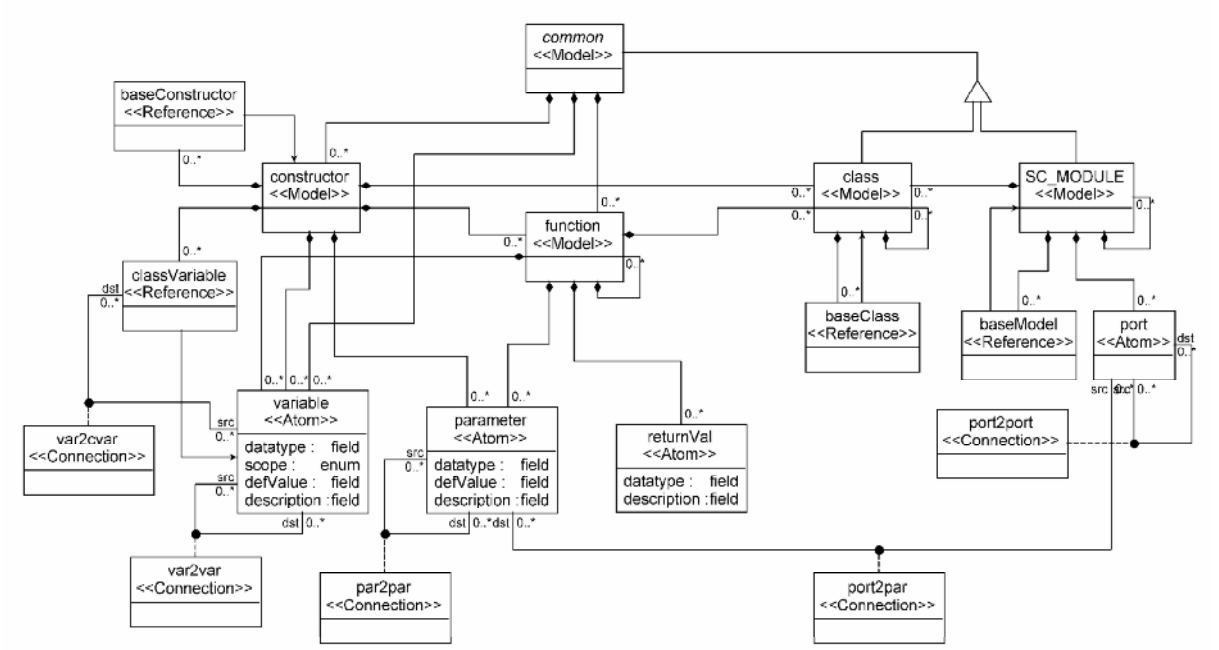
metamodela sam). Model `algorthm_flow` lahko vsebuje parametre (`parameter`), spremenljivke (`variable`), referenco na modele `SC_MODULE` (`ref2Mod`), referenco na konstruktor modela (`instantiatingConstructor`) ter povezave med različnimi objekti. Dovoljene so naslednje povezave: med dvema spremenljivkama (`var2var`), parametroma (`par2par`), portoma (`port2port`), med spremenljivko in spremenljivko razreda (`var2cvar`) ter med portom in parametrom (`port2par`). Večina objektov, ki so dovoljeni v modelu `algorithm_flow`, je predstavljena z ekvivalenti – njihov pomen si bomo podrobneje ogledali na sliki 34. Primer povezave, določene v tem modelu, je povezava med referenco na model (`ref2Mod`) in referenco na konstruktor (`instantiatingConstructor`). Pri tem obe referenci zopet kažeta na modela, ki sta definirana drugje. Povezava med modeloma je določena z odebeljeno piko, ki se nahaja na povezavi med njima. Ta ista pika je tudi črtkano povezana z objektom `connection`, ki določa pomen povezave. Kot bralec lahko vidi, je potrebno določiti prav vse elemente, ko bodo uporabljeni pri modeliranju. Elementi ali povezave, ki niso določene, niso dovoljene. Nadaljnje, natančneje določene omejitve glede povezav ali uporabe objektov je mogoče določiti s pomočjo pravil na osnovi že omenjenega jezika OCL. Teh zaradi preglednosti ne bomo vključili v primer.




Slika 33 – Metamodel algoritma sestavljata dva glavna dela; knjižnični del in potek

Do sedaj smo pri določanju metamodela algoritma uporabili veliko ekvivalentov. Ti služijo predvsem preglednosti metamodela, saj je tako celoten model mogoče zgraditi modularno (po

pomensko zaključenih enotah) in enostavno uporabiti dele modela, ki so opisani drugje v hierarhiji. Slika 34 prikazuje definicije sestavnih delov modela. Razredi (`class`) in moduli jezika SystemC (`SC_MODULE`) imajo veliko skupnih lastnosti, ki pa jih ni potrebno določiti večkrat, ampak lahko uporabimo mehanizem dedovanja. Skupne lastnosti so predstavljene z modelom `common`, te pa se prenesejo (podedujejo) na oba izpeljana razreda. To je prikazano s trikotnikom, pri čemer je njegov vrhnji del povezan z razredom prednikom, njegov spodnji del pa na oba naslednika. Ta model predstavlja tudi najbolj enostavnega od treh načinov dedovanja. V tem primeru nasledniki enostavno podedujejo vse lastnosti prednikov. Model `common` je napisan poševno, kar pomeni, da je abstrakten in da ga ni mogoče uporabiti za tvorbo objektov. Bralec je do sedaj najverjetneje že ugotovil, zakaj smo eno prejšnjih poglavij namenili jeziku UML, saj je razumevanje le-tega ključnega pomena za razumevanje metamodela.



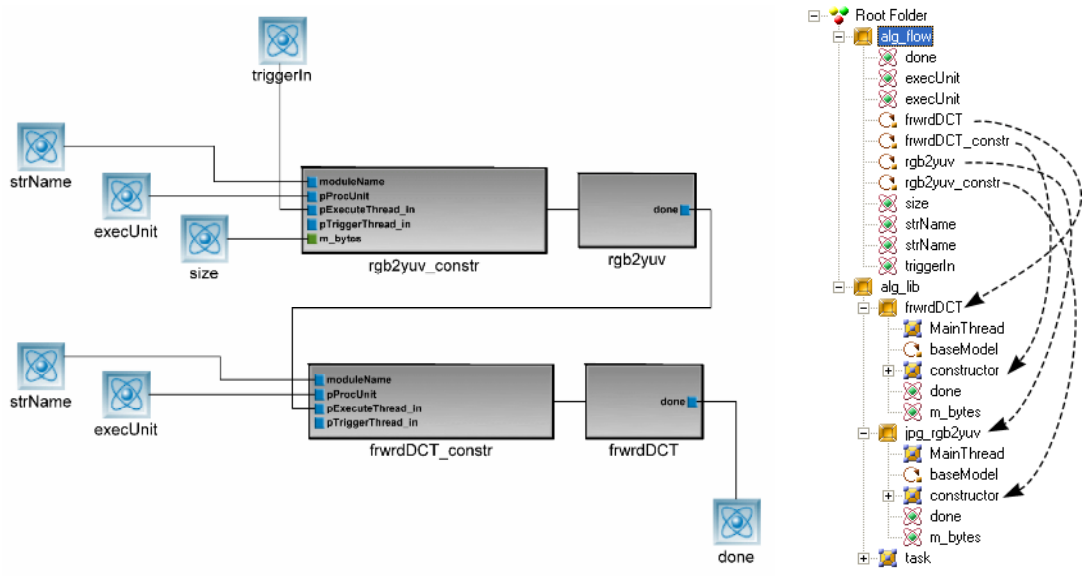

Slika 34 – Metamodel podsklopov algoritma

Skupne lastnosti modelov `class` in `SC_MODULE`, ki so predstavljene z razredom `common`, lahko opišemo z naslednjim besedilom. Skupni model lahko vsebuje funkcije (`function`), spremenljivke (`variable`) in konstruktorje (`constructor`). Funkcije lahko vsebujejo nove funkcije, razrede (`class`), lahko vrnejo vrednost (`returnVal`) in sprejemajo parametre (`parameter`). Parametri so določenega tipa (`datatype`), lahko imajo privzeto vrednost (`defValue`) in opis (`description`). Parametrom podobne so spremenljivke (`variable`), ki

imajo poleg omenjenih lastnosti še obseg uporabe (`scope`). Konstruktorji so malce bolj zapleteni modeli. Za potrebe začetnih nastavitev vrednosti lahko ti vsebujejo referenco na konstruktorja prednika (`baseConstructor reference`) in spremenljivke razreda (`classVariable reference`). Poleg tega lahko vsebujejo še spremenljivke, parametre, funkcije in razrede. Model lahko vsebuje tudi določene povezave med spremenljivkama, parametroma, med spremenljivko in referenco na spremenljivko razreda, s katerimi pa je bil bralec že seznanjen. Bralec, ki je seznanjen s programiranjem, je lahko našel vez med tem opisom in *običajnim* programiranjem. Nenazadnje je to tudi naš namen, saj želimo zgraditi samodejno povezavo med grafičnim modeliranjem in generiranjem spodaj ležeče programske kode.

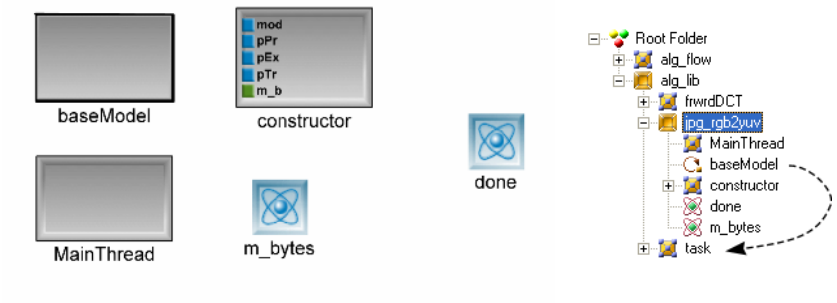
Vse omenjene skupne lastnosti model `class` nadgradi z vsebovanjem samega sebe in referenco na razred prednika (`baseClass reference`). Model `SC_MODULE` pa doda referenco na `SC_MODULE` prednika, vsebnost modula `class` in porte (`port`). To predstavlja učinkovit mehanizem ponovne uporabe in omogoča boljši nadzor nad posameznimi lastnostmi modela.

Okolje GME po meri prilagodimo tako, da predstavljeni metamodel interpretiramo, tj. prevedemo v obliko, primerno za konfiguracijo okolja GME (skladno s sliko 31). V metamodelu poteka algoritma (desna stran slike 33) smo določili, katere objekte lahko vsebuje potek algoritma (`algorithm_flow`). Dovoljene objekte lahko poljubno dodajamo, povezujemo in nastavljamo pripravljene parametre. Pri tem nam okolje dopušča samo to, kar smo z metamodelom dovolili. Na sliki 35 je prikazan potek algoritma, ki vsebuje dve referenci na model tipa `SC_MODULE` (to sta `rgb2yuv` in `frwrdDCT`). V drevesni strukturi smo referencam dodali črtkane puščice, da je mogoče enostavneje najti objekt, na katerega ta kaže. V prikazu modela se reference ločijo od ostalih objektov po odebeljeni obrobi, v drevesni strukturi pa po simbolu . Poleg teh dveh referenc sta še referenci na konstruktorja (`rgb2yuv_constr` in `frwrdDCT_constr`), parametri (`triggerIn`, `strName`, `execUnit` in `done`). Vse povezave med objekti so skladne z določenimi znotraj metamodela poteka algoritma. V primeru, da bi npr. hoteli narediti povezavo med parametrom in spremenljivko, bi okolje vrnila napako z razlago, da tip povezave v metamodelu ni določen. Če bi v metamodel vnesli tudi določene omejitve v obliki jezika OCL, bi se te omejitve tudi sprotno preverjale.




Slika 35 – Model algoritma in njegova drevesna struktura

Slika 36 prikazuje, kako smo zgradili podsklop `rgb2yuv`, katerega smo v prejšnjem modelu (slika 35) uporabili za referenco. Ta model je izpeljan iz osnovnega iz objekta `task`, zato vsebuje referenco, ki kaže nanj. Skladno z metamodelom vsebuje konstruktor (`constructor`), port (`done`), spremenljivko (`m_bytes`) in funkcijo (`MainThread`). Povezave tu niso potrebne.




Slika 36 – Model podsklopa algoritma (`rgb2yuv`) in njegova drevesna struktura

Model konstruktorja podsklopa `rgb2yuv` algoritma je prikazan na sliki 37. V konstruktorju se nahaja referenca na razred, iz katerega je podsklop `rgb2yuv` izpeljan. Ta mehanizem je podoben mehanizmu programiranja (npr. določitev začetnih vrednost konstruktorja razreda prednika v jeziku C++). Poleg reference na konstruktor razreda prednika se nahajajo še parametri (`moduleName`, `pProcUnit`, `pExecuteThread_in` in `pTriggerThread_in`) in referenca na spremenljivko razreda, kateremu konstruktor pripada (`m_bytes`). Vse to smo seveda določili z metamodelom podsklopov (slika 34). Na drevesni strukturi lahko tudi vidimo strukturo modela `task`, ki predstavlja osnovni razred podsklopoma algoritma `frwrdDCT` in `jpg_rgb2yuv`.


Slika 37 – Model konstruktorja podsklopa algoritma (`rgb2yuv`) in njegova drevesna struktura

Vse te relacije metamodela so sprva lahko nejasne ali celo težko razumljive. Upoštevati je potrebno, da do takšnega metamodela pridemo postopoma skozi njegovo uporabo (tj. z uporabo aplikaciji prilagojenega modelirnega okolja), kjer postopoma ugotavljamo lastnosti, ki bi jih želeli modelu dodati. Kljub morebitni kompleksnosti metamodeliranja pa je takšna gradnja vseeno neprimerljivo učinkovitejša od gradnje lastnega modelirnega okolja. To je tudi razlog za sosledje predstavljenih tem v tej doktorski disertaciji. Brez lastnih izkušenj o gradnji modelirnega okolja bi zaradi začetne kompleksnosti tudi sami omahovali glede njegove ustreznosti, ko bi se prvič srečali z generičnim modelirnim okoljem (okoljem GME).

Pomembno je tudi dejstvo, da prikazano modeliranje (metamodeliranje) predstavlja gradnjo domeni prilagojenega okolja in je nekaj, česar končnemu uporabniku ne bo potrebno razumeti. Dodatno dejstvo, ki morda rahlo otežuje celotno sliko, je ravno generičnost okolja, ki je dovolj splošno, da si je mogoče vse aspekte modeliranja prilagoditi po meri. S tem smo mislili predvsem gradnjo modulov COM, z uporabo katerih bi lahko zelo elegantno zajeli in prikazali določene aspekte modela, ki smo jih zaenkrat obdelovali v povsem generični obliki. Npr. z uporabo lastnih dekoratorjev bi lahko koncept razredov obravnavali veliko bolj elegantno. To pa ponovno zahteva razmeroma veliko programerskega truda, zato se je smiselno ukvarjati s tem, ko bo okvir metodologije in modeliranj trdneje določen. S tem poglavjem smo namreč postopoma prišli do dela, kjer moramo napisati, da smo postavili določene okvirje in nakazali smernice prihodnjega dela, saj to presega zastavljene okvirje te doktorske disertacije.

### *5.3.2 Okvirji*

V poglavju 4.5 smo predstavili vlogo okvirjev v postopku SNSPO. Bralec bi se lahko vprašal, kakšna je povezava med modeliranjem in okvirji, saj slednjih v prejšnjem poglavju v primeru metamodela modeliranja algoritma nismo omenili.

Pri predstavitvi modelirnega okolja GME smo omenili koncept prilagoditve okolja. Prilagoditev omogoča posebna, v GME vgrajena, interpretacija, ki prestavlja mehanizem pretvorbe modela GME v podatke, na podlagi katerih je okolje mogoče prilagoditi aplikaciji. Na koncu vsakega modeliranja (in ne samo pri gradnji metamodela) sledi interpretacija, ki vrne uporaben opis sistema. Iz tega vidika je modeliranje v GME pravzaprav samo mehanizem učinkovitega načina opisa sistema. Npr. interpretacija metamodela je vrnila podatke za prilagoditev modelirnega okolja našim potrebam in kot taka predstavlja sestavni del modelirnega okolja GME. Vse naslednje interpretacije pa so uporabniško določene; te interpretirajo gradnike modela, ki je prilagojen aplikaciji.

Interpretacija pride na vrsto na koncu modeliranja, ko smo v model sistema vključili vse želene podatke. Npr. v postopku interpretacije modela, ki podpira našo metodologijo, se iz modela sistema tvori koda SystemC. Koda se tvori po določenih pravilih, na podlagi pomena posameznih gradnikov. Pomen in pravila so določeni z okvirji. Okvirji določajo, kaj bo vstavitev posameznega elementa v model sistema pomenila pri gradnji programske kode. Pri tem je uporabljen koncept knjižnic – interpretiramo lahko samo tiste gradnike, za katere imamo pripravljena pravila. Če objekte nadgrajujemo ali izpeljujemo po določenih pravilih, jih je tudi možno samodejno interpretirati. En tak primer je npr. koncept izpeljanih in prilagojenih razredov; mogoče ga je narediti tako v okviru programske kode kakor tudi grafično z modeliranjem. Okvirji morajo torej zajeti paradigmo, sintakso in semantiko – njihovo grafično predstavitev predstavljajo gradniki modelirnega okolja GME, programsko pa so predstavljeni z generirano kodo.

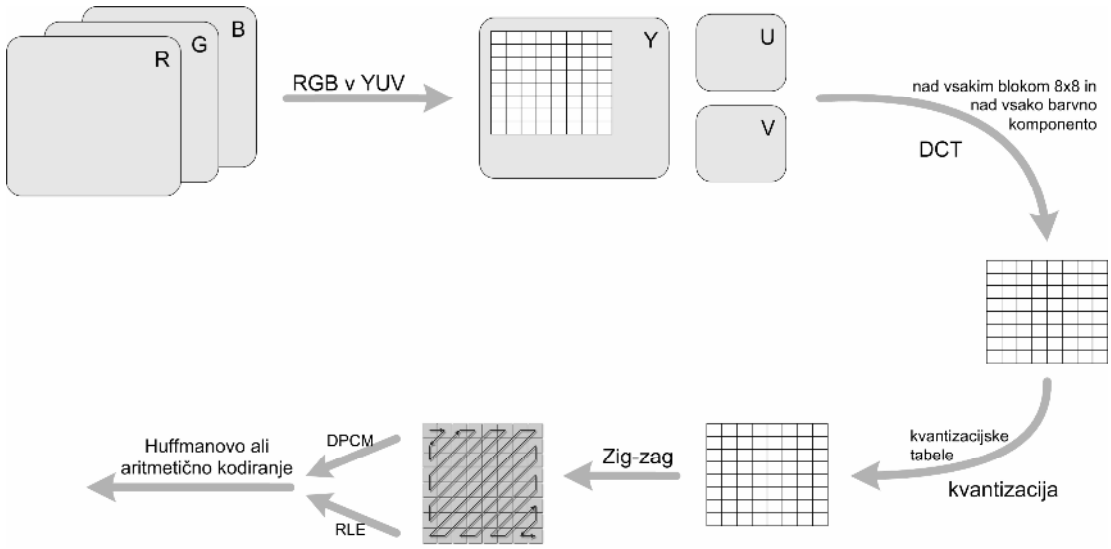
# 6 Prikaz uporabe metodologije SNSPO

Koncepte predlagane metodologije SNSPO bomo prikazali na primeru študije kodirnega sistema JPEG na nivoju sistema. Da bi primer ohranili jasen in pregleden, se bomo osredotočili samo na sistemski del implementacije. Implementacijske podrobnosti, ki izvirajo iz sistemskih odločitev, niso predmet te študije in si jih bralec lahko ogleda npr. v [k4]. Kolikor vemo, je naš visokonivojski način obravnave edinstven in se v tem pogledu naša študija razlikuje od ostalih, zato tudi ni mogoče narediti neposredne primerjave s podobnimi rezultati sorodnega raziskovanja. Nenazadnje pa bi s celotno implementacijo lahko celo zameglili idejo, ki jo želimo predstaviti. Pokazali bomo namreč, da lahko s pomočjo uporabe naše metodologije in koncepta visokonivojskega SNSPO *hitro* ter *učinkovito* pridemo do rezultatov, ki so primerljivi z rezultati, objavljenimi v literaturi (npr. [k4]), ki temeljijo na mnogo bolj podrobnem opisu sistema ali pa dokončni implementaciji.

Slika 38 prikazuje blokovni diagram postopka kodiranja JPEG. Da bi bralec lahko razumel možnosti sistemskega načrtovanja, ki jih bomo predstavili v nadaljevanju, mora biti najprej seznanjen s potekom obdelave podatkov.

Sliko najprej pretvorimo iz barvnega prostora RGB v barvni prostor YUV. To je potrebno zato, ker je v barvnem prostoru YUV stiskanje mnogo bolj učinkovito. Zaradi manjše občutljivosti človeškega očesa na barvni komponenti je običajno, da se barvni ravnini U in V podvzorčita. Nad bloki točk velikosti 8 x 8 se nato opravi diskretna kosinusna transformacija (*discrete cosine transform, DCT*), ki prostorske komponente točk pretvori v frekvenčne. Pri tem višje frekvence pomenijo bolj fine razlike med točkami in jih je na račun kakovosti slike mogoče upoštevati samo delno ali pa povsem neupoštevati. To se opravi v koraku kvantizacije, ki naredi selekcijo informacije, ki jo želimo ohraniti. Informacijo se iz posameznih blokov 8 x 8 nato prebere na poseben *zig-zag* način. Enosmerne komponente (točke (1,1) blokov 8 x 8) se zapiše s pomočjo kodiranja DPCM (*differential pulse code modulation*), izmenične (vse ostale v bloku 8 x 8) pa s kodiranjem RLE (*run length encoding*). V opisanem postopku je edina izgubna transformacija kvantizacija, vse ostale operacije pa so, če zanemarimo numerično napako, brezizgubne. V postopku smo zaradi

preglednosti izpustili pripravo in obdelavo datoteke JFIF (*JPEG File Interchange Format*), s katero se lahko bralec (skupaj s celotno programsko implementacijo) spozna na strani [ss4]. Prav tako smo izpustili manj pomembne podsklope, kot npr. razširitev robov ali postopek podvzorčenja; podrobnejši vpogled v algoritem JPEG je bralcu razložen v [k4].

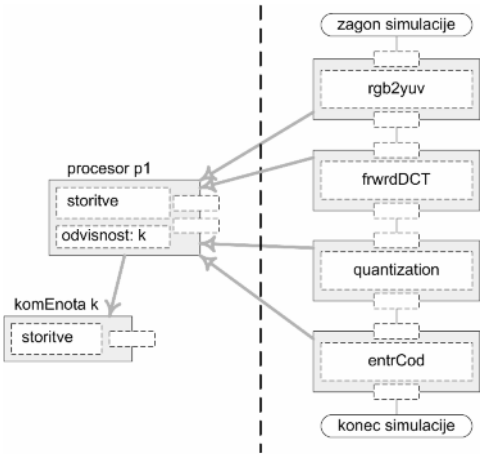



Slika 38 – Blokovni diagram kodiranja JPEG

Enostavnost apliciranja metodologije visokonivojskega raziskovanja načrtovalskega prostora bomo prikazali na primeru implementacije algoritma JPEG. Najprej bomo algoritem implementirali na enem procesorju, ko pa bomo raziskali možnosti pohitritve, bomo prikazali še implementacijo s tremi procesorji. Implementaciji predstavljata samo dve izmed mnogih načrtovalski možnosti, ki bi jih načrtovalec želel preizkusiti v primeru načrtovanja praktično uporabnega sistema. Namen te študije je prikazati, kako lahko našo metodologijo SNSPO uporabimo za hitro in učinkovito preverjanje različnih načrtovalskih možnosti, tj. ovrednotenje načrtovalskega prostora. Čeprav smo v obeh primerih uporabili procesorja, to ne pomeni, da je naša metodologija uporabna samo za obravnavo PO, ampak predstavlja zgolj primer, ki bi ga zlahka lahko dopolnili z uporabniško določenimi strojnimi elementi.

Implementacija algoritma kodiranja JPEG z enim samim procesorjem predstavlja najenostavnejši in največkrat uporabljeni primer. V tem primeru si vsi koraki obdelave sledijo zaporedno. Ko procesor konča obdelavo podatkovnega bloka skladno s predhodnim delom algoritma, lahko prične z obdelavo podatkovnega bloka z naslednjim delom algoritma. Slika 39 prikazuje model kodirnega sistema na visokem nivoju abstrakcije. Skladno s sliko 14 sta nivoja abstrakcije na sliki 39 v domeni izračunavanja na nivoju *prototipov sistemskih funkcij* in v domeni podatkov na nivoju *blokov podatkov*. Poleg omenjenih visokih nivojev

abstrakcije slika vsebuje še ostale komponente predstavljene metodologije SNSPO. Bralec lahko opazi delitev na funkcionalnost (desno) in arhitekturo (levo) ter komponentno načrtovanje (delitev na podsklope algoritma in arhitekture). Imena blokov (delov funkcionalnosti), ki smo jih uporabili pri programiranju, izvirajo iz vloge, ki jo ima ta blok: pretvorba iz barvnega prostora RGB v YUV (`rgb2yuv`), transformacija DCT iz barvnega v frekvenčni prostor (`frwrdDCT`), kvantizacija elementov (`quantization`) in entropijsko kodiranje (`entrCod`).




Slika 39 – Model kodirnega sistema JPEG z enim procesorjem

V primeru implementacije kodirnega algoritma JPEG z enim procesorjem vse storitve izvajanja algoritma prevzame procesor `p1` (slika 39). Ta je naprej odvisen od komunikacijske enote `k`, ki nudi procesorju komunikacijske storitve. Tu lahko vidimo nakazano prednost uporabe okvirjev in vmesnikov, saj lahko npr. zamenjamo komunikacijsko enoto in preverimo, kako vpliva na delovanje sistema, ne da bi bilo potrebno spreminjati kar koli drugega v sistemu.

Koncept visokonivojskega opisa algoritma si poglejmo na primeru transformacije DCT. Transformacijo prostorskih komponent bloka 8 x 8 v frekvenčni prostor prikazuje enačba (9), medtem ko enačba (10) predstavlja obratno transformacijo. Enačbi predstavljata t. i. dvodimenzionalno diskretno kosinusno transformacijo (2-D DCT) oz. dvodimenzionalno inverzno diskretno kosinusno transformacijo (2-D IDCT). Kompleksnost izračunavanja je mogoče zmanjšati tako, da opravimo najprej transformacijo nad vrsticami bloka 8 x 8 in nato še nad stolpci. Enačba (11) predstavlja 1-D DCT, enačba (12) pa 1-D IDCT. V literaturi lahko najdemo algoritme, ki omenjene transformacije še nadalje prilagodijo SO, tako da število

množenj in seštevanj zmanjšajo na minimum. Uporabili bomo algoritem [p45], ki za 1-D DCT zahteva 8 množenj in 20 seštevanj.

$$F(u,v) = \frac{1}{4}C(u)C(v)\left[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \tag{9}$$

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right] \tag{10}$$

$$F(u) = \frac{1}{2}C(u)\sum_{x=0}^{7} f(x)\cos\frac{(2x+1)u\pi}{16} \tag{11}$$

$$f(x) = \frac{1}{2}\sum_{u=0}^{7} C(u)F(u)\cos\frac{(2x+1)u\pi}{16} \tag{12}$$

$$C(u), C(v) = \begin{cases} 1/\sqrt{2} & u,v = 0 \\ 1 & u,v \neq 0 \end{cases} \tag{13}$$

V klasičnem pristopu bi sledil opis algoritma DCT v enem izmed programskih jezikov, v okviru naše metodologije pa bomo zajeli samo njegove visokonivojske lastnosti. Model sistema lahko zgradimo veliko hitreje in enostavneje, pri tem pa izkoristimo vse prednosti, ki nam jih nudi uporaba višjih nivojev abstrakcije (poglavje 4.4.3).

```
void jpg_frwrdDCT::MainThread()
{
    // celotno velikost bloka obdelave (m_bytes) razdelimo v
    // 64B (8x8) in naredimo dvoprehodno 1-D DCT
    // horizontalni prehod: branje 8B, obdelava, zapis 8B
    // vertikalni prehod: branje 64B, obdelava, zapis 64B

    int blocks = ROUND_UP( m_bytes, 64 );

    if( m_bytes % 64 )
        cout << sc_time_stamp()
            << " input size to frwrdDCT is not 8x8 aligned\n";

    for( int i = 0; i < blocks; i++ )
    {
        // prehod 0 - horizontalni, prehod 1 - vertikalni
        for( int pass = 0; pass < 2; pass++ )
        {
            for( int line = 0; line < 8; line++ )
            {
                m_pExecUnit->GetData(this, (pass==0)?8:32, false);

                // 11 množenj, 20 seštevanj
                m_pExecUnit->Mult( 11 );
                m_pExecUnit->Add( 20 );

                m_pExecUnit->WriteData(this, (pass==0)?8:32, false);
            }
        }
    }
}
```

Slika 40 – Primer kode visokonivojskega opisa algoritma DCT

Slika 40 prikazuje primer programske kode, ki zajame samo visokonivojske aspekte predstavljene transformacije DCT. Obnovimo, kar smo zajeli s kodo. Transformacijo DCT je mogoče narediti samo na bloku podatkov 8 x 8, če je algoritmu podan blok podatkov, ki ni

poravnan na blok 8 x 8, algoritem na to opozori. Število blokov določa število transformacij, ki jih je potrebno narediti (zunanja zanka `for`). Znotraj vsakega bloka naredimo najprej prehod preko vseh vrstic (srednja zanka `for`, `pass=0`) in nato še preko stolpcev (`pass=1`). Tako vrstic kot stolpcev je 8 (notranja zanka `for`). Za vsako zanko potrebujemo 11 množenj (`Mult(11)`) in 20 seštevanj (`Add(20)`). Z implementacijo množenja in seštevanja se nam ni treba ukvarjati tu, saj je to določeno v izvršilni enoti (`m_pExecUnit`). Le-to lahko enostavno zamenjamo z drugo (ki implementira isti vmesnik) in bomo dobili nove rezultate izvajanja. Upoštevali smo tudi dostop do točk bloka, ki se nahajajo v spominu; če beremo točke vrstice, se podatki v spominu nahajajo zaporedno, če pa beremo točke stolpca, so podatki v spominu razkropljeni. To pomeni, da je v primeru 32-bitnega podatkovnega vodila samo prvi bajt tisti, ki ga želimo. Kako se dopolnjujeta funkcionalni in arhitekturni opis sistema, si lahko bralec ogleda na primeru opisa dela arhitekturne komponente, prikazanega na sliki 41.

V zelo kratko kodo smo strnili veliko aspektov izvajanja algoritma. Pri tem smo zajeli aspekte, ki nas zanimajo na tem nivoju abstrakcije, ne da bi se nam bilo potrebno poglabljati v implementacijske podrobnosti, ki na tem nivoju znanja o sistemu niti niso potrebne. Že pri tem enostavnem primeru se lahko namreč zgodi, da ugotovimo, da kakršna koli programska implementacija transformacije DCT ne zadovolji naših zahtev. V tem primeru se je potrebno obrniti na strojno implementacijo, trud, ki bi ga vložili v nizkonivojski programski opis, pa bi bil v tem primeru povsem odveč.

```
void processor::Mult( unsigned int nOfMult )
{
    wait( nOfMult * 100, SC_NS );
}
```

Slika 41 – Primer kode visokonivojskega opisa množenja

Na podoben način smo opisali vse dele algoritma kodiranja JPEG in tudi arhitekturne komponente ter dobili model sistema, ki smo ga prevedli in dobili simulacijsko izvršilni model. Opomnimo naj, da tu nismo prikazali podpornih oz. knjižničnih gradnikov gradnje modela, saj ti niso del algoritma. Ti so zajeti v sistemski knjižnici za podporo metodologije SNSPO in so predstavljeni v poglavjih 4.5 in 4.6.

Na osnovi izvajanja dobljenega modela smo prišli do rezultatov, ki so predstavljeni v tabeli 2 in sliki 42. Zgornji del tabele 2 prikazuje čase izvajanja obeh arhitekturnih delov sistema (`p1` in `komEnote`), spodnji del tabele pa opisuje čase izvajanja delov algoritma (funkcionalnost) na teh arhitekturnih sredstvih. Časi izvajanja delov algoritma so opisani kot

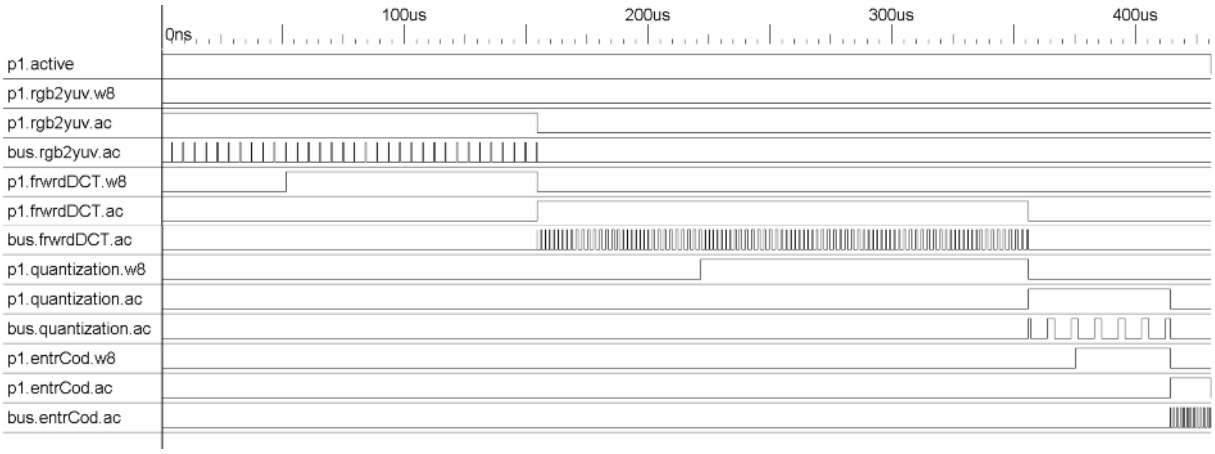
relativni del $T_{IAS}$ (čas izvajanja arhitekturnega sredstva). Iz tega lahko hitro ugotovimo, kateri deli algoritma so zahtevnejši ter kot taki najprimernejši kandidati za pohitritev. Časi čakanja predstavljajo čas, ko je določen del algoritma že imel na voljo vse potrebne podatke, obdelava pa se ni začela, ker je bil procesor še zaposlen z obdelavo predhodne stopnje. Uporabili smo namreč programsko shemo obdelave podatkov, pri kateri je procesor naenkrat obdelal tretjino zahtevanega bloka podatkov in jih obdelane vrnil nazaj v spomin. Če bi bila na voljo dodatna strojna komponenta, bi ta lahko že takoj pričela z nadaljnjo obdelavo. Ker te ni bilo, je ta čas predstavljal "mrtvi" čas, tj. čas čakanja.

| **Arhitekturno sredstvo** | **p1** | | **komEnota** | |
|---|---|---|---|---|
| $T_{IAS}$† [ns] | 430 680 | | 93 240 | |
| **Funkcionalnost** [%$T_{IAS}$] | čas izvajanja | čas čakanja | čas izvajanja | čas čakanja |
| rgb2yuv | 35.9 | 0 | 12.7 | 0 |
| frwrdDCT | 46.8 | 23.9 | 61.8 | 0 |
| quantization | 13.5 | 31.2 | 18.5 | 0 |
| entrCod | 3.8 | 9.0 | 7.0 | 0 |
| skupaj [ns] | 430 680 | | | |

† čas izvajanja arhitekturnega sredstva

Tabela 2 – Časi izvajanja podsklopov algoritma JPEG z enim procesorjem

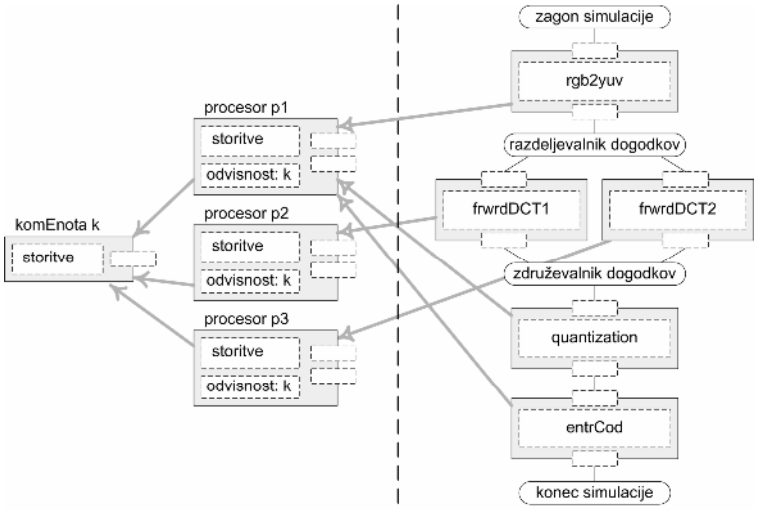
Časovne sledi izvajanja (slika 42) dopolnjujejo podatke v tabeli 2. V slike je lepo vidno, kdaj je del algoritma aktiven (okrajšava *ac*) in kdaj določen algoritem čaka (okrajšava *w8*).




Slika 42 – Časovne sledi izvajanja algoritma JPEG z enim procesorjem

Sedaj se lahko osredotočimo na načrtovalske aspekte. Na osnovi pridobljenih informacij o kodirnem sistemu JPEG ugotovimo, da je računsko najbolj zahteven del algoritma transformacija DCT. Da bi lahko izkoristili prednosti razpoložljive strojne opreme, moramo najprej proučiti možnosti sočasne obdelave, ki jo omogoča algoritem sam. Stopnja vzporedne obdelave podatkov je v končni implementaciji lahko kvečjemu manjša od največje možne, ki

jo omogoča algoritem. Po podrobnem proučevanju algoritma kodiranja JPEG (slika 38) ugotovimo, da je to kodiranje izvrsten primer za študijo vzporedne obdelave podatkov. Pri pretvorbi iz barvnega prostora RGB v YUV je najmanjša enota obdelave ena sama točka, ki jo iz prvega v drug prostor pretvorimo s pomočjo transformacijske matrike. To pomeni, da bi z dovolj SO pretvorbo naredili v enem samem koraku. Dejanska implementacija upošteva seveda še ostale dejavnike in zato predstavlja uravnotežen načrtovalskih kompromis. Ko so komponente barvnega prostora YUV določene, jih lahko obdelujemo ločeno druga od druge. Transformacija DCT postavlja omejitev glede velikosti bloka obdelave, ki mora biti velikosti 8 x 8 točk (ene same barvne ravnine). Tu imamo torej tudi veliko možnosti sočasne obdelave. Podobno je tudi v ostalih korakih algoritma, zato bomo z možnostmi vzporedne obdelave na tem mestu zaključili.




Slika 43 – Model kodirnega sistema JPEG s tremi procesorji

Odločili smo se, da bomo implementirali različico cevovodne strukture, pri kateri bo prvi procesor (`p1`) poskrbel za računsko manj zahtevne dele algoritma, transformacijo DCT pa bosta prevzela dodatna procesorja (`p2` in `p3`). Povedano prikazuje slika 43, ki vsebuje še dva elementa, ki skrbita za pravilen potek obdelave signalov. `Razdeljevalnik dogodkov` krožno razdeljuje opravila (*round Robin scheduling*), `združevalnik dogodkov` pa združuje sporočila procesorjev `p2` in `p3` in jih poda naprej procesorju `p1`. Bralec lahko opazi, da so vsi trije procesorji odvisni od storitev komunikacijske enote. Tako smo modelirali skupno podatkovno vodilo in skupen spomin.

Rezultati simulacije drugega modela so prikazani v tabeli 3 in sliki 44. Čas celotne obdelave iste količine podatkov se je skrajšal za približno 40 %. Na sliki 44 je viden potek

obdelave podatkov, ki pa je v tem primeru že malce bolj zapleten. Najprej je procesor `p1` obdelal tretjino podatkov, katere je prevzel procesor `p2`. Procesor `p1` je nato nadaljeval z obdelavo druge tretjine podatkov, jih predal procesorju `p3`, pričel z obdelavo tretje tretjine podatkov, medtem pa je procesor `p1` že zaključil z obdelavo prvega bloka in čakal (*p2.active*). Ko je procesor `p1` pripravil še zadnji del podatkov prvega dela algoritma (`rgb2yuv`), je te procesor `p2` lahko takoj sprejel, procesor `p1` pa je nadaljeval z obdelavo prve tretjine podatkov tretjega dela algoritma (`quantization`), ki so ga že čakali. Ko je `p1` končal z obdelavo, procesor `p3` še ni dokončal druge tretjine podatkov drugega dela algoritma (`frwrdDCT`), zato se je procesor `p1` lotil obdelave prvega dela podatkov zadnjega dela algoritma (`entrCod`). Medtem je `p2` že zaključil z delom, zato je druga tretjina podatkov zadnjega dela algoritma morala čakati. To je `p1` zaključil takoj in čakal, da `p2` zaključi še zadnjo tretjino podatkov drugega dela algoritma (`frwrdDCT`). Ko so bili ti na voljo, je `p1` opravil še tretji in zadnji del obdelave zadnje tretjine podatkov.

| **Arhitekturno sredstvo** | **p1** | | **p2** | | **p3** | | **komEnota** | |
|---|---|---|---|---|---|---|---|---|
| $T_{IAS}$[†] [ns] | 233 460 | | 148 880 | | 75 100 | | 93 240 | |
| **Funkcionalnost** [%TIAS] | čas izvajanja | čas čakanja | čas izvajanja | čas čakanja | čas izvajanja | čas čakanja | čas izvajanja | čas čakanja |
| rgb2yuv | 66.7 | 0 | - | - | - | - | 12.7 | 1.4 |
| frwrdDCT | - | - | 100 | 0 | 100 | 0 | 41.2/20.6 | 15.5/8.5 |
| quantization | 24.9 | 17.7 | - | - | - | - | 18.5 | 0 |
| entrCod | 8.4 | 0 | - | - | - | - | 7.0 | 3.32 |
| skupaj [ns] | 261 660 | | | | | | | |

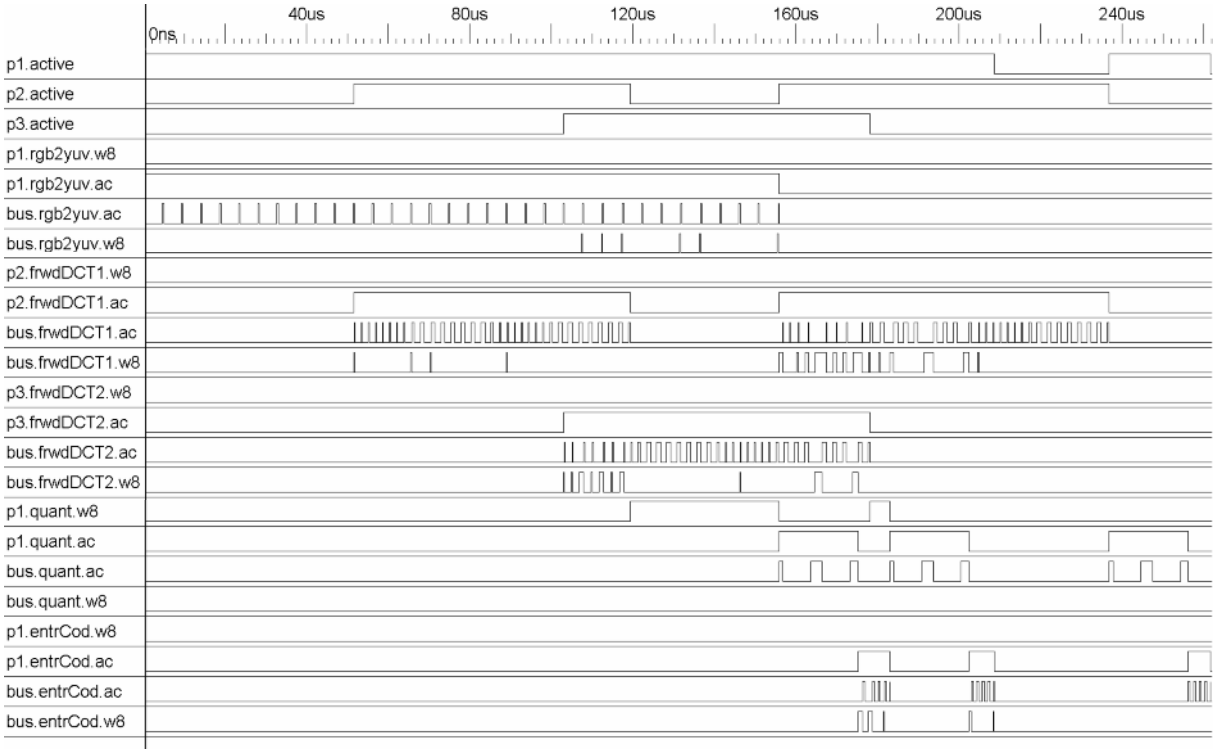
[†] čas izvajanja arhitekturnega sredstva

Tabela 3 – Časi izvajanja podsklopov algoritma JPEG s tremi procesorji

Nekateri časi tabele 3 potrebujejo dodatno razlago. $T_{IAS}$ procesorja `p2` ni natančno dvakraten čas $T_{IAS}$ procesorja `p3` (čeprav `p2` prevzame natanko dvakrat več podatkov kot `p3`), ker je čas obdelave odvisen tudi od zasedenosti komunikacijske enote `k` (spomnimo se, da so procesorji odvisni od uslug komunikacijske enote, slika 43). Stolpci komunikacijske enote imajo pri drugi stopnji obdelave podatkov (`frwrdDCT`) dva časa; prvi pripada `p2`, drugi pa `p3`.

Kaj smo ugotovili? Prvič: že tako zelo enostaven primer je razmeroma težko analizirati, če v sistem vpeljemo vzporedno obdelavo podatkov, pri tem pa smo zastoje, ki so nastali zaradi zasedenosti skupnega vodila, samo omenili. Drugič: čeprav smo opisali sistem visokonivojsko, nam je opis sistema vrnil zelo uporaben nabor informacij, ki so nam učinkovito služile za nadaljnji razvoj sistema. Tretjič: kljub preprostim specifikacijam glede izvajanja smo ugotovili, da bi se izvajanje zaključilo prej, če bi uporabili naprednejše tehnike

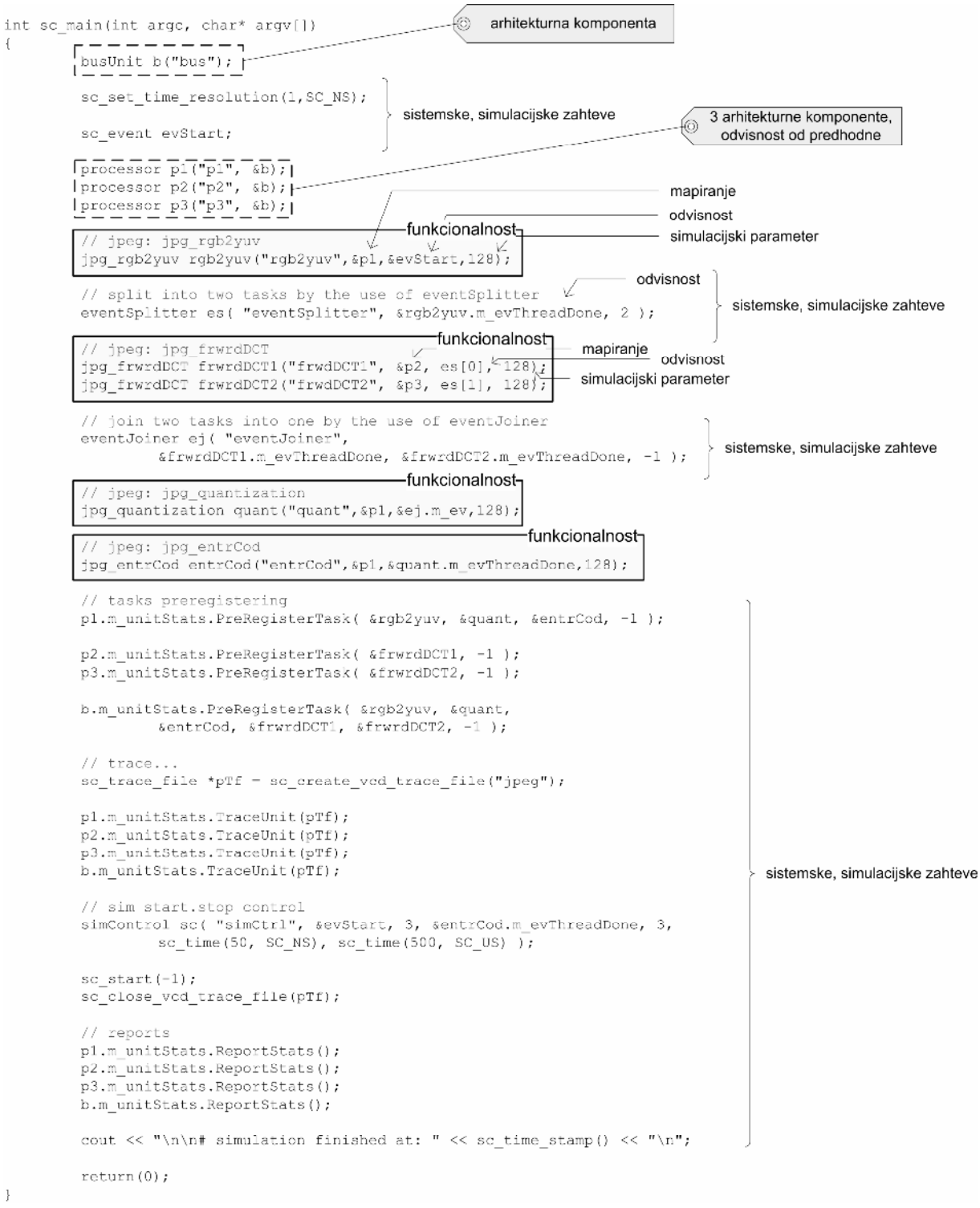
časovnega dodeljevanja sredstev in ne mehanizem prvi-pride-prvi-izvaja. Ugotovimo lahko, da je že ta zelo preprost primer nabito poln z naprednimi koncepti SNSPO.




Slika 44 – Časovne sledi izvajanja algoritma JPEG s tremi procesorji

V drugem, izboljšanem, koraku modeliranja smo sistem kodiranja JPEG pohitrili. Ugotovimo, da bi bilo morda smiselno procesor `p1` razbremeniti obdelave vseh treh blokov prve stopnje algoritma (`rgb2yuv`), in da bi bilo morda potrebno uvesti prioriteto na deljenem vodilu za opravila, ki trajajo dalj časa ... Vse to predstavlja že napredne koncepte SNSPO in zahteva avtomatizacijo raziskovanja načrtovalskega prostora ter uporabo optimizacijskih algoritmov. Cilj te doktorski teze je bil priti do te točke in v tem smo naredili nekaj edinstvenega. Zastavili smo smernice ogrodja SNSPO, ki bi zajelo sistem na učinkovit in uporabniku prijazen način ter hitro vrnilo informacije o sistemskih odločitvah. Šele sedaj je primeren čas za uporabo vseh sledečih – zahtevnih in ozko usmerjenih – algoritmov SNSPO.

## 6.1 Povezava z okvirji in okoljem GME




```
int sc_main(int argc, char* argv[])
{
        busUnit b("bus");

        sc_set_time_resolution(1,SC_NS);

        sc_event evStart;

        processor p1("p1", &b);
        processor p2("p2", &b);
        processor p3("p3", &b);

        // jpeg: jpg_rgb2yuv
        jpg_rgb2yuv rgb2yuv("rgb2yuv",&p1,&evStart,128);

        // split into two tasks by the use of eventSplitter
        eventSplitter es( "eventSplitter", &rgb2yuv.m_evThreadDone, 2 );

        // jpeg: jpg_frwrdDCT
        jpg_frwrdDCT frwrdDCT1("frwdDCT1", &p2, es[0], 128);
        jpg_frwrdDCT frwrdDCT2("frwdDCT2", &p3, es[1], 128);

        // join two tasks into one by the use of eventJoiner
        eventJoiner ej( "eventJoiner",
                &frwrdDCT1.m_evThreadDone, &frwrdDCT2.m_evThreadDone, -1 );

        // jpeg: jpg_quantization
        jpg_quantization quant("quant",&p1,&ej.m_ev,128);

        // jpeg: jpg_entrCod
        jpg_entrCod entrCod("entrCod",&p1,&quant.m_evThreadDone,128);

        // tasks preregistering
        p1.m_unitStats.PreRegisterTask( &rgb2yuv, &quant, &entrCod, -1 );

        p2.m_unitStats.PreRegisterTask( &frwrdDCT1, -1 );
        p3.m_unitStats.PreRegisterTask( &frwrdDCT2, -1 );

        b.m_unitStats.PreRegisterTask( &rgb2yuv, &quant,
                &entrCod, &frwrdDCT1, &frwrdDCT2, -1 );

        // trace...
        sc_trace_file *pTf = sc_create_vcd_trace_file("jpeg");

        p1.m_unitStats.TraceUnit(pTf);
        p2.m_unitStats.TraceUnit(pTf);
        p3.m_unitStats.TraceUnit(pTf);
        b.m_unitStats.TraceUnit(pTf);

        // sim start.stop control
        simControl sc( "simCtrl", &evStart, 3, &entrCod.m_evThreadDone, 3,
                sc_time(50, SC_NS), sc_time(500, SC_US) );

        sc_start(-1);
        sc_close_vcd_trace_file(pTf);

        // reports
        p1.m_unitStats.ReportStats();
        p2.m_unitStats.ReportStats();
        p3.m_unitStats.ReportStats();
        b.m_unitStats.ReportStats();

        cout << "\n\n# simulation finished at: " << sc_time_stamp() << "\n";

        return(0);
}
```

Slika 45 – Primer glavnega povezovalnega dela kode za tri procesorje

Slika 45 prikazuje glavno povezovalno datoteko sistema kodiranja JPEG v primeru treh procesorjev. Programska koda služi sistemskemu povezovanju komponent in predstavlja eno izmed možnih načrtovalskih odločitev, ki bi jo želeli preveriti. Ta ista koda je bila uporabljena

za kreiranje simulacijskega modela treh procesorjev, katere rezultate smo prestavili. Bralcu verjetno ni težko ugotoviti, kako bi iz te kode dobil prvopredstavljeni primer z enim procesorjem. Programsko kodo smo dopolnili s komentarji, ki prikazujejo, kateri koncept SNSPO predstavlja posamezna vrstica. Pri prebiranju kode lahko opazimo določene vzorce, ki se ponavljajo. Ti vzorci nam služijo kot osnova za gradnjo okvirjev, ki povezujejo modeliranje v okolju GME in ustrezno generirano programsko kodo.

Ni težko najti povezave med parametri funkcij oz. razredov in konceptom modeliranja, ki smo ga prestavili v poglavju 5.3.1. Čeprav je bilo prej omenjeno poglavje morda zahtevno za razumevanje, si lahko predstavljamo ekvivalenten primer programiranja v visokonivojskem vizualnem okolju. Za uporabo določenega gumba npr. ne rabimo vedeti nič več kot to, da moramo gumb postaviti v okno aplikacije in mu določiti funkcijo, ki se bo izvedla ob določenem dogodku (npr. ob kliku). Tako zelo enostavna uporaba navkljub ogromni spodaj ležeči kompleksni implementaciji (knjižnice) je mogoča zaradi objektno orientiranega koncepta, ki skrije spodnje nivoje izvedbe. Povsem paralelno razmišljanje velja pri uporabi naše metodologije. S pomočjo naprednih konceptov modeliranja, ki jih podpira GME, in tudi objektno orientiran pristop jezika C++ (SystemC), bo končna uporaba tudi tako enostavna, kot izbira komponente iz knjižnice in ustrezna povezava. Ne smemo pa pozabiti, da v okviru te disertacije razlagamo implementacijo sámo in še to v začetni stopnji razvoja okvirjev. Pot do uporabniku prijaznega sistema je seveda še dolga in predstavlja eno izmed področij našega dela, katero smo pustili odprto za nadaljnja raziskovanja.

# 7 Sklep

## 7.1 Prispevki k znanosti

Pomembnejši izvirni prispevki k znanosti so naslednji:

- predlog enovite obravnave elektronskega sistema, ki zajema vse začetne načrtovalske korake,
- razširitev klasičnega koncepta abstrakcije in ovrednotenje sistema na osnovi informacij na višjem nivoju abstrakcije,
- razvojno okolje za avtomatizacijo načrtovalskega postopka s pomočjo modeliranja in s podporo predlaganega sočasnega načrtovanja strojne in programske opreme ter
- komponentno zajemanje podatkov iz različnih domen obravnavanega sistema.

Naše raziskovalno delo smo usmerili v celovito obvladovanje mehanizmov sodobnega načrtovanja kompleksnih elektronskih sistemov. Rezultati in spoznanja predstavljajo temeljno ogrodje za nadaljnje raziskovalne dejavnosti s področja SNSPO, ki poteka v Laboratoriju za načrtovanje integriranih vezij.

## 7.2 Možnosti za nadaljnje delo

Koncepti obvladovanja kompleksnosti in metodologija SNSPO predstavljajo okvirje nadaljnjega dela. K raziskovanju SNSPO smo pristopili z druge strani, kot je to običajno pri ostalih raziskovanjih na tem področju, in se nismo osredotočili na specifične algoritme ožje usmerjenih podsklopov. Ti spodaj so ravno tako zelo pomembni, saj visokonivojsko obravnavo sistema dopolnijo s podrobno obravnavo nižjenivojskih aspektov načrtovanja. Samo preko njih je mogoče postopoma priti do končne implementacije sistema.

Zaenkrat je med spodaj ležečimi mehanizmi in našim delom še določena razdalja, ki jo bo potrebno v nadaljnjem raziskovanju premostiti in postopoma v ogrodje naše metodologije vključiti prav vse detajle načrtovanja sistema. Da bi bilo metodologijo SNSPO mogoče uspešno nadgraditi oz. dopolniti, je potrebno raziskati možnosti modularnega nadgrajevanja metodologije. Ti dopolnjevalni moduli bodo predstavljali implementacije algoritmov, ki obravnavajo nižje aspekte SNPO.

Čisto zgornji nivo modeliranja, ki predstavlja stik z načrtovalcem, je še v zelo grobi obliki in tudi potrebuje nadgradnjo, ki ga bo naredila uporabniku prijaznega in zares učinkovitega. Nakazali smo, kako bi to lahko učinkovito naredili s pomočjo modelirnega okolja GME. Pomemben del nadgradnje predstavlja tudi gradnja knjižnic elementarnih gradnikov s podporo konceptov SNSPO.

## 7.3 Zaključek

Pokazali smo, da je za uspešno in obvladljivo načrtovanje kompleksnih elektronskih sistemov potreben posodobljen oz. celovit pristop. Visokonivojske načrtovalske odločitve, za katere je običajna praksa, da se sprejemajo ad hoc, močno vplivajo na končno učinkovitost izvedbe celotnega sistema. Navkljub temu je to področje slabo raziskano in še slabše podprto z načrtovalskimi orodji. V doktorski disertaciji smo se zato osredotočili na visokonivojske koncepte obvladovanja kompleksnosti, predstavili mehanizme obravnavanja abstraktnih višjenivojskih informacij sistema in prikazali smernice okolja, ki bi nudilo podporo za predstavljeno metodologijo. Premišljeno smo izbrali pristop, v katerem smo želeli čim učinkoviteje izrabiti že obstoječe nižjenivojske načrtovalske pristope (tj. obstoječe programske jezike in njim pripadajoča orodja). Tu smo se osredotočili predvsem na njihovo vgradnjo v metodologijo SNSPO in podporo za njihovo uporabo na višjih nivojih abstrakcije.

V jedru doktorske disertacije smo se osredotočili na posamezne koncepte obvladovanja kompleksnosti, katere smo povezali v zaključeno celoto. Te smo definirali ločeno po poglavjih in prikazali, na kateri aspekt SNSPO vplivajo. Tu je še posebnega pomena predlog uporabe višjih nivojev abstrakcije, ki predstavlja naš izvirni pristop k načrtovanju na sistemskem nivoju. Na koncu je celotna metodologija SNSPO predstavljena z usklajeno povezavo vseh teh konceptov. Ti koncepti in jasno določene načrtovalske stopnje metodologije predstavljajo temelje načrtovanja, ki jih je mogoče formalizirati in avtomatizirati v okviru ogrodja, ki implementira celotno metodologijo.

Uporaba visokonivojskih konceptov obvladovanja kompleksnosti omogoča učinkovitejšo načrtovanje sistema in kot tako pripomore k nižanju razkoraka med tehnološkimi in načrtovalskimi zmožnostmi. Pri načrtovanju sistemov je zlasti pomembna sposobnost obvladovanja načrtovalskega prostora, saj predstavlja potencialne kandidate implementacije sistema, katerega načrtovanje je v prvi vrsti usmerjeno z načrtovalskimi kompromisi. Tu je ovrednotenje na višjem nivoju abstrakcije ključnega pomena in omogoča hitrejšo usmeritev načrtovalskega napora k detajlom sistema, ki so zares pomembni. Kako učinkovito je visokonivojsko ovrednotenje načrtovalskih odločitev, smo pokazali s praktičnim primerom.

Implementacija konceptov visokonivojske obravnave načrtovanja elektronskih sistemov s predstavljenimi mehanizmi obvladovanja načrtovalskega prostora bo v končnem cilju omogočala hitro in učinkovito načrtovanje zapletenih sodobnih sistemov. Takšen pristop bo

omogočal pregleden nadzor nad potekom načrtovanja, načrtovanje usmerjal s pomočjo hitrih visokonivojskih ovrednotenj in poenoteno zaobjel celoten načrtovalski cikel, vse od opisa začetne ideje do končne implementacije.

Sistematična in premišljena obravnava izredno širokega področja implementacije bo predstavljala temelje utemeljenega odločanja za načrtovanje sistema na vseh nivojih abstrakcije. Rezultat takšnega načrtovanja bo predloga izvedbe sistema, ki bo predstavljala učinkovit način realizacije podanih načrtovalskih zahtev. Z učinkovitim načinom pa ne mislimo samo na tehnološki izkoristek razpoložljive strojne opreme, ampak tudi učinkovito vključevanje in izrabo intelektualne lastnine. Če se povrnemo nazaj v uvod, v katerem smo razložili razkorak med tehnologijo izdelave integriranih vezij in njeno sedanjo neučinkovito uporabo, sta ideja in namen pričujoče doktorske teze tako povsem jasni.

# *Okrajšave*

| | |
|---|---|
| ALE | aritmetična logična enota |
| API | programski vmesnik aplikacije (*application programming interface*) |
| ASIC | integrirana vezja za določen namen (*application specific integrated circuit*) |
| DB | podatkovna baza (*data base*) |
| DCT | diskretna kosinusna transformacija (*discrete cosine transform*) |
| DPCM | diferencialno pulzno moduliranje (*differential pulse code modulation*) |
| DSP | digitalni signalni procesor (*digital signal processor*) |
| FPGA | programirljivo polje vrat (*field programmable gate array*) |
| FSM | avtomat s končnim številom stanj (*finite state machine*) |
| GME | generično modelirno okolje (*generic modeling environment*) |
| HDL | jezik za opis strojne opreme (*hardware description language*) |
| JFIF | oblika datoteke JPEG za izmenjavo (*JPEG file interchange format*) |
| JPEG | skupina združenih fotografskih strokovnjakov (*joint photographic experts group*) |
| MoC | računski model (*model of computation*) |
| MPSoC | večprocesorski sistem na integriranem vezju (*multiprocessor system on chip*) |
| OCL | jezik za opis omejitev objektov (*object constraint language*) |
| OS | operacijski sistem (*operating system*) |
| PO | programska oprema (*SW, software*) |
| RLE | kodiranje dolžine ponavljanja (*run length encoding*) |
| RTOS | operacijski sistem s podporo za sprotno izvajanje (*real time operating system*) |
| SLDL | jezik za načrtovanje na nivoju sistema (*system level design language*) |
| SNSPO | sočasno načrtovanje strojne in programske opreme (*HW/SW co-design*) |
| SO | strojna oprema (*HW, hardware*) |
| SoC | sistem na integriranem vezju (*system on chip*) |
| TLM | modeliranje na nivoju transakcij (*transaction level modeling*) |
| UML | poenoten jezik modeliranja (*unified modeling language*) |
| VLIW | zelo dolga instrukcijska beseda (*very long instruction word*) |

# *Literatura in reference*

## *Članki*


[p1] Ahmed A. Jerraya, Wayne Wolf. "Hardware/Software Interface Codesign for Embedded Systems", IEEE Computer Society, zv. 38, št. 2, februar 2005, str. 63–69.

[p2] A. A. Jerraya, "Long Term Trends for Embedded System Design", *EUROMICRO Symposium on Digital System Design (DSD 2004)*, Rennes, Francija: sept. 2004.

[p3] W. O. Cesário et al., "Multiprocessor SoC platforms: A component based design approach", IEEE Design and Test of Computers, zv. 19, izdaja 6 (November 2002), ISSN:0740-7475, str. 52–63.

[p4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli. "Metropolis: An Integrated Electronic System Design Environment", IEEE Computer Society, zv. 36, št. 4, april 2003, str. 45–52.

[p5] L. Kaouane, M. Akil, Y. Sorel, T. Grandpierre, "From algorithm graph specification to automatic synthesis of FPGA circuit: a seamless flow of graph transformations," *FPL Proceedings*, september 2003, str. 934–943.

[p6] E. A. Lee, "Embedded Software – An Agenda for Research", ERL Technical Report UCB/ERL št. M99/63, University of California, Berkeley, december 1999.

[p7] *G. Stitt, R. Lysecky, F. Vahid*, "Dynamic Hardware/Software Partitioning: A First Approach", Design Automation Conference Proceedings 2003 (DAC'03), str. 250–255.

[p8] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, M. Aboulhamid, F.-R. Boyer (2003), "SPACE: A Hardware/Software SystemC modeling platform including an RTOS", Forum on Design Languages (FDL03), Frankfurt, Nemčija: 09/2003, str. 704–715.

[p9] L. Cai, A. Gerstlauer, D. Gajski, "Retargetable Profiling for Rapid, Early System Level Design Space Exploration", Proceedings of the 41st annual conference on Design automation, 2004, ISBN:1-58113-828-8, str. 281–286.

[p10] L. Cai, D. Gajski, "Transaction Level Modeling: An Overview", CODES+ISSS'03, Newport Beach, California, USA: oktober 2003, ISBN:1-58113-742-7, str. 19–24.

[p11] D. C. Suresh, W. A. Najjar J. Villareal, G. Stitt, F. Vahid, "Profiling Tools for Hardware/Software Partitioning of Embedded Applications", Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2003), San Diego, CA, junij 2003.

[p12] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Zvezek 1: Introduction to Ptolemy II)," Technical Memorandum UCB/ERL M05/21, University of California, Berkeley, CA USA 94720, 15. julij 2005.

[p13] L. Kaouane, M. Akil, T. Grandpierre, Y. Sorel: "A Methodology to Implement Real-Time Applications onto Reconfigurable Circuits", The Journal of Supercomputing, zvezek 30, št. 3, december 2004, str. 283–301.

[p14] T. Grandpierre and Y. Sorel., "From algorithm and architecture specifications to automatic generation of distributed real-time executives: A seamless flow of graphs

transformations", First ACM& IEEE Intl. Conference on formal methods and models for codesign. MEMOCODE'03, Mont Saint-Michel, Francija, junij 2003.
[p15] A. Bouchhima, S. Yoo, A. A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software Using HW/SW Interface Simulation Model", ASP-DAC 2004, Yokohama, Japonska.
[p16] S. Gupta, N. Dutt, R. Gupta, A. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations", International Conference on VLSI Design, januar 2003.
[p17] *R. Hartenstein,* Data-stream-based Computing: Models and Architectural Resources, MIDEM Proceedings 2003, oktober 2003, str. 29–37.
[p18] B. Grattan, G. Stitt, and F. Vahid, "Codesign-Extended Applications", 10th International Symposium on Hardware/Software CoDesign, Estes Park, CO, maj 2002, str. 1–6.
[p19] Z. Guo, W. Najjar, F. Vahid, K. Vissers, "A Quantitative Analysis of the Speedup Factors of FPGAs over Processors", Proceedings of the 2004 ACM/SIGDA, 2004, ISBN:1-58113-829-6, str. 162–170.
[p20] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Communications of the ACM, zv. 21, št. 8, avgust 1978, str. 613–641.
[p21] M. Glesner, T. Murgan, L. S. Indrusiak, M. Petrov, S. Pandey, System design and integration in pervasive appliances, MIDEM proceedings '03, str. 97–108.
[p22] M. Weiser, "Some Computer Science Problems in Ubiquitous Computing", Communications of the ACM 36 (7), julij 1993, str. 74–84.
[p23] N. Sung Kim et all, "Leakage Current: Moore's Law Meets Static Power", Computer, zv. 36, št. 12, december 2003, str. 68–75.
[p24] J. J. Resano, M. E. Perez, D. Mozos, H. Mecha, J. Septien, Analyzing communication overheads during hardware/software partitioning, Microelectronics Journal 34 (2003), str. 1001–1007.
[p25] M. Akil, High-level Synthesis based upon Dependence Graph for Multi-FPGA, MIDEM Proceedings 2003, oktober 2003, str. 83–96.
[p26] T. Grandpierre, C. Lavarenne and Y. Sorel, "Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors", CODES'99 7th International Workshop on Hardware/Software Co-Design, Rim, maj 1999.
[p27] A. Jantsch, S. Kumar, A. Hemani, A Metamodel for Studying Concepts in Electronic System Design, Design & Test of Computers IEEE, zv. 17, št. 3, julij–september 2000, str. 78–85.
[p28] A. Jantsch, S. Kumar, A. Hemani, The Rugby Model: A Conceptual Frame for the Study of Modelling, Analysis and Synthesis Concepts of Electronic Systems, Design, Automation and Test in Europe (DATE '99), 1999, str. 256.
[p29] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi, "Benefits and Challenges for Platform-Based Design", Proceedings of the 41st annual conference on Design automation (DAC '04), 2004, str. 409–414.
[p30] S. Swan, Cadence Design Systems Inc., An Introduction to System Level Modeling in SystemC 2.0, maj 2001.
[p31] K. Einwich, C. Clauss, G. Noessing, P. Schwarz, H. Zojer, SystemC Extensions for Mixed-Signal System Design, FDL'01, Lyon, Francija, 3.–7. sept. 2001.
[p32] J. Dedič, A. Trost, A. Žemva, "Seamless HW/SW Co-design Flow", Informacije MIDEM, leto 34, št. 1, marec 2004, ISSN 0352-9045, str. 18–25.

[p33] J. Dedič, T. Žontar, A. Janhar, A. Trost, Design and implementation of customized embedded platform, MIDEM Proceedings, 2002, str. 213–218.
[p34] J. Dedič, A. Trost, A. Žemva, PC based HW/SW codesign support for embedded system target, MIDEM Proceedings, 2003, str. 249–254.
[p35] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "A Unified Codesign Run-time Environment for the UltraSONIC Reconfigurable Computer", Field-Programmable Logic and Applications Proceedings, september 2003, str. 396–405.
[p36] T. Wiangtong, Peter Y.K. Cheung, W. Luk, "Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign", *International Journal on Design Automation for Embedded Systems*, Kluwer Academic Publishers, 2002, zv. 6, str. 425–449.
[p37] K. Wallace, "The JPEG Still Picture Compression Standard, Communications of the ACM," april 1991, zv. 34, št. 4, str. 30–44.
[p38] V. D. Zivkovic, P. Lieverse, "An Overview of Methodologies and Tools in the Field of System-level Design", Lecture Notes in Computer Science, Tutorial on Embedded Processor Design Challenges, Systems, Architectures, Modeling, and Simulation (SAMOS'02), 2002, str. 74–89.
[p39] N. Wirth, Hardware/software co-design then and now, Information Processing Letters, zv. 88 , izdaja 1–2, oktober 2003, ISSN:0020-0190, str. 83–87.
[p40] Mentor Graphics, SoC Design and Verification White Paper; M. Andrews, Abstraction: Managing Design Complexity through High-Level C-Model Verification
[p41] D. Berner, H. Patel, D. Mathaikutty, J. P. Talpin, S. Shukla, "SystemCXML: An Extensible SystemC Front End Using XML", To be published in Proceedings of the Forum on specification and design languages (FDL), Lausanne, Švica, september 2005
[p42] FERMAT – Extensible SystemC Front End Using XML: http://systemcxml.sourceforge.net/
[p43] P. Lieverse, T. Stefanov, P. van der Wolf, E. Deprettere, System Level Design with Spade: an M-JPEG Case Study, Proceeding of int. conference on computer aided design (ICCAD'01), San Jose CA, USA, 4.–8. nov. 2001.
[p44] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, P. Volgyesi, "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, 17. maj 2001.
[p45] C. Loeffler, A. Ligtenberg, G. S. Moschytz, "Practical fast 1-D DCT Algorithms with 11 multiplications", Proceedings of ICASSP, zv. 2, 1989, str. 988–991.

## *Knjige*

[k1] F. Vahid, T. Givargis, "Embedded System Design – A Unified Hardware/ Software Introduction," John Wiley & Sons, 2002, ISBN: 0-471-38678-2.

[k2] G. De Micheli, "Synthesis and optimization of digital circuits," McGraw-Hill, Inc., 1994, ISBN 0-07-016333-2.

[k3] P. Marwedel, "Embedded System Design," Kluwer Academic Publishers, 2003, ISBN 1-4020-7690-8.

[k3.a] Chapter 2.2 – Models of computation

[k4] V. Bhaskaran, K. Konstantinides, "Image and Video Compression Standards", Second Edition, Kluwer Academic Publishers, 1997.

[k5] David C. Black & Jack Donovan, "SystemC from the ground up," Springer, ISBN: 1402079885, junij 2004.

[k5.a] Chapter 2, TLM-based methodology

[k5.b] Chapter 7.2, Simplified Simulation Engine

[k6] J. Bhasker, "A SystemC Primer," Star Galaxy Pub, junij 2002, ISBN: 0965039188.

[k7] U. Breymann, "Designing Components with the C++ STL: A New Approach to Programming," 2nd edition, Addison-Wesley Professional, 1999, ISBN: 0201674882.

[k8] S. H. Kaisler, "Software Paradigms", John Wiley & Sons, april 2005, ISBN: 0-471-48347-8.

[k9] M. Abd-El-Barr, H. El-Rewini, "Fundamentals of Computer Organization and Architecture", januar 2005, ISBN: 0-471-46741-3.

[k10] T. Basten, M. Geilen, H. de Groot, "Ambient Intelligence: Impact on Embedded System Design", Kluwer Academic Publishers, 2004, eBook ISBN: 0-306-48706-3, Print ISBN: 1-4020-7668-1.

[k11] K. Parnell, N. Mehta, Programmable Logic Design Quick Start Hand Book, 2nd Edition, Xilinx, januar 2002.

[k12] A. Jerraya, W. Wolf, "Multiprocessor Systems-on-Chips (The Morgan Kaufmann Series in Systems on Silicon)", Morgan Kaufmann, 28. september 2004, ISBN: 012385251X.

[k12.a] F. Balarin et al., Metropolis: A Design Environment for Heterogeneous Systems, poglavje 16.

[k12.b] W. O. Cesario, A.A. Jerraya: Component-Based Design for Multiprocessor Systems-on-Chip, poglavje 13.

[k12.c] J. M. Paul and D. E. Thomas: Models of Computations for Systems-on-Chips, poglavje 15.

[k13] Robert A. Maksimchuk, Eric J. Naiburg, UML for Mere Mortals, Addison Wesley Professional, 2004, Print ISBN-10: 0-321-24624-1.

[k14] R. Miles, K. Hamilton, Learning UML 2.0, O'Reilly Media, First Edition, april 2006, ISBN: 0-596-00982-8.

[k15] S. Prata , C++ Primer Plus, Fourth Edition, Sams Publishing, november 2001, ISBN : 0672322234.

[k16] Institute for Software Integrated Systems, Vanderbilt University, A Generic Modeling Environment: GME 5 User's Manual, Version 5.0.

## *Raziskovalne skupine*

[rs1] TIMA Laboratory, System-Level Synthesis (SLS) Group: http://tima.imag.fr/sls/
A. A. Jerraya: http://tima-cmp.imag.fr/Homepages/jerraya/jerraya.html

[rs2] Department of Electrical Engineering and Computer Sciences (EECS), UC, Berkely: http://www.eecs.berkeley.edu/
Center For Electronic Systems Design: http://www-cad.eecs.berkeley.edu/

[rs2.a] Ptolemy Project homepage: http://ptolemy.eecs.berkeley.edu/
Ptolemy II Project homepage: http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm
Edward A. Lee: http://ptolemy.eecs.berkeley.edu/~eal/

[rs2.b] Polis homepage: http://www-cad.eecs.berkeley.edu/research/hsc/

[rs2.c] Metropolis homepage: http://www.gigascale.org/metropolis/
A. Sangiovanni-Vincentelli: http://www-cad.eecs.berkeley.edu/~alberto/

[rs3] Center for Embedded Computer Systems (CECS) at UC Irvine:
SpecC system: http://www.cecs.uci.edu/~specc/
SoC Environment – SCE: http://www.cecs.uci.edu/~cad/sce.html
D. D. Gajski: http://www.cecs.uci.edu/~gajski/

[rs4] Department of Computer Science and Engineering, UC, San Diego:
http://www.cse.ucsd.edu/index.php
SPARK homepage: http://mesl.ucsd.edu/spark/
R. Gupta: http://www-cse.ucsd.edu/~gupta/

[rs5] AAA methodology and SynDEx homepage
http://www-rocq.inria.fr/syndex/

[rs6] OCAPI-XL (at IMEC - Interuniversity MicroElectronics Center):
http://www.imec.be/design/ocapi/

[rs7] CIRCUS, Codesign Research Group, part of Polytechnique's Microelectronic Research Group (GRM, http://www.grm.polymtl.ca/) of École Polytechnique de Montréal (http://www.polymtl.ca/en/), prof. Guy Bois
SPACE: http://www.grm.polymtl.ca/circus/en/projects/space/index.html

## *Spletne strani*

[ss1] SystemC OSCI homepage: http://www.systemc.org/

[ss2] SpecC Technology Open Consortium homepage: http://www.specc.org

[ss3] Object Management Group (OMG) Unified Modeling Language (UML) homepage:
http://www.uml.org/

[ss4] Independent JPEG Group: http://www.ijg.org

[ss5] Xilinx homepage: http://www.xilinx.com/

[ss6] Altera homepage: http://www.altera.com/

[ss7] Cadence TestBuilder homepage: http://www.testbuilder.net/

[ss8] Intel Exploratory Research: http://www.intel.com/research/

[ss9] Tensilica homepage: http://www.tensilica.com/

[ss10] Handel-C homepage: http://www.celoxica.com/technology/c_design/handel-c.asp

[ss11] Rational Rose homepage: www.ibm.com/software/rational

[ss12] Simulink® - Simulation and Model-Based Design:
http://www.mathworks.com/products/simulink/

[ss13] LabVIEW homepage: www.ni.com/labview/

[ss14] Extensible Markup Language: (XML) http://www.w3.org/XML/
XML Tutorial: http://www.w3schools.com/xml/

[ss15] The Code Project homepage: http://www.codeproject.com/

[ss16] Institute for Software Integrated Systems, Vanderbilt University, Generic Modeling Environment homepage: http://www.isis.vanderbilt.edu/projects/gme/index.html also available at Escher research institute: http://www.escherinstitute.org/tools/suites/mic/gme/

## *Spletni članki*

[a1] P. Bennett, The why, where and what of low-power SoC design, www.embedded.com, ID=55300196, december 2004.

[a2] D. Lammers, Guide to Success: Fear and loathing at next node, www.embedded.com, ID=173602492, november 2005.

[a3] Transistors per integrated circuit trends, www.icknowledge.com, 14074b.pdf, 06/04/2003

[a4] Microlithography trends, www.icknowledge.com, Microlithography.pdf, 2001

[a5] I. Page, Compiling software to gates, www.embedded.com, ID=55801142, december 2004.

## *Wikipedia*

[wp1] *Integrated circuit,* http://en.wikipedia.org/wiki/Integrated_circuit

[wp2] *John von Neumann*, http://en.wikipedia.org/wiki/John_Von_Neumann

[wp3] *NP-complete*, http://en.wikipedia.org/wiki/NP-Complete

[wp4] *Formal specification*, http://en.wikipedia.org/wiki/Formal_specification

[wp5] *Formal verification*, http://en.wikipedia.org/wiki/Formal_verification

# *Izjava*

V skladu s Pravilnikom o doktorskih disertacijah Fakultete za elektrotehniko v Ljubljani izjavljam, da sem doktorsko disertacijo z naslovom Enovito razvojno okolje za sočasno načrtovanje strojne in programske opreme izdelal samostojno, pod vodstvom mentorja prof. dr. Andreja Trosta. Izkazano pomoč ostalih sodelavcev sem v celoti navedel v zahvali.

Jože Dedič