VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

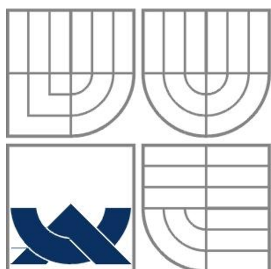
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO GRAFICKÉ PROTOTYPOVÁNÍ SYSTÉMŮ NA ČIPU

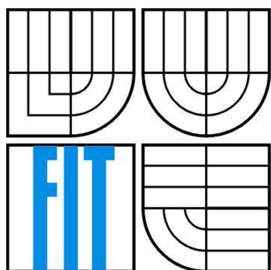
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Ondřej Netočný
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO GRAFICKÉ PROTOTYPOVÁNÍ SYSTÉMŮ NA ČIPU

GRAPHICAL TOOL FOR RAPID PROTOTYPING OF SYSTEM ON THE CHIP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Ondřej Netočný
AUTHOR

VEDOUCÍ PRÁCE Ing. Karel Masařík, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se věnuje návrhu a implementaci nástroje pro vývoj víceprocesorových systémů na čipu, seznámí čtenáře s touto problematikou a představí možnosti, jak lze problém řešit ve vývojovém prostředí Codasip Studio. Práce představuje jak samotný nástroj, kterým je grafický editor víceprocesorových systémů na čipu, tak sadu podpůrných prostředků pro rychlý a efektivní vývoj. Jedná se zejména o sadu interaktivních průvodců, kteří usnadní start nových projektů. Ke zvládnutí této problematiky je nutné porozumět jazyku pro popis architektury CodAL, vývojovému prostředí Eclipse a nástrojům GMF (Graphical Modeling Framework) a EMF (Eclipse Modeling Framework), s jejichž pomocí je grafický editor implementován.

Klíčová slova

MPSoC, grafický editor, Eclipse, jazyk CodAL, EMF, GMF, HW/SW codesign.

Abstract

This thesis deals with design and implementing of a tool for development of MPSoC (multiprocessor systems on chip). It is going to apprise the reader with this matter and introduces several ways how to solve these issues in Codasip Studio IDE (integrated development environment). The graphical editor for multicore system development and a set of support tools for fast and effective development are introduced in this thesis. These are mainly interactive wizards which help user to start new projects. To handle the subject matter it is necessary to understand CodAL language, Eclipse IDE, GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Framework) which are used for graphical editor implementation.

Keywords

MPSoC, graphical editor, Eclipse, CodAL language, EMF, GMF, HW/SW codesign.

Citace

Netočný Ondřej: Nástroj pro grafické prototypování systémů na čipu. Brno, 2013, diplomová práce, FIT VUT v Brně.

Nástroj pro grafické prototypování systémů na čipu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Karla Masaříka, Ph.D. Další informace mi poskytli Ing. Zdeněk Přikryl, Ph.D. a Ing. Ondřej Ilčík. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Ondřej Netočný
21. 5. 2013

Poděkování

Děkuji Ing. Zdeňkovi Přikrylovi, Ph.D., Ing. Ondřeji Ilčíkovi a Ing. Karlu Masaříkovi, Ph.D. za jejich ochotu a čas věnovaný nespočtu odborných konzultací, které moje práce vyžadovala.

© Ondřej Netočný, 2013.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod ... 3
2 Cíle práce ... 5
3 Současný stav ... 7
3.1 Projekt Lissom ... 7
3.2 Jazyk CodAL ... 8
3.3 Grafické editory v Codasip Studio ... 11
4 Analýza a návrh řešení ... 14
4.1 UML diagram komponent ... 14
4.2 Vlastní návrh nástroje ... 15
4.3 Popis standardních komponent ... 18
4.4 Úvod k nástrojům EMF a GMF ... 21
4.4.1 Eclipse Modeling Framework ... 21
4.4.2 Graphical Modeling Framework ... 22
4.5 Podpůrné nástroje ... 23
4.5.1 Interaktivní průvodci ... 23
4.5.2 Online projekty ... 24
5 Implementace ... 26
5.1 Interaktivní průvodci ... 26
5.2 Online projekty ... 30
5.2.1 Grafické uživatelské rozhraní a datová struktura ... 30
5.2.2 Síťová komunikace ... 31
5.3 Implementace grafického nástroje ... 34
5.3.1 Definice metamodelu a generování kódu ... 34
5.3.2 Úpravy kódu diagramu ... 38
5.3.3 Integrace do vývojového prostředí ... 41
5.3.4 Generování výstupů ... 43
5.4 SoC projekty ... 46
6 Výsledné nástroje ... 47
6.1 Možná rozšíření ... 50
6.1.1 Hromadné targety SoC projektů ... 50
6.1.2 Generování výstupu v jazyce IP-XACT ... 50
6.1.3 Grafický editor IP Core komponent ... 50
6.1.4 Online dokumentace ... 50
6.1.5 Vizuální průvodce ... 50

7 Závěr 51
Literatura 52
A Obsah CD 54

1 Úvod

V posledních desetiletích neustále roste využití mikroprocesorů a vestavěných systémů. Víceprocesorové systémy na čipu a vestavěné systémy se už dnes vyskytují téměř všude. Jejich rozšíření nejprve zasáhlo do průmyslu a přes automobily a spotřební elektroniku se dostalo i do domácností. Kromě klasického uplatnění ve spotřební elektronice nachází dnes už využití i v exotičtějších odvětvích jako je biomedicína a biometrie. Tajemství jejich úspěchu je prosté – jsou často malé, snadno konfigurovatelné, mají nízkou spotřebu, ale hlavně mají obrovské uplatnění téměř všude, kde je potřeba něco automatizovaně ovládat a řídit. V souvislosti s rozvojem a rozšířením multiprocesorů roste také oblíbenost technologie víceprocesorových systémů na čipu, zkráceně MPSoC (*Multiprocessor system on chip*). Složitost jednotlivých procesorů může růst jen do určité míry a je jednodušší řešené problémy dekomponovat na moduly. Jedná se tedy o víceprocesorové systémy, které jsou integrovány na jednom čipu a vzájemně spolu komunikují a spolupracují. Jednotlivé procesory v systémech slouží ke specifickým účelům, může se jednat o řízení nejrůznějších periferií, komunikaci s okolím, řízení celého systému nebo nějaký druh DSP (*Digital Signal Processor*). Většina z těchto procesorů je určena ke specifickému účelu, proto se nazývají zkratkou ASIP (*Application Specific Instruction-set Processors*, dále už jen procesory).

Vývoj takových procesorů i celých systémů je vysoce náročná činnost, která je ovšem stále žádanější. Současné trendy v této oblasti se odvracejí od tradičních metod vývoje [1] [2] a tak na řadu přichází hardware/software codesign. Metoda, která umožňuje souběžný návrh samotných procesorů a aplikací pro takto navržené procesory. Touto problematikou se na Fakultě informačních technologií na VUT v Brně zabývá projekt *Lissom* [3]. Tato výzkumná skupina vyvíjí sadu nástrojů pro návrh kompletních procesorů, generování nástrojů pro práci s nimi a simulaci jejich činnosti. V rámci projektu vzniká také integrované vývojové prostředí, které spojuje několik nástrojů projektu Lissom a umožňuje rychlý vývoj v jednom prostředí. V tomto prostředí jsou uživateli k dispozici nástroje pro modelování a simulaci jednotlivých procesorů popsaných jak v grafické, tak v textové podobě prostřednictvím jazyka pro popis architektur ADL (*Architecture Description Language*) [4] s názvem *CodAL* [5]. Doposud zde ale chybí nástroj pro modelování, zejména grafické modelování, komplexních víceprocesorových systémů. Tato práce se zabývá návrhem a implementací takového nástroje a jeho integrací do celého vývojového prostředí. Při návrhu bylo třeba zahrnout nároky na uživatelskou přívětivost a jednoduchost celého editoru, proto byla navržena a implementována řada interaktivních průvodců a podpůrných nástrojů, které práci na návrhu sytému usnadňují. Práce je členěna do několika kapitol. Její začátek seznámí čtenáře s nezbytnými prerekvizitami – jazykem CodAL a integrovaným vývojovým prostředím Lissom – *Codasip Studio* [6]. V následující části je představen koncept grafického nástroje pro návrh systémů na čipu a jeho propojení se současnými nástroji v prostředí. Jeho implementací a integrací do prostředí se zabývá opět samostatná kapitola.

Další část se věnuje návrhu a implementaci podpůrných nástrojů, které jsou pro efektivní práci s grafickým editorem nezbytné. Poslední část je věnována budoucnosti celého systému, zhodnocení přínosu práce a návrhům na možná rozšíření.

2 Cíle práce

Cílem této práce je vytvořit kvalitní, komplexní a pohodlný nástroj pro návrh víceprocesorových systémů na čipu. Samotná jednotlivá jádra systému – zejména procesory – mohou být poměrně složité systémy a jejich následné propojení může vést k celé řadě komplikací. Takto navržený systém se může stát často špatně čitelným, a tudíž velice náchylný k chybám. V takovém případě je vhodné zavést opatření, která tomuto zabrání nebo alespoň budou takovým problémům předcházet. Práce se snaží klást na tato opatření důraz a zohledňuje je v návrhu. Stěžejním přístupem je grafický návrh celého systému, výsledný nástroj bude totiž modelovat víceprocesorové systémy pomocí grafických diagramů, které mohou být, a velice často bývají, názornější a jednodušší na pochopení než textový popis. Nástroj bude zároveň stavět na existujících grafických editorech pro návrh jednotlivých procesorů a jejich instrukčních sad. Toto propojení bude obousměrné a návrh komplexních systémů se tak rozdělí do několika zapouzdřených a oddělených úrovní. Výsledný nástroj bude součástí komplexního nástroje Codasip Studio, který umožňuje návrh celých procesorů a následné generování nástrojů pro práci s nimi a simulaci jejich činnosti. Jeho součástí jsou také zmíněné grafické editory pro návrh procesoru a jeho instrukční sady. Výsledný grafický nástroj by měl fungovat v podobném duchu tak, aby byla práce v prostředí ucelená a jednotná.

Jedním z důvodů, proč jsou vyvíjeny systémy na čipu, je jejich znovupoužitelnost. Jednotlivá jádra plní svou specifickou funkci a do systémů jsou svazována vývojáři podle toho, co od celého systému vyžadují a jaké funkce potřebují. Stejná jádra se tak často používají znovu a znovu, bez jediné vnitřní změny, jsou tedy v roli jakési černé skříňky. Při návrhu nástroje pro modelování těchto systémů jsem tato fakta respektoval a vydal se podobným směrem. Vývojové prostředí Codasip Studio, do kterého bude výsledný nástroj integrován, bude umožňovat projít celým procesem návrhu systémů na čipu, od modelování jednotlivých procesorů, přes simulaci jejich činnosti až k jejich skládání do systémů. I zde bude zachována znovupoužitelnost jednotlivých komponent. Návrh jednotlivých úrovní systému bude oddělen pomocí různých úrovní diagramů, výsledný nástroj bude všechny stávající diagramové nástroje zastřešovat jedním komplexním diagramem, který bude modelovat propojení jednotlivých komponent. Ačkoliv je už samotný přechod ke grafickému popisu jistým krokem ke zlepšení čitelnosti celých systémů, může být práce v prostředí poměrně složitá, zejména pro začínající uživatele tohoto prostředí. Proto je nutné do prostředí vložit také sadu nástrojů, které urychlí a zjednoduší start nových projektů v prostředí Codasip Studio. Pro vývoj systémů na čipu není znalost jednotlivých komponent stěžejní, důležitá jsou pouze rozhraní a jejich standardizovaný popis. Nicméně komponenty systémů jsou pro návrh systému nezbytné. Proto vznikne v rámci této práce sada podpůrných nástrojů, které umožní tento problém překlenout. Uživatel tak bude moci generovat jednotlivé komponenty pomocí jednoduchých dialogů nebo

stahovat už hotová jádra ze sítě, která bude moci dle libosti editovat nebo přímo použít beze změny ve výsledném modelu systému na čipu.

3 Současný stav

Jak již bylo řečeno výše, výsledný nástroj bude integrován ve vývojovém prostředí Codasip Studio, které zastřešuje některé nástroje projektu Lissom (představení projektu je v kapitole 3.1). Toto vývojové prostředí zároveň plní funkci zjednodušení a částečnou automatizaci celého vývojového procesu. Následující kapitoly čtenáři přiblíží, o co v projektu Lissom jde a jak vypadá práce ve vývojovém prostředí, zejména se zaměřím na popis těch částí prostředí, na které bezprostředně naváže budoucí nástroj pro návrh víceprocesorových systémů. Jedná se o grafické nástroje pro popis procesoru – editory diagramů (kapitola 3.3). Sada těchto diagramů je ekvivalentní textovému popisu procesorů pomocí jazyka CodAL, kterému je věnována kapitola 3.2.

3.1 Projekt Lissom

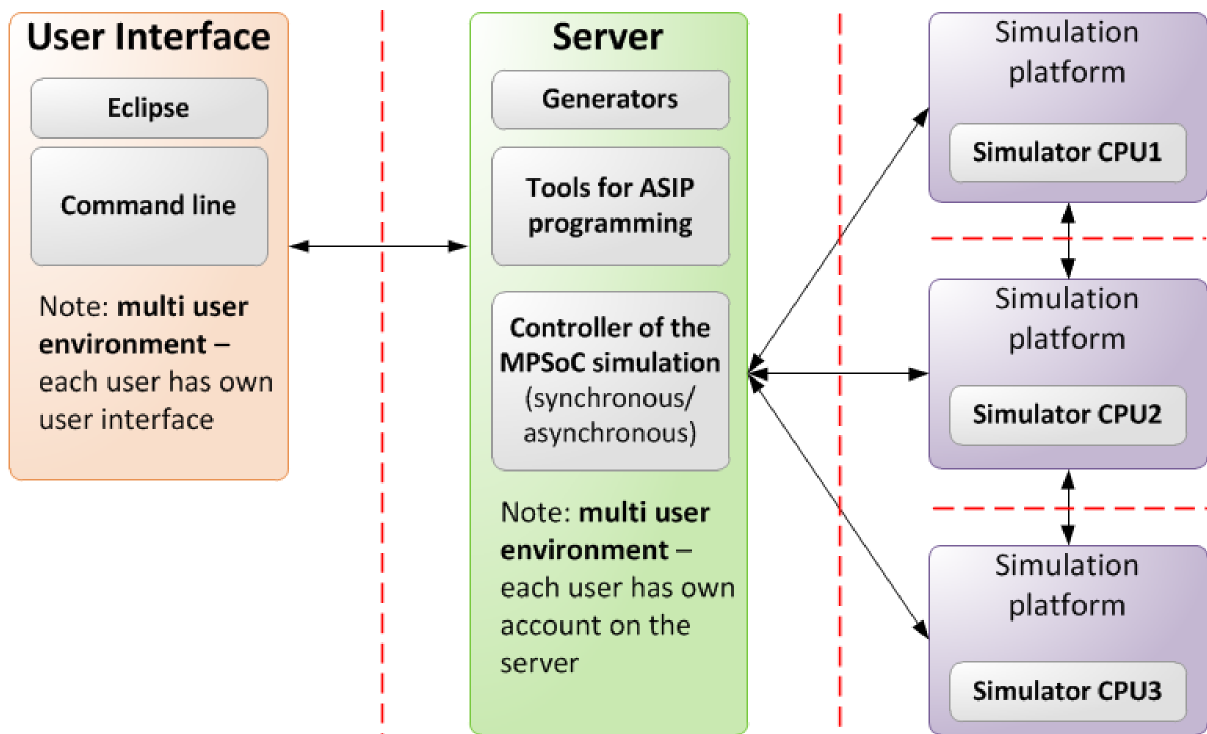
Projekt Lissom funguje už několik let na Fakultě informačních technologií na Vysokém učení technickém v Brně. Činnost projektu se rozpadá na dvě hlavní oblasti, které spolu ale úzce souvisí. První z nich je jádro celého projektu, jazyk CodAL (dříve *ISAC*). Jedná se o jazyk pro popis architektury ADL, který umožňuje komplexní popis procesoru včetně jeho instrukční sady. Druhou částí projektu je generování nástrojů pro vývoj a simulaci činnosti procesorů a víceprocesorových systémů popsaných pomocí jazyka CodAL. Obě oblasti jsou zastřešeny ve vývojovém prostředí Codasip Studio, které umožňuje současný návrh a vývoj jak softwarových nástrojů, tak hardware [7] [3].

Projekt Lissom je tedy zaměřen na:

- Vývoj jazyka pro popis architektury CodAL.
- Sadu softwarových nástrojů, které tento jazyk využívají:
    - *Kompilátor jazyka* CodAL umožňuje převod textového popisu architektury do XML (*Extensible Markup Language*) modelu, se kterým se dále pracuje.
    - *Kompilátor* a *dekompilátor jazyka C*, který umožňuje kompilaci a ladění programů na libovolné architektuře.
    - *Generátor* assembleru, disassembleru a simulátorů, umožňující vytvoření nástrojů pro vývoj software založených na interním modelu hardware. Simulátor je vytvořen ve dvou verzích: tzv. *instruction accurate* a *cycle accurate*. Obě varianty umožňují ladění aplikace. Simulátor podporuje i víceprocesorovou simulaci.
    - Vývojové prostředí Codasip Studio.

Vývojové prostředí Codasip Studio je postaveno na třívrstvé architektuře (viz obrázek 1) a aplikace tudíž funguje jako tenký klient. Vrstvy mezi sebou komunikují pomocí proprietárního protokolu, který je přenášen protokolem TCP/IP. Jednotlivé vrstvy architektury jsou tyto [8]:

- *Uživatelské rozhraní* je postaveno na platformě *Eclipse* a umožňuje uživatelům jak komplexní návrh procesorů, tak zasílání příkazů k simulaci jejich činnosti, generování výše popsaných nástrojů a zobrazování výsledků simulací. Součástí této vrstvy bude také implementovaný nástroj pro grafický návrh systémů na čipu.
- *Middleware* je vrstva, která se stará o zpracovávání příkazů z uživatelského rozhraní a zprostředkovává komunikaci se simulační vrstvou. Tato vrstva je zodpovědná za generování některých nástrojů a simulátorů a informuje o průběhu činnosti vrstvu uživatelského rozhraní.
- *Simulační vrstva* umožňuje umístit jednotlivé simulátory kdekoliv v síti a decentralizovat tak simulaci, což její průběh může značně urychlit. Činnost simulátorů je spravována vrstvou middleware.



Obrázek 1: Třívrstvá architektura vývojového prostředí Codasip [8]

3.2 Jazyk CodAL

Jazyk CodAL je klíčovým prvkem celého projektu a zároveň jsou na něm vystavěny nástroje pro grafický popis architektury. Jazyk slouží v projektu Lissom pro textovou specifikaci architektury procesorů a víceprocesorových systémů. Jazyk byl navržen tak, aby pokryl potřeby paralelního vývoje architektury v hardware a programového vybavení procesorů. Na základě specifikace

architektury je možné automaticky generovat nástroje pro simulování a programování ASIP. Jazyk CodAL zároveň umožňuje automatické generování implementace architektury v jazycích pro popis hardware HDL (*Hardware description language*) [4]. Proces automatizace je založen na generátorech, které umožňují přímé generování simulačních nástrojů a hardwarové implementace. Definice procesoru je překládána do vnitřní XML reprezentace, která je dále používána ke generování hardwarových a softwarových nástrojů pro práci s procesorem [5] [9].

Základní struktura jazyka je členěna do dvou velkých skupin. První oblastí je *resource* sekce. Tato sekce popisuje zdroje procesoru, jeho komponenty a vzájemná propojení. Součástí sekce je popis jednotlivých prvků pomocí povinných i řady volitelných atributů. V této sekci se mohou objevit prvky jako paměťové elementy (například *ram*, *cache*…), signály, registry a sběrnice. Dále sekce resource obsahuje kompletní popis fyzického propojení těchto prvků, *pipeline* a *mapování paměti* (adresní prostor procesoru). Velké množství zmíněných prvků je v modelu pouze volitelné, ovšem *resource* sekce vyžaduje povinnou přítomnost těchto prvků:

- paměť pro uložení zdrojového kódu,
- speciální registr *program counter* (programový čítač, sekce resources obsahuje pouze jeden),
- základní mapování paměti.

Druhou částí jazyka CodAL je sekce pro popis instrukční sady a událostí (*instructionset* sekce). Tato popisuje zejména stavbu instrukční sady, textovou a binární reprezentaci instrukcí a popis jejich chování. Dále obsahuje popis reakcí na jednotlivé události (*events*) v procesoru (například popis chování v jednotlivých stupních pipeline, chování po resetu procesoru atd.). Jazyk CodAL vyžaduje, aby každý model obsahoval alespoň jednu instrukci a popis těchto základních událostí:

- *Main* – událost, která popisuje činnost procesoru na začátku každého cyklu (v této události může být například dekódována následující instrukce v kódu, může zde být řešeno přerušení atd.).
- *Halt* – událost, která nastane pouze při ukončení simulace, zde je možné doplnit požadované chování v takové situaci (například výpisy pro ladící účely).
- *Reset* – nastává při resetu procesoru, zde je možné například vynulovat programový čítač nebo provést jiné požadované operace.

Samotná struktura instructionset sekce je hierarchická, a kromě událostí a instrukcí (*element*), obsahuje také skupiny (*set*). Do skupin lze sdružovat instrukce i jiné skupiny. Popis jednotlivých instrukcí obsahuje popis textové reprezentace v *asembleru* a *binární reprezentace*. Binární reprezentace instrukce se může lišit podle různých parametrů (například šířka paměti), proto je možné tuto sekci členit do jednoduchých větví pomocí standardních řídících konstrukcí *if-else* a *switch*. Specifikace instrukce dále obsahuje popis chování instrukce v podmnožině jazyka ANSI C a popis

tzv. časování (*timing*). Jedná se o popis sledu událostí, které po instrukci (nebo také události) následují. V popisu časování jsou opět použity standardní řídící konstrukce jako v binární části.

Následující kód znázorňuje jednoduchý model v jazyce CodAL:

```
//structure section
program_counter bit[8] pc;
ram bit[8] mem
{
	endianess = little,
	lau = 8,
	size = 256,
	flags = {r,w,x}
};
bus bit[8] defbus
{
	lau = 8,
	endianess = little
};
memorymapping defaultmap
{
	bus defbus: 0..255 = mem[7..0];
};

//instructionset and events section
element inop
{
	assembler {"nop"};
	binary {0b0000};
}
event halt {}
element ihalt
{
	use halt;
	assembler {"halt"};
	binary {0b1111};
	timing {halt;};
}
set instructions = inop, ihalt;
event main
{
	use instructions;
	start { {instructions;}};
	decoders (pc) { {instructions(bus[pc]);}};
}
event reset
{
	semantics
	{
		pc = 0;
	}
}
```

V sekci `structure` je definováno několik hardwarových komponent a dále je uvedena definice mapování paměti:

- programový čítač (`pc`) o velikosti 8 bitů,

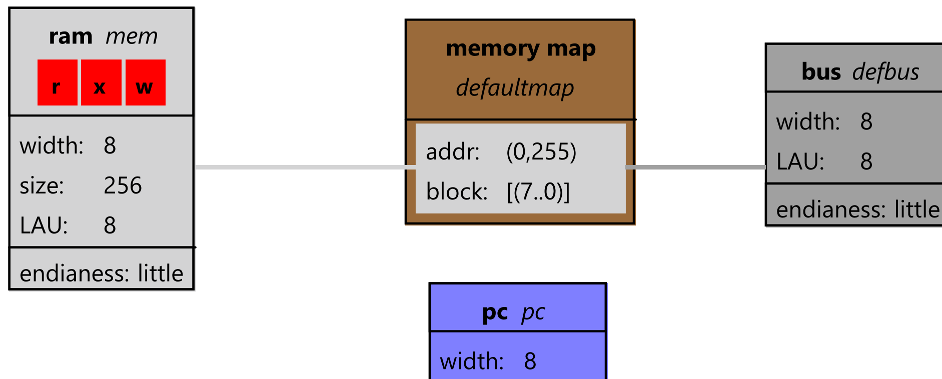
- paměťový prvek typu `ram` (`mem`) s 256 bloky, každý o velikosti 8 bitů, z této paměti je možno číst, zapisovat do ní a vykonávat kód, který je v ní uložený, paměť ukládá data (*endianita*) od nejméně významného bitu LSB (*least significant bit*) po nejvíce významný MSB (*most significant bit*) – little endian, zkratka `lau` značí nejmenší adresovatelnou jednotku (*least addressable unit*) a je velikosti 8 bitů,
- sběrnice (`defbus`) o šířce 8 bitů, s podobnými parametry jako dříve definovaná paměť,
- prvek definující mapování paměti (`defaultmap`) obsahuje pouze mapování nadefinované paměti (`mem`), ve větších modelech může obsahovat i více mapování, v tomto případě je paměť mapována do prostoru adres od 0 do 255, adresování respektuje endianitu paměti, proto `[7..0]`, paměť je připojena pomocí dříve definované sběrnice (`defbus`).

Dále následuje sekce `instructionset`, která popisuje instrukční sadu a události. Nejprve je definována instrukce (`element`) `inop`, je uvedena jak její textová reprezentace pro asembler, tak její binární reprezentace pro kompilátory atd. Sekce dále obsahuje všechny tři povinné události – `halt`, `main` a `reset`. Událost `halt` má svůj zvláštní význam při simulaci, a tak nemusí být implementována sekce popisující její chování (`semantics`). Zatímco na chování události `reset` je názorně resetován programový čítač. Abychom byli schopni aktivovat událost `halt`, je nutné definovat další element – `ihalt`, to je popsáno v sekci časování (`timing`), `halt` musí být také uveden v sekci použitých prvků (`use`). Dále jsou instrukce `ihalt` a `inop` umístěny do jedné skupiny (`set`) pojmenované `instructions`. Událost `main` je volána na začátku každého cyklu procesoru a jejím úkolem je začít s vykonáváním další instrukce uložené v paměti na adrese dané programovým čítačem [5].

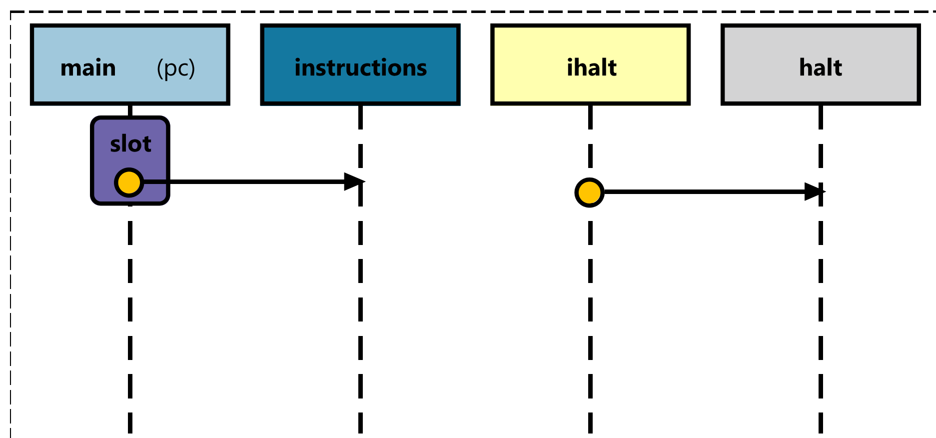
3.3 Grafické editory v Codasip Studio

Textový popis architektury může být, zejména pro začínající návrháře, poměrně složitý a nepřehledný na pochopení. Proto obsahuje vývojové prostředí Codasip Studio nástroje pro grafické modelování procesoru a jeho instrukční sady. Jedná se o sadu nástrojů, přesněji grafických editorů, které slouží pro popis architektury pomocí jednoduchých diagramů. Diagramy dohromady popisují kompletní procesor i jeho instrukční sadu, stejně jako jazyk CodAL, a jsou rozděleny podle sekcí jazyka na *instructionset diagram*, *structure diagram* a *timing diagram*. Časování (*timing*) není sice samostatnou sekcí jazyka, nicméně se jedná o velice důležitou součást popisu procesoru, a je opět přínosné tuto část popsat diagramem. Diagramy jsou rozděleny podle jednotlivých zaměření kvůli zvýšení přehlednosti. Stěžejní vlastností diagramů je schopnost generovat textový popis procesoru v jazyce CodAL. Generování textového popisu z diagramů probíhá téměř 1:1, protože diagramy zahrnují skoro veškeré vlastnosti a prvky jazyka CodAL. Při každém uložení digramu dojde k vygenerování kódu v jazyce CodAL. Ten je možné zobrazit a dále editovat v interaktivním editoru (jedná se o upravenou

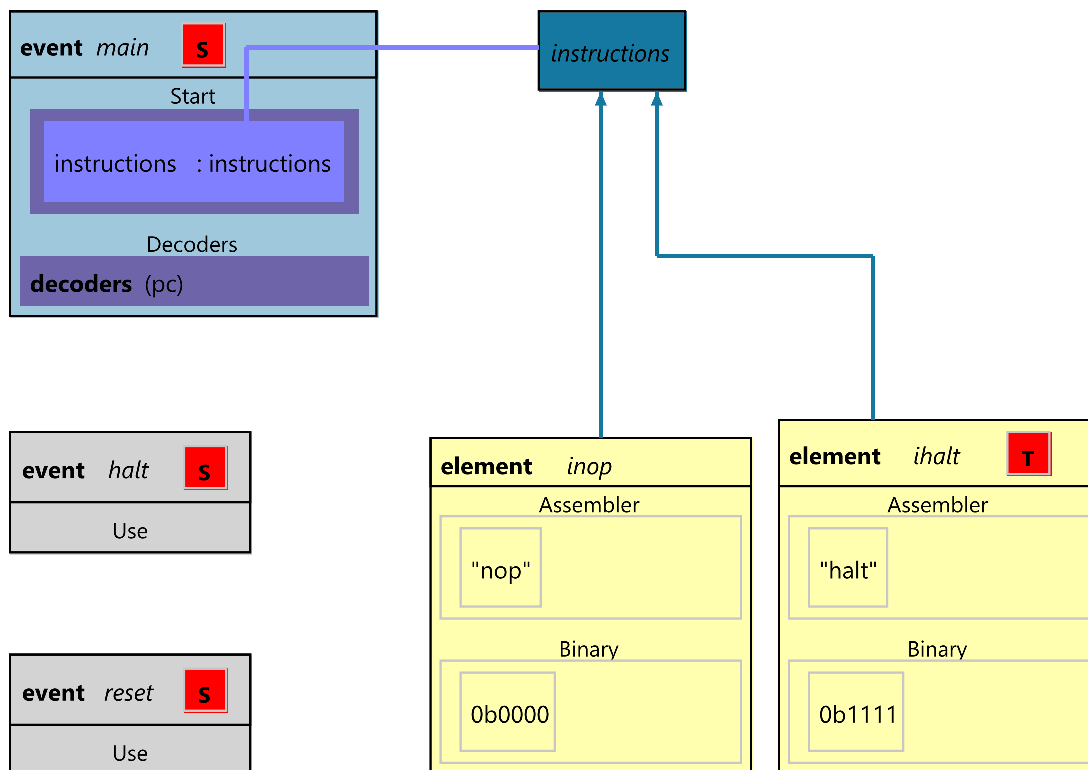
verzi CDT editoru [10], což je platforma pro vývoj C/C++ aplikací pod Eclipse). Diagramy dále poskytují možnost jednoduché validace, která je opět prováděna při každém uložení. Validace hlídá, zda jsou v diagramu dodrženy základní pravidla jazyka CodAL. Pokud některý z prvků pravidla nesplňuje, prostředí o tom informuje návrháře pomocí visuálních značek a textového popisu chyby. Na následujících obrázcích je pomocí diagramů znázorněn model, který odpovídá výše uvedenému kódu.



Obrázek 2: Ukázka structure diagramu



Obrázek 3: Ukázka timing diagramu

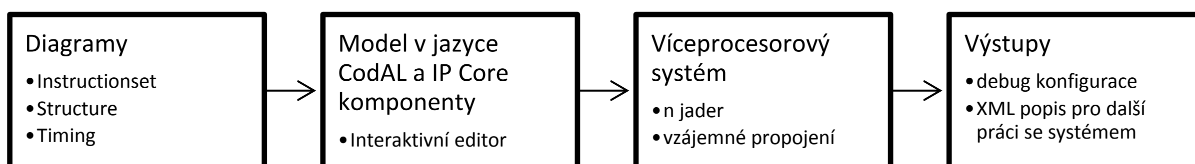


Obrázek 4: Ukázka instructionset diagramu

Práce s diagramy je velice jednoduchá a intuitivní. Vývojář umisťuje jednotlivé prvky z palety nástrojů na pracovní plochu, kde také může editovat většinu vlastností a atributů prvků. Některé prvky (například *set* a *element*) lze propojovat pomocí spojení. Takové prvky mají většinou ve své kontextové nabídce možnost skrýt všechna spojení, což je užitečné u větších procesorů. U prvku *set* lze takto skrýt nejen spojení, ale také všechny potomky v hierarchické struktuře směrem dolů. Diagramy timing a instructionset jsou navzájem propojeny a přístup z jednoho do druhého je usnadněn pomocí poklepání na ikonu *T* (v instructionset) nebo přímo na daný v prvek (v timing). Některé sekce lze pro přehlednost skrývat pomocí tlačítek se symboly plus a minus. Sekce popisující chování (*semantics*) instrukce (a také speciální sekce *decoders*) je popsána ve speciálním textovém souboru, každá instrukce s touto sekcí má svůj externí soubor, kam vývojář zapisuje popis chování v podmnožině jazyka ANSI C. Tyto soubory jsou s diagramem svázány stejným způsobem, jako jsou provázány timing a instructionset diagramy [8]. Z uvedeného popisu grafických editorů je vidět, že jejich úroveň uživatelské přívětivosti je poměrně vysoká, stejným směrem by se měl ubírat budoucí nástroj pro grafický návrh víceprocesorových systémů. Diagramy jsou plně integrovány do prostředí Codasip Studio a jsou implementovány v jazyce Java na platformě Eclipse [11] [12] pomocí nástrojů EMF (Eclipse Modeling Framework) [13] a GMF (Graphical Modeling Framework) [14].

4 Analýza a návrh řešení

Nástroj pro modelování víceprocesorových systémů na čipu bude plně integrován do prostředí Codasip Studio a bude propojen se současnými nástroji pro grafický i textový návrh procesoru. Nástroj bude pracovat s modely procesorů resp. standardních komponent, které budou blíže specifikovány popisem v jazyce CodAL nebo pomocí ekvivalentních diagramů, resp. standardizovaným textovým popisem. Obrázek 5 ilustruje práci na návrhu víceprocesorových systémů. Vývojář víceprocesorových systémů ale není povinen jednotlivé komponenty stavět úplně na zelené louce, v návrhu víceprocesorových systémů jsou jednotlivé komponenty chápány jako zapouzdřené objekty, pro které je zásadní popis jejich rozhraní a funkce, kterou transformují vstupy na výstupy. Proto je žádoucí, aby tento fakt budoucí nástroj respektoval, z tohoto důvodu jsem navrhl několik podpůrných nástrojů, které vývoj víceprocesorových systémů dále urychlí a zjednoduší. Jedná se o sadu dvou dialogových průvodců, kteří jsou plně integrováni do prostředí Codasip Studio. První z nich umožňuje rychle nakonfigurovat a vytvořit jednoduchý validní model procesoru, druhý umožňuje generování standardních komponent se zvoleným rozhraním. Další způsob, jak může vývojář získat komponenty pro svůj projekt, je databáze hotových řešení na serveru projektu Lissom. Přístup k této databázi je integrován přímo do prostředí Codasip Studio, pomocí tzv. *View* [12]. Výhodou je, že tyto komponenty jsou stále aktuální a uživatel si je pak pomocí *drag&drop*[1] pouze umístí do svého projektu, kde je bude možné dále upravovat. Návrh těchto nástrojů je popsán v kapitole 4.5, jejich implementací se zabývá kapitola 5. Uživatel tak bude mít k dispozici řadu možností, jak hotovou komponentu získat a použít ji v návrhu svého systému.



Obrázek 5: Vývoj víceprocesorového systému

4.1 UML diagram komponent

V předchozích kapitolách byl několikrát zmíněn pojem *komponenta*, v našem případě se jedná o komponentu hardwarového charakteru. Obecná komponenta je fyzickou nahraditelnou částí systému, která obaluje implementaci a poskytuje realizaci specifikovaných rozhraní [15]. Komponenty mohou obsahovat mnoho tříd a realizovat velké množství rozhraní [16]. Takto jsou definovány komponenty

[1] Jedná se o systém hojně využívaný v grafických uživatelských rozhraních. Jednotlivé grafické objekty je možné přemisťovat pomocí táhnutí myší.

z pohledu UML (*Unified Modeling Language*) *diagramu komponent*. Z těchto definic je zřejmá jistá podobnost mezi hardwarovou komponentou a komponentou z diagramu komponent. V zásadě se jedná o tyto vlastnosti:

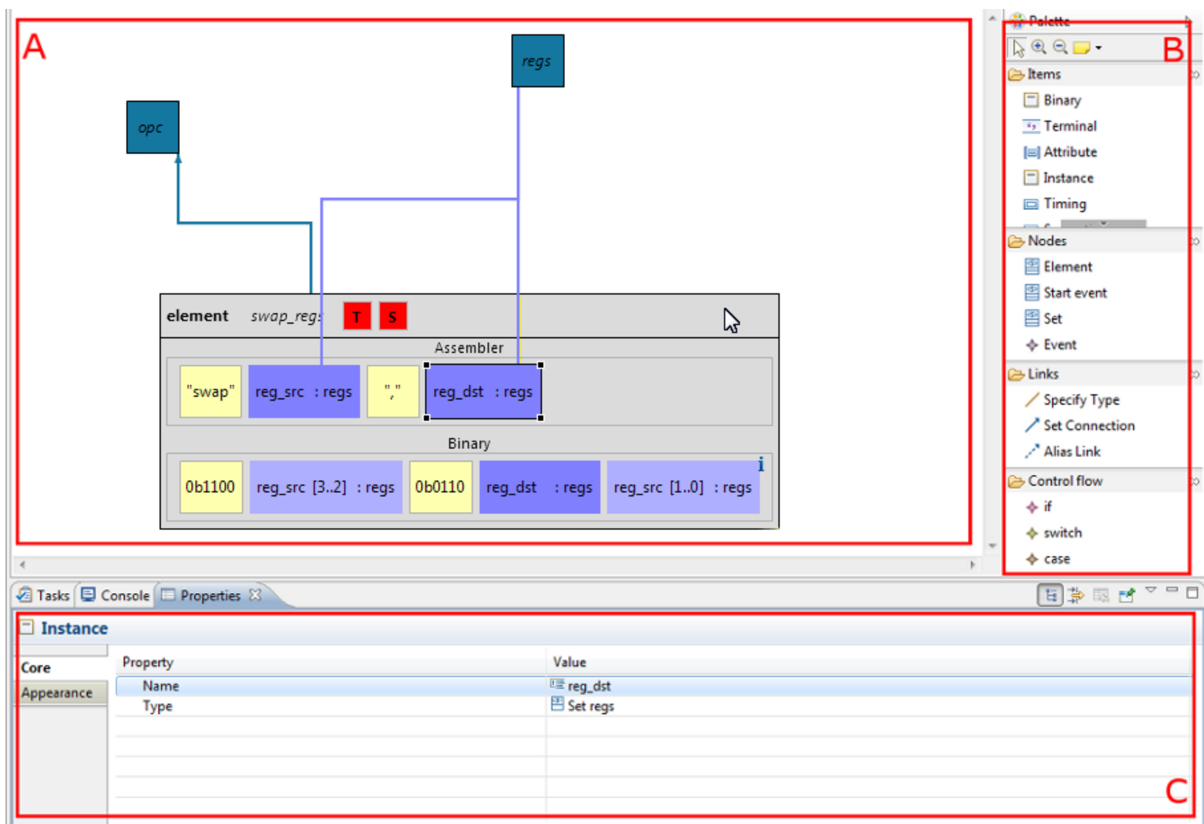
- zapouzdřenost, black box,
- modularita a nezávislost,
- znovupoužitelnost.

Všechny tyto vlastnosti vycházejí jak z objektově orientovaného přístupu k programování, tak z vlastností víceprocesorových systémů, kde jsou hlavním důvodem, proč tyto systémy vůbec zavádět. Výše popsané paralely bych rád využil při návrhu grafického nástroje pro tvorbu víceprocesorových systémů zejména proto, že je tento přístup osvědčený a velmi dobře zdokumentovaný. Diagram komponent kromě definice samotných komponent definuje také závislosti mezi komponentami. Nejdůležitější z nich jsou *rozhraní* a *porty*. Rozhraní je způsob komunikace, který podporuje modularitu a znovupoužitelnost. Obecně se jedná o seznam vlastností, které daná komponenta nabízí jiným komponentám. V diagramu je tato skutečnost znázorněna symboly *vyžaduje rozhraní* a *nabízí rozhraní*. Komponenty, které vyžadují rozhraní tak nejsou závislé na konkrétních komponentách, ale pouze na těch, které požadované rozhraní nabízejí, a pro jejich činnost je marginální, která z možných komponent rozhraní poskytne. Porty jsou volitelnou součástí každé komponenty a sdružují rozhraní do skupin.

4.2 Vlastní návrh nástroje

Hlavními požadavky na vývojový grafický nástroj jsou rychlost, srozumitelnost a uživatelský komfort. Tyto požadavky by měly být zajištěny zejména grafickým přístupem k návrhu víceprocesorových systémů, o čemž pojednává tato kapitola, a dále jednoduchým přístupem k hotovým řešením komponent, na kterých lze celý systém stavět (kapitola 4.5). V kapitole 4.1 byl popsán diagram komponent a jeho podobnost s problémem modelování víceprocesorových systémů, proto je budoucí nástroj inspirován právě touto notací. Popis komponent podle specifikace UML se přesně shoduje s vlastnostmi komponent víceprocesorových systémů. Jak již bylo řečeno, nový nástroj bude integrován do prostředí Codasip Studio (platforma Eclipse), kde již funguje řada diagramových editorů, které jsou postaveny na nástrojích GMF a EMF. Je proto vhodné použít osvědčenou kombinaci a navrhnout a implementovat nový nástroj pomocí nich. Stěžejním krokem v návrhu diagramového nástroje v GMF je specifikace jeho *Ecore metamodelu* [13], který zobrazuje vztahy jednotlivých objektů, jejich závislosti a atributy. Tento metamodel je popsán pomocí diagramu, který vychází z diagramu tříd UML. Více se modelováním v EMF a GMF zabývá kapitola 4.4. Práce s takto vytvořeným diagramovým nástrojem vypadá následovně: okno aplikace je rozděleno v zásadě na 3 hlavní části – *plátno*, *nástrojovou paletu* a zobrazení *vlastností* vybraného objektu (viz obrázek 6). Uživatel pak vybírá objekty z palety nástrojů a umisťuje je na plátno, kde

také může upravovat většinu atributů a vlastností objektů. Dále je možné pomocí nástrojů z palety vytvářet spojení jednotlivých existujících objektů. Přehled všech atributů objektu je zobrazen v pohledu vlastnosti, kde je možné také tyto vlastnosti editovat.



Obrázek 6: Ukázka diagramového nástroje GMF (A - plátno diagramu, B - nástrojová lišta, C - vlastnosti)

Na plátno nástroje bude možné umisťovat jak *modely procesorů* popsané v jazyce CodAL (jednotlivým procesorům odpovídají hardwarové projekty v Codasip Studiu), tak *standardní komponenty*, jako jsou registry, hardwarové implementace zásobníků, FIFO, atd. Pro účely grafického modelování je nutný pouze popis rozhraní těchto komponent (tedy *porty*, jejich *šířky* a *směry*), ten bude umístěn v samostatných textových souborech. Popisu standardních komponent se věnuje kapitola 4.3. Úkolem grafického nástroje bude poskytovat uživateli aktuální nabídku dostupných komponent, které se nalézají v pracovním prostředí Codasip Studia. Jednotlivé komponenty budou nabízeny formou nástrojů v nástrojové liště. Uživatel tak bude moci umisťovat a propojovat konkrétní komponenty. Dále je nutné udržovat konzistenci mezi vzory komponent a jejich instancemi na plátně editoru. Jednotlivé změny v komponentách a hardwarových projektech bude nutné reflektovat také v modelech víceprocesorových systémů. Modely procesorů v jazyce CodAL je nutné před použitím a simulací přeložit do interní XML reprezentace, tento překlad zajistí také validaci a jeho výstupem je simulovatelný hardwarový projekt. Model víceprocesorového systému bude pracovat pouze s modely procesorů, které jsou validní a je možné je simulovat. Je tedy vyžadováno, aby byly před umístěním na plátno přeloženy.

Jednotlivé komponenty víceprocesorových systémů fyzicky komunikují přes vstupní a výstupní porty, které mají různou šířku. Tyto porty jsou navzájem propojeny a *výstupní porty* komponent tak posílají *signály vstupním portům* jiných komponent. Dále existují dva specifické signály, které jsou obvykle vedeny jako samostatné vstupní jednobitové porty – *CLK* (hodinový signál pro synchronizaci) a *RST* (reset). Dále je možné k jednomu výstupnímu portu připojit řadu vstupních (například spojení generátor hodinového signálu – CLK port). Typ portů (vstupní, výstupní) je možné řešit pomocí přístupu z *diagramu komponent,* tedy tak, že ke každému portu bude buď přiřazeno rozhraní, nebo naopak symbol, který bude zastávat funkci vyžadovaného rozhraní. Tento přístup je samozřejmě možné vynechat a v metamodelu diagramu specifikovat přímo dva typy portů – vstupní a výstupní. Je vhodné také přidat třetí typ portu, který je kombinací obou zmíněných – *vstupně-výstupní* porty. Typy portů je pak možné odlišit barevně nebo specifickým tvarem. Při návrhu diagramového nástroje jsem zvolil přístup, kdy jsou použity pouze barevně odlišené porty, nicméně v průběhu implementace a ve fázi prvních testování se může ukázat, že použití přístupu z diagramu komponent (za použití rozhraní) je z nějakého důvodu nutný nebo lepší. Dále je potřeba vyřešit problém různé šířky jednotlivých propojovaných portů, protože diagramový nástroj bude systémy modelovat pouze na úrovni spojení, nikoliv samotných portů. Nabízí se tři možnosti, jak tento problém řešit. První řešení uživateli vůbec nedovolí propojit nekompatibilní porty, druhým řešením je validace celého modelu (GMF umožňuje vizuálně označit chyby v modelu) a posledním řešením je benevolentní přístup, kdy dojde k propojení, ale některé piny zůstanou volné nebo nepřipojené. Toto řešení jsem po konzultaci s vedoucím práce zvolil. Při spojování portů tak mohou nastat obecně tři situace, pro které je nutné zavést pravidla chování:

- Bitová šířka výstupního portu (označme jako m) je větší než bitová šířka vstupního portu (označme jako n), se kterým je propojen. V takovém případě je k cílovému vstupnímu portu připojeno pouze n spodních bitů (počítáno od LSB).
- Opačný případ, tedy situace, kdy m je menší než n. V tomto případě je ke spodním m pinům vstupního portu připojen celý výstupní port a na nejvýznamnějších $n - m$ pinů vstupního portu je implicitně připojena logická nula.
- Oba porty jsou stejné šířky, všechny piny se spárují v pořadí od MSB k LSB.

Codasip Studio nabízí uživateli tzv. *debugovací perspektivu*. Jedná se o specifický pohled na aktuální projekty, který je přizpůsoben potřebám simulace a ladění procesorů. Účelem této perspektivy je *simulovat činnost* jednotlivých procesorů, na kterých může běžet obecně různé množství softwarových aplikací. Při simulaci běhu procesoru je možné sledovat stav hardwarových zdrojů a krokovat činnost. Celá perspektiva tak slouží zejména pro odhalení slabých míst systémů. Každý procesor má během simulace přidělen jeden ze softwarových projektů, které specifikují obslužnou činnost procesoru. Pro účely simulace víceprocesorových systémů je nutné pro každý procesor specifikovat aplikaci, která na daném procesoru poběží. Dále je nutné specifikovat, na jakém

hostitelském stroji bude simulace probíhat. Tato nastavení musí být zohledněna ve výsledném grafickém nástroji a je nutná jejich jednoduchá editace.

Výsledný model bude možné dále transformovat na výstupy, které se dále použijí ve vývojovém cyklu. Prvním výstupem je *konfigurace* pro výše zmíněnou debugovací perspektivu, tedy konfigurace (definice aplikace a stroje, na kterém simulace poběží) jednotlivých procesorů zúčastněných v modelu. Konfigurací může být pro jeden typ procesoru nebo celých víceprocesorových systémů více. Tyto konfigurace se mohou lišit jak v použitých aplikačních projektech, tak v detailech simulace. Během vývoje systému často dochází k simulacím různých konfigurací a při větším počtu zúčastněných komponent je poměrně pracné tyto konfigurace měnit. Je proto výhodné každou z těchto konfigurací ukládat a podle potřeby spouštět, persistenci takových konfigurací zajistí právě modely víceprocesorových systémů. Pro ukládání modelů víceprocesorových systémů a jejich konfigurací bude sloužit speciální typ projektu, který tak zastřeší současné hardwarové a softwarové projekty. Pro další práci s celým modelem je vhodné jej také popsat nějakým standardním způsobem, nejlépe pomocí značkovacího jazyka XML. Tento popis bude obsahovat instance všech komponent a procesorů v modelu a jejich propojení. Pokud shrneme výše zmíněné odstavce, budoucí nástroj bude poskytovat následující funkčnost:

- modelovat procesory, standardní komponenty a vazby mezi nimi,
- nastavovat simulační detaily jednotlivým instancím procesorů,
- vkládat na plátno jednotlivé procesory komponenty dostupné v souborovém systému vývojového prostředí Codasip Studio,
- dynamicky měnit nabídku komponent podle změn v souborovém systému vývojového prostředí,
- dynamicky měnit vlastnosti použitých komponent podle změn v jejich vzorech,
- umožnit generovat výstupy pro debugovací perspektivu a znovupoužitelný standardní popis v jazyce XML.

4.3 Popis standardních komponent

V předchozích kapitolách byla zmíněna takzvaná standardní komponenta, jinde je tento objekt nazýván také IP Core (*intellectual property core*). Obecně se jedná o jakoukoliv komponentu použitelnou ve víceprocesorovém systému, v případě Codasip Framework se jedná o komponentu třetích stran, kterou lze chápat jako černou skříňku, a na rozdíl od procesorů modelovaných jazykem CodAl nás nezajímá její vnitřní stavba. Tyto komponenty často bývají součástí různých externích knihoven a jejich obsah je tak opravdu často zapouzdřen. Pro účely modelování víceprocesorových systémů jsou tyto komponenty nezbytné. Jedná se například o zásobníky, hardwarové realizace front atd. Z hlediska víceprocesorových systémů je důležité zejména rozhraní, tedy popis vstupních, výstupní a vstupně-výstupních portů a jejich šířek.

K takovému popisu se hodí značkovací jazyk XML. XML je jednoduchý textový formát pro reprezentaci strukturovaných informací, jako jsou dokumenty, data, konfigurace, transakce atd. XML je odvozen od staršího standardu SGML (*Standard Generalized Markup Language*) a jedná se o jeden z nejrozšířenějších standardů pro sdílení strukturovaných dat mezi lidmi a programy [17]. V projektu Lissom je XML velice hojně využíváno k interpretaci a persistenci nejrůznějších dat. Proto je v zájmu práce se tohoto standardu držet a navrhnout strukturu XML dokumentu pro popis rozhraní standardních komponent. Tento popis pak bude jednak na výstupu průvodce pro vytváření standardních komponent, ale také na vstupu grafického editoru. XML kód komponenty bude samozřejmě možné podle potřeb upravovat. Konkrétní požadavky na popis komponent jsou následující:

- uchovat informace o počtu jednotlivých portů vstupních, výstupních a vstupně-výstupních,
- uchovat bitovou šířku každého portu, jeho směr a jeho identifikátor, který je jedinečný v rámci komponenty,
- uchovávat jméno komponenty a informace, zda obsahuje speciální jednobitové vstupní porty CLK a RST.

V počátcích této práce jsem vytvořil vlastní jednoduchou XML strukturu pro popis standardních komponent a v prvních verzích jsem ji pro persistenci komponentních rozhraní používal. Nicméně v průběhu práce se objevil požadavek na standardizaci tohoto popisu na úrovni mezinástrojové výměny dat. K tomuto účelu byl vybrán standard *IP-XACT* [18], který v budoucnosti nahradí většinu strukturovaných dokumentů XML v projektu Lissom. IP-XACT je *IEEE* standard pro popis elektronických komponent a jejich návrhů pomocí jazyka XML. Byl vytvořen konsorciem SPIRIT (Open SystemC Initiative). Cílem je sjednocení popisu komponent pomocí nezávislého formátu, který umožní výměnu knihoven návrhů mezi jednotlivými nástroji [7]. Tento standard je nesrovnatelně robustnější a univerzálnější proto obsahuje také některé další povinné informace, které v mém původním XML popisu nebyly obsaženy, protože jich nebylo potřeba. Následující XML kód názorně ilustruje popis stejné komponenty pomocí původního XML popisu a následně pomocí IP-XACT standardu.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<component clk="true" name="fifo" rst="true">
  <in id="input">4</in>
  <out id="output">4</out>
</component>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<spirit:component
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5">
  <spirit:name>fifo</spirit:name>
  <spirit:version>1.2</spirit:version>
  <spirit:vendor>codasip.com</spirit:vendor>
  <spirit:library>default</spirit:library>
```

```xml
  <spirit:model>
    <spirit:ports>
      <spirit:port>
        <spirit:portid>rst</spirit:portid>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:portid>clk</spirit:portid>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:portid>input</spirit:portid>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
          <spirit:vector>
            <spirit:left>3</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:portid>output</spirit:portid>
        <spirit:wire>
          <spirit:direction>out</spirit:direction>
          <spirit:vector>
            <spirit:left>3</spirit:left>
            <spirit:right>0</spirit:right>
          </spirit:vector>
        </spirit:wire>
      </spirit:port>
    </spirit:ports>
  </spirit:model>
</spirit:component>
```

Jedná se o jednoduchou komponentu reprezentující FIFO frontu. Komponenta obsahuje jak `rst`, tak `clk` porty, a dále jeden vstupní (`input`) a jeden výstupní (`output`) čtyřbitový port. Na první pohled je zřejmý rozdíl mezi standardem IP-XACT a mým vlastním řešením. Nicméně výhody a zejména rozšíření standardu IP-XACT jsou nesporné. Popis komponent je totiž jen malá část z množiny funkcí tohoto standardu. Za povšimnutí stojí poměrně vysoká redundance textu, která je u XML dokumentů typická. Všechny prvky standardu IP-XACT jsou uvozeny *prefixem*, ten je definován na začátku dokumentu pomocí tzv. *namespace*. V mém vlastním formátu jsem jméno a přítomnost `rst` a `clk` portů reprezentoval jako atributy kořenového uzlu dokumentu. Oproti tomu ve standardu IP-XACT se atributy nevyskytují a vše je reprezentováno pomocí uzlů a jejich hodnot. Speciální porty `rst` a `clk` je zde nutné reprezentovat jako obyčejné jednobitové vstupní porty s vyhrazenými identifikátory. Tuto skutečnost je také nutné reflektovat v průvodci pro vytváření standardních komponent a hlídat, zda uživatel nespecifikuje jako identifikátor portu některou z vyhrazených zkratek.

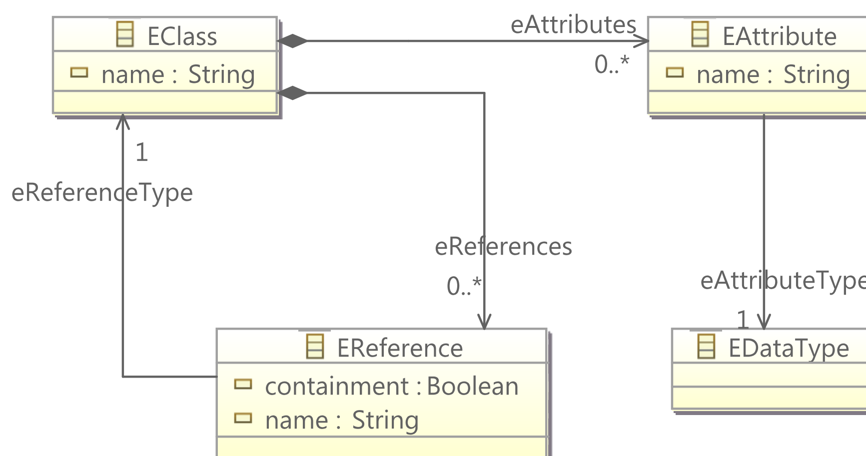
4.4 Úvod k nástrojům EMF a GMF

Diagramový nástroj bude vytvořen pomocí *Eclipse Modeling Framework* a *Graphical Modeling Framework*. Jedná se o nástroje, které jsou velice flexibilní a komplexní. Proto existuje nezměrné množství možností, které oba nástroje poskytují. EMF je naprosto nezávislý framework pro modelování *metamodelů*[2]. Zatímco GMF vychází z EMF a staví na metamodelech v něm vytvořených. Celý koncept funguje tak, že pomocí EMF je definován metamodel, ten je pak převzat GMF, kde je nutné nakonfigurovat několik souborů, které zajistí správné a požadované zobrazování diagramu. Někdy je nutné zasáhnout také do generovaného kódu, který bývá velice obsáhlý a složitý [9].

4.4.1 Eclipse Modeling Framework

Eclipse Modelling Framework je nástroj modelování, který umožňuje jednoduché generování kódu pro tvorbu aplikací založených na strukturovaných datových modelech. Nástroj poskytuje jednoduché generování struktury tříd implementovaných v jazyce Java. Mimo to poskytuje také základní nástroje pro jednoduché editování vytvořených modelů. Metamodely mohou být v EMF navrženy třemi způsoby. Prvním z nich je jednoduchá struktura tříd v Javě, které obsahují pouze atributy. Druhou možností je modelovat metamodel v jazyce XML a do třetice je možné vztahy v navrhovaném metamodelu definovat pomocí UML. Všechny tyto možnosti umí EMF zpracovat a vytvořit z nich tentýž metamodel. K unifikované reprezentaci metamodelů se používá jiný metamodel zvaný *Ecore*. Ten je svým vlastním metamodelem a je tedy možné jej považovat za *meta^{2}model*. Obrázek 7 znázorňuje jeho základní zjednodušenou strukturu. Všechny třídy metamodelu Ecore vycházejí ze základního objektu *EObject*. Pomocí Ecore metamodelu je možné definovat vlastní třídy (*EClass*), uvnitř těchto tříd atributy (*EAtributte*) libovolného typu (*EDataType*). Mezi třídami je možné definovat vztahy, které jsou reprezentovány třídou *EReference*. Přesněji, třída EReference reprezentuje vždy jeden konec asociace mezi třídami. Pro správu metamodelů má EMF velice jednoduché nástroje. Návrhář má možnost editovat metamodel v rámci diagramu tříd nebo pomocí jednoduchého editoru stromové struktury metamodelu. Výsledný metamodel je tak výše popsanou strukturou tříd EClass. K tomu, aby byla data modelů uchovávána perzistentně a bez zbytečných redundantních informací, slouží serializace modelů do souborů ve formátu XMI (*XML Metadata Interchange*). EMF disponuje funkcemi, které umožňují takto uložená data nejen ukládat, ale také načítat přímo do struktury objektů EObject [9].

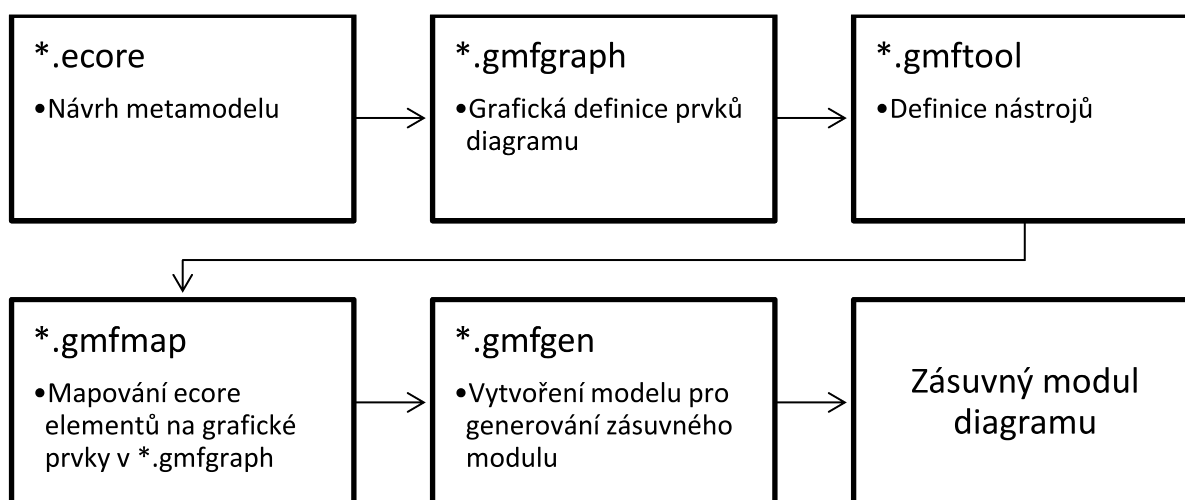
[2] Metamodel – specifikuje, jakým způsobem bude vypadat budoucí model, popisuje objekty a jejich vztahy, jejich vlastnosti, datové typy atd.



Obrázek 7: EMF meta²model

4.4.2 Graphical Modeling Framework

Graphical Modeling Framework je o poznání složitější než EMF a zatímco k EMF existuje dostupná oficiální literatura [13], GMF má k dispozici pouze několik tutoriálů a jednoduchou wiki. Proto je práce s GMF obtížnější a vývoj nástrojů je velice ovlivněn dostupností informací. GMF je framework, který spojuje nástroj pro práci se strukturovanými daty (EMF) a s nástrojem pro tvorbu grafických editorů GEF (*Graphical Editing Framework*) [19] v prostředí Eclipse. Jedná se o grafickou nástavbu EMF, přičemž nenutí uživatele porozumět GEF. Základní funkcí GMF je pomocí několika konfiguračních nástrojů vytvořit grafický editor diagramů na základě metamodelu Ecore. K tomu je využit Framework GEF. Na cestě od Ecore metamodelu k funkčnímu diagramu je několik kroků, které je nutné provést. Ty jsou znázorněny na obrázku 8. Soubor *gmfgragh* obsahuje stromovou strukturu, jejímž kořenem je objekt reprezentující celý diagram. Do něj je možné přidávat prvky, které pak bude diagram zobrazovat. Jako synovské uzly je možné těmto prvkům přiřazovat jednotlivé uzly, které rodičovskému uzlu přidávají požadované vlastnosti. Lze tak tedy ovlivnit barvy, tvar, popisky apod. Obsahem souboru *gmftool* je specifikace nástrojů, které se v diagramu objeví v paletě nástrojů. Zde je možné těmto nástrojům přiřadit ikony a seskupovat je do podskupin. Nejdůležitějším krokem je vytvoření souboru *gmfmap*, který spojuje *Ecore metamodel* s předešlými dvěma soubory (*.gmfgraph a *.gmftool). Zde uživatel specifikuje, jak budou objekty metamodelu zobrazeny, resp. na jaký grafický prvek budou vázány. Posledním krokem je soubor *gmfgen*, který vše předešlé zapouzdřuje. Z tohoto souboru se automaticky vygeneruje kód zásuvného modulu. V tomto kódu je možné provádět úpravy, které předešlé nástroje neumožňují. Jedná se hlavně o vlastnosti zobrazení některých prvků apod. [9].



Obrázek 8: Proces vývoje GMF diagramového editoru

4.5 Podpůrné nástroje

4.5.1 Interaktivní průvodci

Na začátku kapitoly 4 jsem zmínil, že pro zjednodušení práce s grafickým editorem víceprocesorových systémů vznikly dva typy dialogových průvodců (tzv. průvodce typu *next, next, finish*). Tito průvodci jsou k dispozici v nabídce v prostředí Codasip Studio. Prvním z nich je tzv. *Simple Component Wizard*, jehož primárním účelem je vygenerovat komponentu umístitelnou do diagramového nástroje pro modelování víceprocesorových systémů. Z předchozích kapitol je zřejmé, že pro tuto komponentu je zásadní počet vstupních a výstupních portů, jejich bitová šířka, jejich unikátní identifikátor a to, zda bude obsahovat speciální porty RST a CLK. Tyto vlastnosti je v průvodci nutné specifikovat společně s umístěním výstupního souboru. Ten prošel jistým vývojem, jak bylo popsáno v kapitole 4.3, a je ve formátu, který odpovídá standardu IP-XACT. S tímto souborem se dále pracuje, zejména v rámci grafického editoru, kde je možné novou komponentu vizualizovat a umisťovat v grafech systémů na čipu. Druhým implementovaným průvodcem je tzv. *CPU Wizard*, jehož výstupem je zejména model procesoru v jazyce CodAL. V kapitole 3.2 byly uvedeny základní požadavky na minimální obsah sekcí resources a instructionset. V souhrnu se jedná o tyto prvky (některé nejsou povinné):

- události (prvky event) main, reset a halt,
- jednoduchá instrukce (prvek element) a skupina (set), do které patří,
- paměť typu ram a její jednoduché mapování,
- speciální registr pro *programový čítač* a jeden další pomocný registr,
- pipeline (pokud se jedná o cycle acurate model).

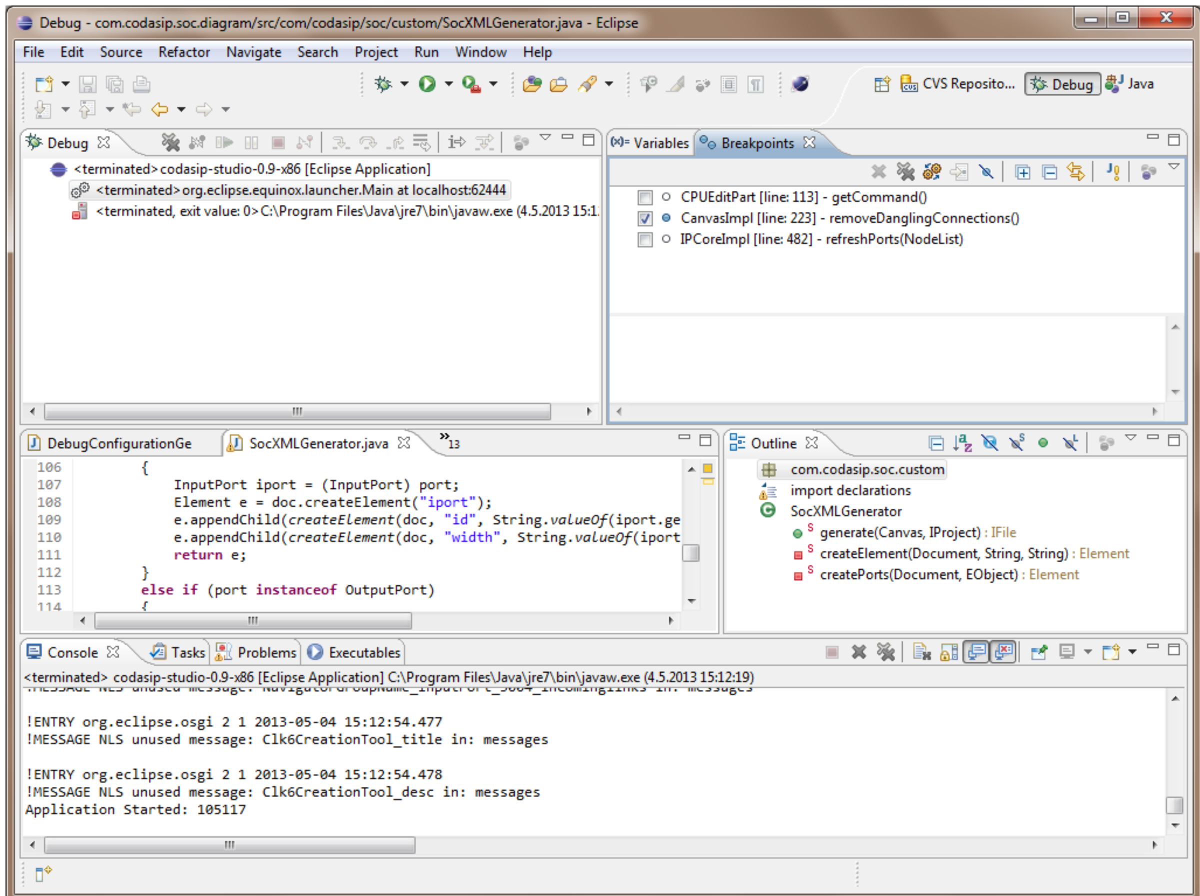
V tomto průvodci má uživatel možnost na několika stranách specifikovat vlastnosti povinných částí procesoru. Ty se mohou lišit podle toho, zda se jedná o instruction acurate nebo cycle acurate

model. Výstupem tohoto průvodce je funkční model procesoru v jazyce CodAL nebo také diagramy jednotlivých částí modelu (pokud uživatel zvolí, že je chce také generovat). Výsledný model je plně validní a lze jej bez chyby přeložit do interního XML modelu, se kterým lze dále v prostředí pracovat, například spouštět simulace. K tomu, aby překlad proběhl bez problémů, musí být model validní nejen po syntaktické, ale hlavně po sémantické stránce. Během generování z průvodce je nutné dopočítat a doplnit několik atributů, například adresový rozsah v mapování paměti se vypočítá ze zadaných atributů paměti ram. Podobně se doplní šířka programového čítače a pomocného registru. Pokud uživatel zvolí, že se jedná o cycle acurate model, je na poslední stránce průvodce možné specifikovat počet stupňů pipeline a jejich jména. Poté dojde k vygenerování této pipeline ve structure diagramu, a navíc je ke každému stupni pipeline vygenerována jedna událost v instructionset diagramu.

4.5.2 Online projekty

Poměrně zásadním zjednodušením při práci v prostředí Codasip Studio je přístup k online projektům a hotovým řešením, které jsou pro registrované uživatele dostupné na serveru projektu Lissom. Mezi nabízenými projekty se nachází jak hardwarové platformy, tak aplikační projekty, které lze na těchto platformách spouštět. Mimo to tato databáze obsahuje také dokumentaci a manuály. Doposud byly tyto projekty k dispozici pouze přes webové rozhraní, uživatelé je nejprve museli stáhnout do lokálního úložiště přes webový prohlížeč a následně importovat do prostředí Codasip Studio. Jedním z cílů této práce je umožnit procházení a importování hotových řešení přímo v rámci vývojového prostředí. K tomu platforma Eclipse nabízí několik možností. Okno celého prostředí lze rozdělit na několik tzv. pohledů - *Views*, které lze libovolně umisťovat a měnit jejich velikost (na obrázku 9 je pro ilustraci znázorněna ladící perspektiva Eclipse, která využívá celou řadu pohledů). Mou snahou je tedy implementovat a poskytnout takový pohled, který umožní výběr, stažení a import hotového řešení přímo do Codasip Studia. Podobnou funkcionalitu lze nalézt i u nástroje *XMOS* [20] s podobným zaměřením, který je navíc také vyvíjen na platformě Eclipse. Pro přístup k hotovým řešením se musí uživatel autentizovat, následnou autorizací je mu přidělen výčet hotových řešení, ke kterým má přístup. O tom jaké projekty má daný uživatel v nabídce k dispozici rozhodne úroveň jeho oprávnění v projektu Lissom. Výhodou tohoto řešení je aktuálnost dostupných projektů, celková pružnost a jednoduchý způsob rozšiřování této databáze mezi uživatele. Ti nemusí nic explicitně stahovat (např. z webu projektu) a poté samostatně exportovat mezi ostatní projekty, ale budou mít vše pod jednou střechou ve vývojovém prostředí. Uživateli se tak stanou hotové projekty dostupnější. Podle libosti si je pak může stahovat do svého vlastního pracovního prostoru, kde je má možnost dále upravovat a pracovat s nimi při návrhu systému. Během stahování projektu z databáze totiž proběhne vytvoření lokální kopie projektu, ke které pak má uživatel plný přístup. Pokud se rozhodne pro

opětovné stažení téhož projektu z databáze znovu, je nutné, aby byl vyzván, zda chce původní (upravenou) verzi projektu přepsat.



Obrázek 9: Ukázka rozdělení okna na několik pohledů

5 Implementace

Předchozí kapitoly shrnuly požadavky na implementované nástroje a navrhly řešení jak tyto požadavky dodržet. Tato kapitola popisuje mou práci na implementaci navržených řešení. Vzhledem k tomu, že tato práce si klade za cíl vytvořit celou sadu nástrojů, je tato kapitola rozčleněna do podkapitol, které se věnují jednotlivým řešením samostatně. Implementace spolu s použitými technologiemi s sebou přináší spousty nových poznatků, ty nejdůležitější jsem se snažil uvést a vysvětlit.

5.1 Interaktivní průvodci

Tato kapitola se věnuje implementaci dříve popsaných dialogových průvodců – *CPU Wizard* a *Simple Component Wizard*. Tyto dialogy je možné v Eclipse implementovat jako zásuvné moduly. Každý zásuvný modul je specifikován souborem *plugin.xml*, kde je mimo jiné uvedeno, jakým způsobem platformu rozšiřuje a které třídy se o toto rozšíření starají. Následující XML kód ilustruje, jak je popsán zásuvný modul, který platformu rozšiřuje o CPU Wizard.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         point="org.eclipse.ui.newWizards">
      <wizard
            name="CPU wizard"
            icon="icons/sample.gif"
            category="com.codasip.ui.wizards.category"
            class="com.codasip.wizard.NewCPUWizard"
            id="com.codasip.wizard.SampleNewWizard">
      </wizard>
   </extension>
</plugin>
```

Tzv. `extension point` specifikuje, o jaký zásuvný modul se jedná, `category` specifikuje do jaké kategorie se má nový průvodce umístit, dále je možné specifikovat zobrazované jméno (`name`) a Java třídu (`class`), která se stará o implementaci tohoto zásuvného modulu. O zásuvný modul obsahující průvodce CPU Wizard se stará třída `NewCPUWizard`, ta musí implementovat rozhraní `IWizard` a dále rozšiřuje třídu `Wizard`. Celý průvodce je složen ze stránek tzv. `WizardPage`, které implementuje stejnojmenná abstraktní třída. Ty se ve třídě implementující průvodce skládají do seznamu stránek. Pro dosažení vlastních stránek je nutné tuto třídu rozšířit. Při inicializaci instance třídy `Wizard` je pak nutné vytvořit seznam stránek, které bude průvodce obsahovat. Jednotlivé stránky obsahují metodu `createControl`, která je zodpovědná za všechny objekty grafického uživatelského rozhraní (GUI) na stránce. Na stránku lze umisťovat pomocí tzv. *layout* (rozložení) všechny objekty z Eclipse knihovny *SWT* [21]. Následující kód ilustruje implementaci metody

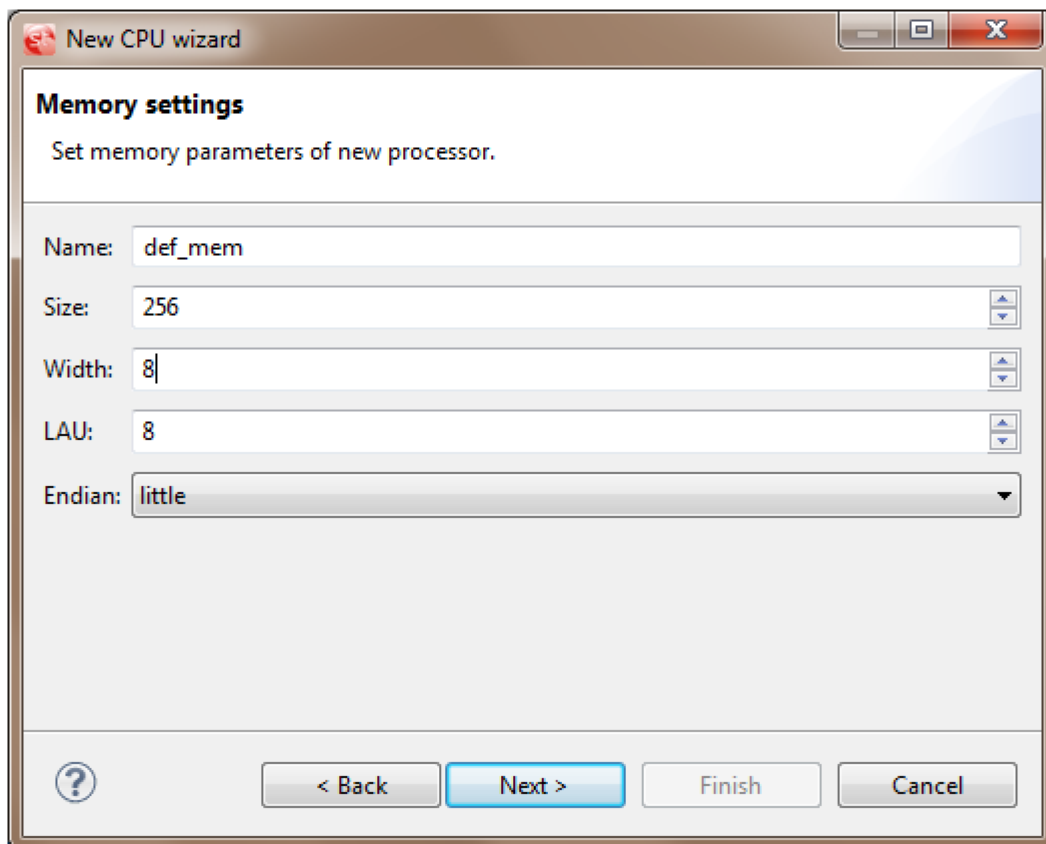
`createControl` třídou `MemoryCPUWizardPage`, která je zodpovědná za nastavení atributů paměti ram.

```
@Override
public void createControl(Composite parent)
{
	// create main composite with grid layout (2 columns, 9px spacing)
	Composite container = new Composite(parent, SWT.NULL);
	GridLayout layout = new GridLayout();
	container.setLayout(layout);
	layout.numColumns = 2;
	layout.verticalSpacing = 9;
	// label and text to field for memory name, inserted into parent container
	Label l = new Label(container, SWT.NULL);
	l.setText("Name:");
	nameText = new Text(container, SWT.BORDER | SWT.SINGLE);
	nameText.setText("def_mem");
	nameText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
	nameText.addModifyListener(new ModifyListener()
	{
		public void modifyText(ModifyEvent e)
		{
			// respond to text modificaion
			dialogChanged();
		}
	});

	l = new Label(container, SWT.NULL);
	l.setText("Size:");
	l.setToolTipText("Number of words");
	sizeSpinner = new Spinner(container, SWT.BORDER | SWT.SINGLE);
	sizeSpinner.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
	sizeSpinner.setMinimum(1);

	...
	// this method handles mainly inputs validation
	dialogChanged();
	setControl(container);
}
```

Stránka je tvořena *kontejnerem*, do kterého lze libovolně (podle pravidel SWT) vkládat další objekty a kontejnery. V tomto případě jsou objekty skládány podle rozložení `GridLayout` do mřížky. Do tohoto rozložení jsou postupně vkládány nápisy, textová pole atd. V kódu je vytvořeno také textové pole pro editaci jména paměti `nameText`. K tomuto poli je přiřazen a implementován tzv. *listener* (posluchač), který je volán pokaždé, když dojde ke změně obsahu textového pole. V tomto případě je nutné zkontrolovat, zda jméno není prázdné. Dále je do kontejneru vložen tzv. *spinner* pro nastavení počtu slov paměti, další prvky jsou pro zjednodušení vynechány. Na konci metody je nutné zaregistrovat *kontejner* jako hlavní objekt grafického uživatelského rozhraní této stránky. Výsledná stránka je znázorněna na obrázku 10. Obdobným způsobem jsou implementovány všechny stránky obou průvodců.

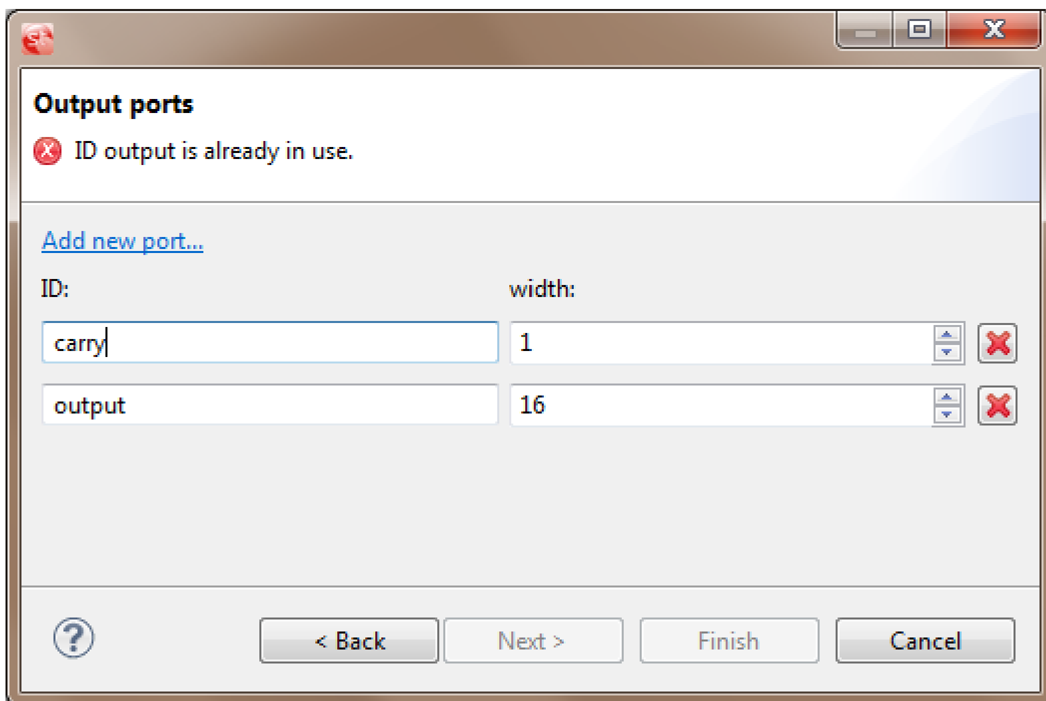


Obrázek 10: Stránka průvodce CPU Wizard

Pořadí stránek je možné modifikovat podle voleb, které uživatel udělá během procházení průvodcem. Toho jsem využil například v případě CPU Wizard, kdy je nutné reagovat na to, zda uživatel zvolil cycle acurate nebo instruction acurate model. Tuto volbu uživatel provádí na první stránce průvodce, nicméně každá z nich vyžaduje v pozdější fázi průvodce zobrazení své vlastní stránky. Při inicializaci průvodce je nutné vytvořit a registrovat obě tyto stránky, protože ještě není jasné, kterou uživatel zvolí. Poté co je jeho rozhodnutí jasné, je možné měnit následující zobrazovanou stránku pomocí implementace metody `getNextPage`.

Další možnost, kterou nabízí třída `WizardPage,` je validace uživatelských vstupů a zablokování dalšího postupu, dokud nejsou zadány validní hodnoty. K tomuto účelu je možné použít metody `setErrorMessage` a `setPageComplete`. Za zmínku stojí využití této možnosti v Simple Compoment Wizard, kde je nutné kontrolovat unikátnost zadaných identifikátorů portů. K tomuto účelu je nutné předávat si data mezi jednotlivými stránkami pomocí speciálního objektu, v tomto případě se jedná o objekt typu `HashMap<String, Integer>`. Průvodce totiž obsahuje odděleně stránky pro vstupní, výstupní a vstupně-výstupní porty, nicméně identifikátory musí být jedinečné v rámci celé komponenty. Do zmíněné mapy se ukládají použité identifikátory a jejich četnost. Tímto způsobem je možné detekovat unikátnost identifikátoru napříč stránkami průvodce. Na obrázku 11 je znázorněna stránka pro editaci výstupních portů komponenty s chybovou hláškou informující o použití již použitého identifikátoru. Po přechodu na IP-XACT bylo nutné přidat také kontrolu, zda uživatel nezadává jako identifikátor zkratku *rst* nebo *clk*. Jak bylo popsáno v kapitole 4.3, IP-XACT

chápe clock a reset porty jako obecné vstupní porty šířky jedna, a proto je nutné je do modelu komponenty vygenerovat společně s ostatními porty a zkratky pro rst a clk označit jako vyhrazená slova.



Obrázek 11: Pokus o přidání již existujícího identifikátoru „output"

Po ukončení průvodce tlačítkem *Finish* je automaticky volána metoda `performFinish` ve třídě `Wizard`. Tuto metodu je nutné implementovat, obyčejně se načtou posbíraná data ze všech stránek průvodce, zpracují se a je z nich vygenerován soubor. Data zapsaná na jednotlivých stránkách je nutné získat přes veřejné metody, které je nutné implementovat. Pokud je výstupem průvodce soubor na disku, je vhodné zvážit, zda obsah metody `performFinish` nespustit ve zvláštním vlákně tak, aby nedošlo k zamrznutí celého prostředí. Uživatel je navíc o průběhu generování informován pomocí textových popisků. Takto to funguje i v případě obou dialogových průvodců. V případě Simple Component Wizard je do zvoleného umístění vygenerován soubor obsahující nastavené parametry nové komponenty shodný se standardem IP-XACT. Ke generování XML výstupu je vhodné použít některou z dostupných knihoven, kterých je v Javě celá řada. Já jsem zvolil knihovny z veřejně dostupného balíku `org.w3c.dom`, které implementují XML pomocí rozhraní DOM (*Domain Object Model*) [22]. Vytvořený model je pak možné zapsat pomocí tříd z balíku `javax.xml`.

V případě CPU Wizard je situace poněkud složitější. Nejprve je nutné vytvořit modely všech tří diagramů a vytvořit v nich objekty s požadovanými parametry. To je možné provést pouze pomocí tzv. *commands*, které jsou typické pro EMF a GMF operace (jedná se o atomické transakce upravující model). Vytváření jednotlivých EMF objektů je navíc nutné provést pomocí *továrních metod* (návrhový vzor *Factory* [23]), které jsou generovány automaticky pro každý EMF objekt

metamodelu. Činnost v doménových modelech je automaticky pozorována GMF, které vytváří odpovídající objekty v diagramech. V této chvíli je nutné dopočítat některé údaje z dat, která vložil uživatel (např. velikost paměti v mapovaném adresovém prostoru). Poté je nutné provést rozmístění objektů v diagramech, to je možné díky metodám z balíku GMF. O generování textového souboru s popisem v jazyce CodAL se stará třída `CodalConvertor`, která pomocí metod `toString()` vytiskne celou hierarchii doménových modelů a spojí je do jednoho souboru [9]. Pokud si uživatel přeje zachovat pouze soubor CodAL, jsou ostatní vygenerované soubory automaticky smazány.

5.2 Online projekty

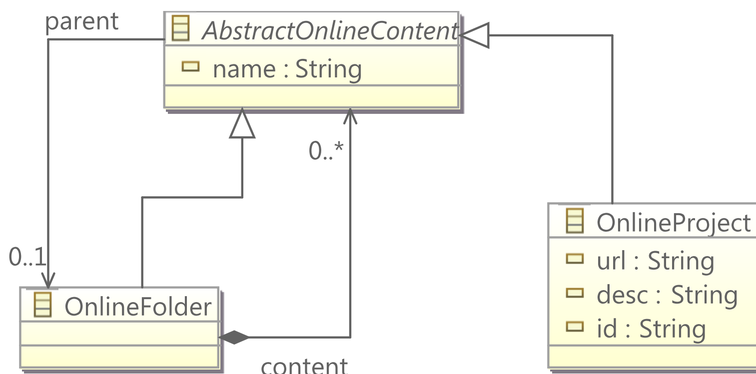
Tato kapitola popisuje problematiku implementace online projektů. Jak již bylo řečeno, Codasip Studio je postaveno na vývojovém prostředí Eclipse. Tato platforma je velice robustní a poskytuje řadu možností pro rozšíření. Okno aplikace postavené na platformě Eclipse je rozděleno na menší okna, která je možno uspořádat dle libosti. Tato okna jsou buď typu *editor* nebo *view*. Editor je hlavní okno pro editaci dat, view je typ okna pro zobrazování dodatečných informací. Platformu Eclipse je možno rozšířit jak o vlastní editor, tak o vlastní view. Jedná se opět o rozšíření ve formě zásuvného modulu – *pluginu*, podobně jako v případě interaktivních průvodců v kapitole 5.1. I v tomto případě je specifikace zásuvného modulu popsána v souboru plugin.xml, nicméně v tomto případě je atribut `extension point` nastaven na hodnotu `org.eclipse.ui.views`. Třída, která zajišťuje funkci tohoto zásuvného modulu, musí být potomkem abstraktní třídy `ViewPart` a implementovat některé její metody, zejména metodu `createPartControl`. Tato metoda obdrží jako svůj jediný parametr objekt typu `Composite`, což je jakýsi kontejner pro umísťování objektů grafického uživatelského rozhraní z knihovny SWT. Do tohoto objektu je v metodě `createPartControl` nutno vložit všechny požadované ovládací a prezentační prvky. Za rozmístění prvků uvnitř kontejneru zodpovídá třída `GridLayout`, která umisťuje své potomky do mřížky libovolných rozměrů.

5.2.1 Grafické uživatelské rozhraní a datová struktura

V případě online projektů je hlavní třídou realizující zásuvný modul třída `OnlineProjectView`. Vytváření a umísťování ovládacích prvků probíhá podobně jako v případě inicializace stránek interaktivních průvodců v kapitole 5.1. Nicméně zde je situace poněkud složitější. Jak bylo popsáno v kapitole 4.5.2, pro přístup k online projektům je vyžadováno přihlášení k serveru projektu Lissom, `OnlineProjektView` proto musí zajistit jednak přihlášení a poté až zobrazení dostupných projektů. Implementačně je tento problém vytvořen následovně: třída `OnlineProjektView` obsahuje dva privátní atributy již zmíněného typu `Composite` – `loginComposite` a `viewerComposite`. První z uvedených objektů obsahuje textová pole a tlačítko pro přihlášení, druhý objekt obsahuje grafické uživatelské rozhraní pro zobrazování dostupných projektů. Po otevření pohledu (view) je do hlavního

kontejneru pohledu vložen kontejner `loginComposite` a uživatel má možnost se přihlásit. Po úspěšném přihlášení a stažení stromu projektů ze serveru je `loginComposite` zahozen a do hlavního kontejneru pohledu vložen `viewerComposite`. Opačná situace nastává při odhlášení uživatele.

Kontejner pro zobrazení stromu projektů obsahuje jednak textové pole pro vyhledávání v projektech, ale hlavně grafický ovládací prvek `TreeViewer`, který je zodpovědný za zobrazení stromových struktur. Tomuto objektu je potřeba předat vlastní třídu implementující rozhraní `ITreeContentProvider`, což je rozhraní, které realizuje roli kontroléru, tedy jakési vrstvy mezi daty na jedné straně a grafickým uživatelským rozhraním na straně druhé. Je nutné implementovat metody, které pro předaný prvek stromové struktury vrátí rodiče, pole potomků atd. Datová struktura stromu projektů je podobná adresářové struktuře souborového systému. Obsahuje dva základní prvky – adresář a projekt. Projekty jsou podle typu a vlastností tříděny hierarchicky do adresářů. Diagram tříd na obrázku 12 ilustruje vztahy v datové struktuře modelující strom projektů. Každý adresář může obsahovat libovolný počet jiných adresářů a projektů, ty jsou specifikovány identifikátorem, jménem, popisem a adresou, ze které je možno je stáhnout. Adresa a identifikátor jsou ukládány pro vnitřní potřebu a uživateli nejsou zobrazeny. Při poklepání na konkrétní projekt dojde k jeho stažení a importu do souborového systému vývojového prostředí, tento proces je detailně popsán v následující kapitole.



Obrázek 12: Diagram tříd stromové struktury projektů

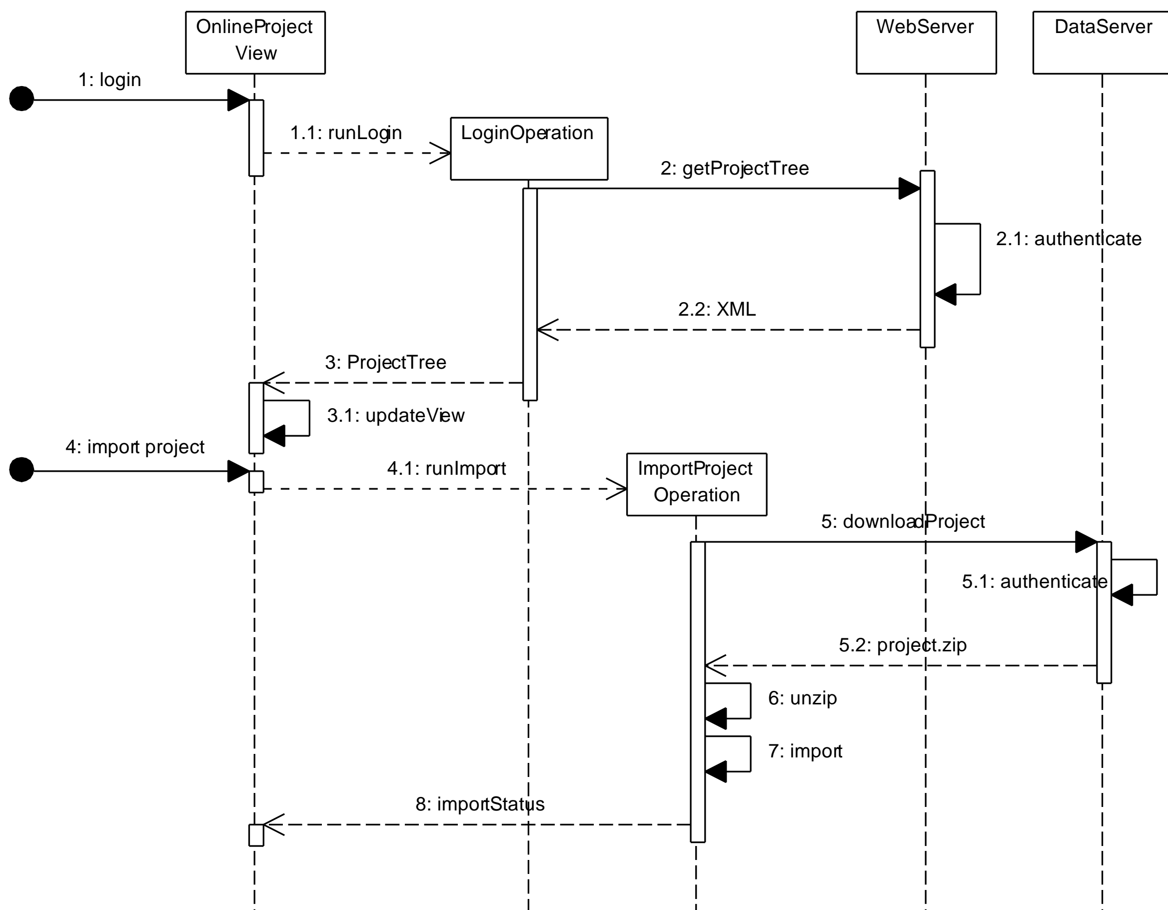
5.2.2 Síťová komunikace

Komunikace se serverem je poměrně časově náročná operace, a proto je implementována jako samostatná třída implementující rozhraní `IRunnableWithProgress`. Třídy implementující toto rozhraní je možné spustit pomocí metody `run` ve speciálním dialogovém okně, skrze které je uživatel informován o průběhu operace a může zde tuto probíhající operaci kdykoli ukončit. Z předchozích kapitol je zřejmé, že uživatel se musí při komunikaci se serverem autentizovat. Autentizace se servery projektu Lissom probíhá přes protokol *Http Digest* [24]. Pro tuto komunikaci jsem zvolil volně dostupnou implementaci http klienta z knihovny *Apache HttpComponents* [25]. V této knihovně se nachází několik implementací http klientů připravených k použití. Je nutné pouze vytvořit instanci

klienta a správně jej nakonfigurovat, včetně autentizačních údajů. Následující fragment kódu znázorňuje konfiguraci a použití takového klienta.

```
// create http host with the desired parameters
HttpHost targetHost = new HttpHost("www.codasip.com", 443, "https");
DefaultHttpClient httpclient = new DefaultHttpClient();
try
{
	// set credentials to login
	httpclient.getCredentialsProvider().setCredentials
	(
		new AuthScope(targetHost.getHostName(),targetHost.getPort()),
		new UsernamePasswordCredentials("user", "password")
	);
	HttpGet httpget = new HttpGet(url);
	// perform GET method
	HttpResponse response = httpclient.execute(targetHost, httpget);
	InputStream is = response.getEntity().getContent();
	...
	// here you can read data from input stream
}
catch (Exception exc)
{
	// handle exeption here
	exc.printStackTrace();
}
finally
{
	httpclient.getConnectionManager().shutdown();
}
```

Klient je reprezentován třídou `DefaultHttpClient` a před jeho použitím je nutné provést celou řadu konfigurací. Nejprve je nutné nastavit tomuto objektu přihlašovací údaje, které byly obdrženy od uživatele, dále je nutné specifikovat cílový server, port a protokol. V tomto případě se jedná o zabezpečený šifrovaný přenos protokolu *https*, kterému je přiřazen *port 443*. Dále je nutné vytvořit objekt, který bude reprezentovat *http metodu*, k dispozici jsou všechny standardní metody, jako *GET*, *POST* nebo *PUT*. Tomuto objektu je pak předána konkrétní adresa, na které se nacházejí požadovaná data. Následně je metoda provedena v kontextu vytvořeného a nakonfigurovaného klienta. Zvolená implementace klienta zajistí veškerou Http Digest komunikaci a vrátí objekt typu `HttpResponse`, ze kterého lze extrahovat požadovaná data. Nakonec je spojení se serverem uzavřeno. Komunikace se servery projektu Lissom probíhá na dvou úrovních, pomocí dvou různých asynchronních operací. Obě využívají výše zmíněného zabezpečeného mechanismu Http Digest. Sekvenční diagram na obrázku 13 znázorňuje zjednodušenou síťovou komunikaci se servery.



Obrázek 13: Sekvenční diagram komunikace třídy OnlineProjectView se servery

Diagram zachycuje dvě události spuštěné uživatelem – nejprve přihlášení, poté import projektu. Jak bylo popsáno výše, každá komunikace se serverem je implementována jako samostatné vlákno v tomto případě se jedná o třídy `LoginOperation` a `ImportProjectOperation`. Každá implementuje zmíněné rozhraní `IRunnableWithProgress`. Třída `LoginOperation` realizuje přihlášení k serveru a stažení souboru, který obsahuje XML popis struktury projektů. Po stažení stromu projektů dojde k jeho načtení pomocí knihovny `javax.xml`, kterou už jsem zmínil v kapitole 4.3, do datové struktury znázorněné na obrázku 12 a následně dojde k vizualizaci pomocí grafického uživatelského rozhraní. Následující XML kód ilustruje popis struktury projektů, pro zjednodušení jsou vynechány atributy obsahující popis projektů, nejsou uváděny url adresy a jsou zjednodušeny identifikátory projektů.

```
<?xml version="1.0" encoding="UTF-8"?>
<projects>
  <folder name="Codasip ASIPs">
    <folder name="CODIX">
      <folder name="codix">
        <folder name="applications">
          <project name="codix_bitcnt" id="codix_bitcnt.zip" url="..."/>
          <project name="codix_mpeg" id=" codix_mpeg.zip" url="..."/>
        </folder>
        <folder name="models">
          <project name="Hardware realization (cycle accurate)">
```

```
            <folder name="0.2.4">
              <project name="codix_ca" id=" codix_ca.zip" url="..."/>
          </folder>
        </folder>
        <folder name="Software development (instruction accurate)">
          <folder name="0.0.3">
            <project name="codix_ia" id=" codix_ia.zip" url="..."/>
          </folder>
        </folder>
      </folder>
    </folder>
  </folder>
</projects>
```

Po načtení a zobrazení dat může uživatel celým stromem procházet a importovat jednotlivé projekty. O import projektů se stará třída `ImportProjectOperation`. Ta nejprve (opět s Http Digest autentizací) stáhne požadovaný projekt zabalený ve formátu zip. Adresu umístění projektu obdrží v rámci zmíněného XML. Poté je stažený projekt nutné rozbalit a konečně importovat do pracovního prostoru prostředí, který se v Eclipse nazývá *Workspace* [12]. Pokud nastane v průběhu této operace chyba, je o tom uživatel informován prostřednictvím dialogového okna s popisem problému, který nastal.

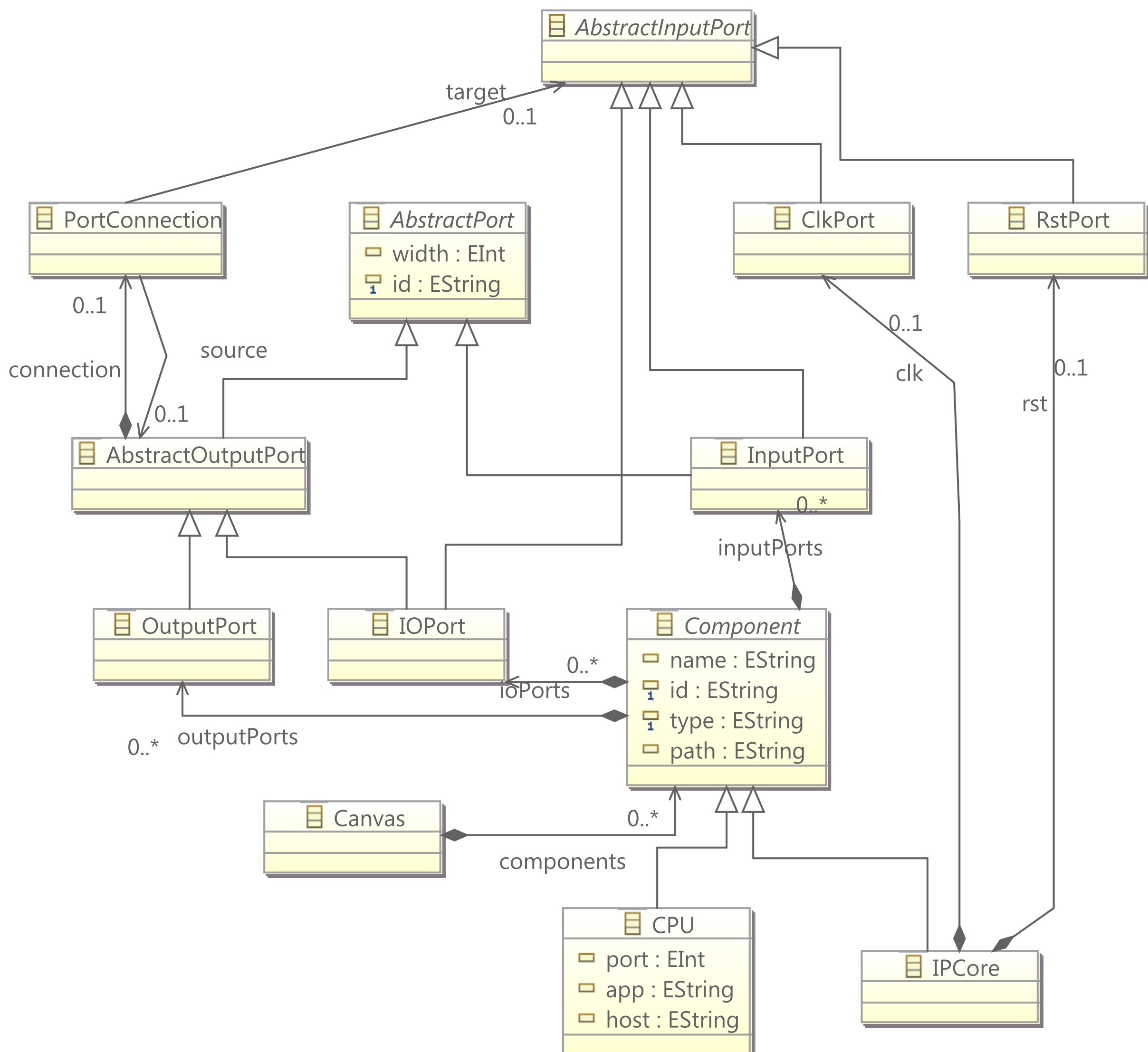
5.3 Implementace grafického nástroje

Grafický editor pro návrh víceprocesorových systémů na čipu je jádrem celé práce, nicméně pro své plnohodnotné uplatnění vyžaduje sadu nástrojů, které byly popsány v předchozích kapitolách. Editor je realizován jako zásuvný modul pro platformu Eclipse pomocí osvědčené kombinace frameworků GMF a EMF, které byly představeny v kapitole 4.4. Návrh editoru se odvíjel od celé řady požadavků, spousta další se objevila během testování prvních verzí a editor byl o ně obohacen, nebo se jedná o jeho možná budoucí rozšíření. Následující kapitoly popisují jednotlivé kroky, které vedly k výslednému editoru.

5.3.1 Definice metamodelu a generování kódu

Definice metamodelu je stěžejním krokem v implementaci nástrojů vytvořených pomocí frameworků GFM a EMF. Kvalitně navržená struktura metamodelu může usnadnit spoustu práce při budoucích dílčích modifikacích kódu. Metamodel je možné specifikovat pomocí několika způsobů – diagramem, kódem v jazyce Java se speciálními anotacemi nebo XML popisem. Já jsem zvolil metodu pomocí diagramu, která se mi osvědčila už v minulosti a je ze všech tří možností nejpřehlednější. Jedná se o diagram, který víceméně odpovídá diagramu tříd UML a je nazván zkratkou SoC (*System on a Chip*). Z provedené analýzy požadavků vyplývá, že v diagramu se budou nacházet dva typy objektů – *CPU* a *IP Core*, které bude možno propojovat pomocí portů. Obě tyto komponenty mají spoustu společných znaků, proto zde bylo vhodné využít dědičnost a spojit tyto společné vlastnosti do abstraktní třídy, ze

které jsou obě komponenty zděděny. Mezi tyto společné znaky patří atributy *jméno*, *identifikátor*, *typ* a *cesta* k souboru nebo projektu, který tato komponenta modeluje a reference na porty. Cesta k souboru nebo projektu se využívá při aktualizaci celého modelu, který musí vždy odpovídat aktuálnímu stavu zdrojů, které popisuje. Typ komponenty odpovídá názvu projektu nebo IP Core a je vyžadováno, aby jej uživatel nemohl měnit. Naopak jméno je pojmenování konkrétní instance komponenty v modelu a je na uživateli, aby je objektu přiřadil. Atributy objektu CPU *port* a *host* slouží ke specifikování stanice, na které se bude spouštět simulace tohoto procesoru, atribut *app* slouží ke specifikaci aplikačního projektu, který během simulace na procesoru poběží.

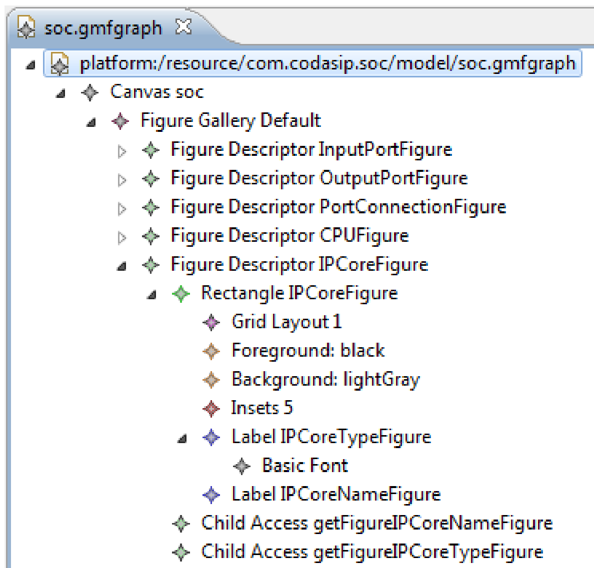


Obrázek 14: Diagram metamodelu

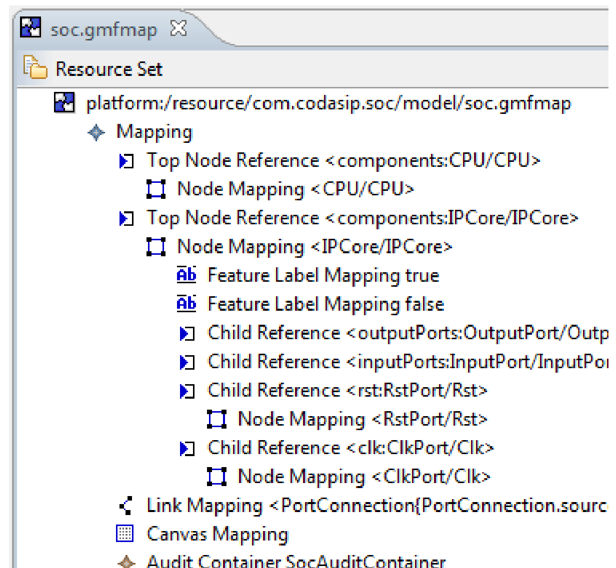
Dědičnosti jsem využil i u tříd reprezentujících porty, zde je graf dědičnosti poněkud složitější, protože výsledný model bude obsahovat čtyři typů portů – vstupní, výstupní, reset a clock. V průběhu vývoje se dále objevil požadavek na vstupně-výstupní port, který je kombinací obou základních portů, jak je vidět v diagramu. Vztah mezi porty a komponentami je podle terminologie diagramu tříd kompozicí, která popisuje skutečnost, že jeden objekt je obsažen ve druhém a nemůže beze svého rodiče existovat. Do hry zde dále vstupuje spojení portů. To je v ECore metamodelu reprezentováno

samostatnou třídou, která obsahuje dvě reference – jednu na zdrojový, druhou na cílový objekt propojení. Obě reference musí mít kardinalitu 0 až 1. Důležitý je vztah propojení a jeho rodič, jedná se opět o kompozici a udává směr, ve kterém je spojení vedeno. V tomto konkrétním případě je spojení reprezentováno třídou `PortConnection`, která je součástí třídy reprezentující výstupní port – `AbstractOutputPort`. Ten je zároveň referencí na zdrojovou třídu spojení. Reference na cíl je nastavena na třídu `AbstractInputPort`. Společnou vlastností všech portů, kromě speciálních případů, kterými jsou clock a reset, je jejich bitová šířka a identifikátor. Identifikátory portů a komponent byly do modelu přidány až v průběhu implementace a jejich využití je zejména při generování výstupního XML souboru, více se tomuto problému věnuje kapitola 5.3.4. Metamodel musí specifikovat kořenovou třídu, která reprezentuje plátno budoucího diagramu, ta je nazvána `Canvas`. Oba typy komponent (CPU, IP Core) jsou s touto kořenovou třídou v relaci kompozice. Na obrázku 14 je celý diagram tříd metamodelu pro editor víceprocesorových systémů.

Po vytvoření diagramu metamodelu následuje vygenerování kódu metamodelu. Framework EMF vygeneruje všechny třídy a jejich atributy specifikované v metamodelu. Ke každému atributu navíc vygeneruje metody *get* a *set*. Všechny třídy modelu mají jednoho společného předka – třídu `EObject`, která obsahuje několik užitečných metod. Jedná se zejména o metodu `eContainer`, která vrátí rodiče daného objektu, opět objekt typu `EObject`, a metodu `eContents`, která vrátí kolekci objektů, které jsou naopak potomky daného objektu. Tento přístup je velice podobný konceptu DOM a je díky němu možné se pohybovat napříč modelem nezávisle na konkrétních typech objektů. Další vlastností třídy `EObject` je kolekce `eAdapters`, jedná se o implementaci návrhového vzoru *Observer* [23]. Do této kolekce je možné libovolně přidávat objekty, tzv. *adaptéry*, které jsou informovány o změnách v objektu modelu. Tomu odpovídají i zmíněné metody set, které zodpovídají za nastavení nové hodnoty atributu, v každé takové metodě EMF vygeneruje také volání `eNotify`, které upozorní všechny zaregistrované adaptéry na změnu v modelu. Této skutečnosti využívá vrstva GMF, která zajišťuje grafickou nástavbu nad EMF v podobě diagramového editoru. Každá změna, která se stane v modelu je promítnuta také do diagramu.



Obrázek 15: Ukázka souboru soc.gmfgraph



Obrázek 16: Ukázka souboru soc.gmfmap

Property	Value
Descriptor	Figure Descriptor IPCoreFigure
Fill	true
Line Kind	LINE_SOLID
Line Width	1
Name	IPCoreFigure
Outline	false
Xor Fill	false
Xor Outline	false

Obrázek 17: Vlastnosti uzlu Rectangle IPCoreFigure

Property	Value
Domain meta information	
Features to display	Component.type:EString
Features to edit	
Misc	
Diagram Label	Diagram Label IPCoreType
Read Only	true
Visual representation	
Edit Method	MESSAGE_FORMAT
Editor Pattern	
Edit Pattern	
View Method	MESSAGE_FORMAT
View Pattern	

Obrázek 18: Vlastnosti uzlu Feature label mapping

Po vygenerování modelu následují kroky, které vedou již k vytvoření zásuvného modulu s diagramovým nástrojem. Nyní je nutné definovat tři soubory, které popisují, jakým způsobem bude budoucí editora vypadat, jak budou vypadat jednotlivé prvky diagramu a jak se budou mapovat na model diagramu. Jedná se o soubory *gmftool*, *gmfgraph* a *gmfmap*. Všechny tři soubory obsahují text v jazyce XML a často bývají poměrně rozsáhlé, obzvláště gmfgraph. GMF nicméně nabízí editory, které umožňuje tyto soubory editovat pomocí uživatelsky přijatelnějších stromových reprezentací. První ze jmenovaných souborů obsahuje pouze seznam nástrojů, které budou v editoru k dispozici. Jedná se o nástroje, které budou staticky přítomny v nástrojové paletě. Pro tento konkrétní editor je ale požadováno, aby jednotlivé komponenty v souborovém systému prostředí (Workspace) měly zastoupení na nástrojové liště diagramu, tudíž tyto nástroje musí být do palety umisťovány dynamicky. Z těch statických je v souboru gmftool zastoupen pouze nástroj pro vytváření spojení mezi porty. Druhým souborem, který je potřeba specifikovat pro vygenerování kódu grafického editoru je gfmgraph. Jedná se o specifikaci jednotlivých grafických objektů a jejich vlastností. Je třeba definovat to, jak budou reprezentovány jednotlivé uzly, popisky a spojení. V definici uzlů je možné specifikovat hierarchii všech objektů, které do uzlu náleží, jejich rozmístění v uzlu, barvu a řez

písma, pozadí atd. Na obrázku 15 je ukázka ze souboru soc.gfmgraph s rozbaleným stromem uzlu, který specifikuje vzhled budoucích IP Core objektů. Jednotlivé vlastnosti prvků v této struktuře je možné editovat pomocí pohledu *Properties* (viz obrázek 17). Z uvedené ukázky je patrné nastavení barvy pozadí, písma a čar v uzlu. Dále je zde specifikována politika rozložení potomků – Grid Layout, která zajišťuje rozložení do tabulky, v tomto případě s jedním sloupcem. Poslední položkou jsou popisky typu a jména komponenty.

Na obrázku 16 je ukázka ze souboru gmfmap, jedná se v podstatě o nejdůležitější soubor z celé trojice zmíněných souborů. Tento soubor obsahuje mapování jednotlivých objektů modelu popsaných v metamodelu ECore na jejich grafické reprezentace popsané v souboru gmfgraph a ty následně spojuje s nástroji ze souboru gmftool. Pomocí uzlů typu *Top Node Reference* jsou specifikovány objekty, které se budou vkládat přímo na plátno diagramu. Atributy (typ, jméno) a potomci (porty) těchto objektů jsou popsány uzly *Feature Label Mapping* a *Child Reference*. Na obrázku 18 je vidět mapování atributu typ na konkrétní grafickou reprezentaci, navíc je zde možno specifikovat, zda bude uživatel moci tento atribut editovat. Po vytvoření všech tří souborů je možné vygenerovat soubor gmfgen, který spojuje všechny tři zmíněné soubory a lze v něm nastavit některé další požadované vlastnosti generovaného kódu. Následně je možno konečně vygenerovat kód zásuvného modulu. Ten je ale potřeba upravit podle požadavků na výsledný editor, tomuto tématu se věnuje následující kapitola.

5.3.2 Úpravy kódu diagramu

Po vygenerování kódu je k dispozici spustitelná verze editoru víceprocesorových systémů, nicméně nyní je potřeba upravit stávající kód tak, aby byly splněny zejména následující požadavky:

- Paleta nástrojů se automaticky aktualizuje podle dostupných zdrojů.
- Načítání a aktualizace vložených komponent podle jejich vzorů.
- Umožnit vkládání pouze přeložených projektů.
- Menší grafické úpravy, které není možné specifikovat v konfiguračních XML souborech.

V nově vygenerovaném kódu je v paletě nástrojů v editoru k dispozici pouze nástroj pro vytváření spojení, nyní je potřeba zabezpečit, aby paleta dynamicky reagovala na změny v souborovém systému vývojového prostředí (dále Workspace). K tomu je potřeba sledovat změny ve Workspace a reagovat na ně. Eclipse nabízí tzv. *Resources API* (application programming interface), které zajišťuje přístup ke všem zdrojům ve Workspace. Eclipse Resources API je abstrakcí nad reálným fyzickým souborovým systémem, umožňuje přistupovat k datům z pohledu Eclipse, tedy skrze strom projektů a jejich obsah. K modelu zdrojů Eclipse je možné přistoupit pomocí statické metody `ResourcesPlugin#getWorkspace`, ta vrátí referenci na objekt typu `IWorkspace`, který odpovídá aktuálnímu Workspace. Přes tento objekt je možné dále přistupovat ke všem projektům a

jejich souborům, upravovat je, mazat a vytvářet nové. To by ale pro požadované reakce na aktuální změny ve Workspace nestačilo, proto rozhraní `IWorkspace` nabízí systém založený na návrhovém vzoru *Observer* [23]. Pomocí metody `addResourceChangeListener` je možné do modelu přidávat a odebírat tzv. *listenery* (posluchače), objekty implementující rozhraní `IResourceChangeListener` s jedinou metodou `resourceChanged` [12] [26]. Prakticky se jedná o kolekci těchto objektů umístěnou v objektu představujícím Workspace. Při jakékoliv změně ve Workspace jsou o této změně informováni všichni registrovaní posluchači, tedy všichni posluchači v této kolekci.

Pro účely projektu Lissom je nad Eclipse Resources API vytvořena další vrstva abstrakce, která nabízí možnosti spjaté se specifickými vlastnostmi Lissom projektů. Tato vrstva nabízí stejný mechanismus pro reakce na změny, který byl popsán výše. Zásuvný modul diagramového editoru je reprezentován třídou `SocDiagramEditorPlugin`, tato třída zajišťuje životní cyklus zásuvného modulu skrze několik obslužných metod, zejména metody `start` a `stop`, které jsou volány při spuštění a ukončení činnosti zásuvného modulu. V těchto místech je vhodné zaregistrovat posluchače, kteří se budou starat o reakce na změny ve Workspace. Před ukončením činnosti je nutné zaregistrované posluchače odebrat pomocí metody `removeResourceChangeListener`. V rámci této práce je nutné sledovat změny ve dvou typech objektů – změny v projektech, konkrétně *Codasip hardware project,* a obecně všechny změny ve Workspace, které se týkají souborů s popisem IP Core. K těmto účelům jsem implementoval dva posluchače, jeden se stará pouze o přidávání hardware projektů do nástrojové lišty, druhý obstarává totéž s nástroji pro instance IP Core, a navíc aplikuje změny v modelu zdrojů na konkrétní instance v diagramu.

Paletu nástrojů je možné pro přehlednost rozdělit do skupin, v této prací jsem vytvořil tři skupiny nástrojů – první obsahuje nástroj pro spojování portů, druhá je určena pro nástroje ke vkládání CPU (hardware projekty) a třetí pro nástroje zodpovědné za tvorbu IP Core. Při každé zaznamenané změně ve Workspace je nutno projít všechny hardware projekty, resp. celý Workspace (pro účely vyhledání všech IP Core souborů), a danou skupinu nástrojů aktualizovat. Následující kód ilustruje rekurzivní prohledávání Workspace, funkce vrací mapu, ve které jsou mapovány cesty ke všem nalezeným souborům IP Core na jména těchto komponent. Algoritmus je zavolán na kořenový prvek Workspace a projde všechny jeho projekty a rekurzivně se do nich zanořuje. Pokud narazí na soubor s příponou „xml“, pokusí se jej načíst jako XML dokument a vyhledat v něm element s názvem „component“. Pokud je úspěšný, extrahuje z XML struktury také element s hodnotou názvu komponenty a společně s cestou uloží do mapy. Po vytvoření mapy se všechny její prvky projdou a pro jednotlivé záznamy jsou vytvořeny nástroje v paletě nástrojů. Jméno se použije jako zobrazovaný název nástroje, cesta je uložena do atributů nástroje a použita při vytváření konkrétní instance komponenty. Mechanismus pro aktualizaci CPU nástrojů je poněkud jednodušší, Eclipse nedovoluje umísťovat projekty rekurzivně do sebe, takže stačí pouze projít seznam všech projektů ve Workspace. Model zdrojů Lissom navíc nabízí metodu, která vrátí seznam všech hardwarových projektů ve Worskpace.

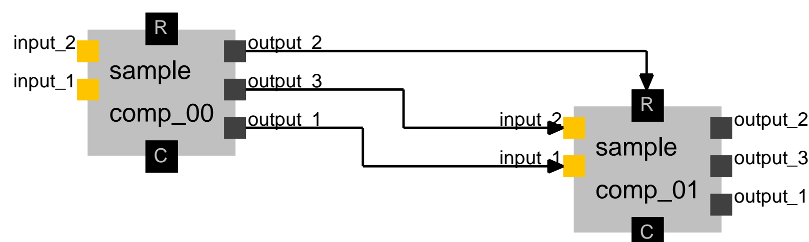
```java
private static Map<String, String> findSingleCompomentFiles(IResource parent)
{
  Map<String, String> map = new HashMap<String, String>();
  if (parent instanceof IFile)
  {
    IFile file = (IFile) parent;
    // we have to check if file is xml
    if (file.getFileExtension() != null && file.getFileExtension().equals("xml"))
    {
      try
      {
        // now look if xml contains top node called component
        Document doc = DocumentBuilderFactory.newInstance().
                       newDocumentBuilder().parse(file.getContents());
        NodeList nodelist = doc.getElementsByTagName(component");
        if (nodelist.getLength() == 1 && nodelist.item(0) instanceof Element)
        {
          Element element = (Element) nodelist.item(0);
          String name = element.getElementsByTagName("name").
                        item(0).getTextContent();
          // final put this component file into result map
          map.put(file.getFullPath().toPortableString(), name);
        }
      }
      catch (Exception e)
      {
        return map;
      }
    }
  }
  else if (parent instanceof IContainer)
  {
    // its project or folder, we have to recursively look into it
    try
    {
      IResource members[] = ((IContainer) parent).members();
      for (IResource res : members)
        map.putAll(findSingleCompomentFiles(res));
    }
    catch (CoreException e1)
    {
      return map;
    }
  }
  return map;
}
```

V metamodelu pro tento diagramový nástroj jsem specifikoval zejména dva textové atributy komponent – *název* a *cesta*. Tyto atributy jsou inicializovány při vložení komponenty na plátno. Při vytváření si tyto komponenty vyzvednou z atributu nástroje, pomocí kterého jsou vytvářeny, cestu k souboru nebo projektu, který reprezentují. Následně se inicializují podle svého vzoru. V případě vkládání CPU je potřeba, aby byl projekt, který reprezentují, přeložen do interního kódu Lissom. Pokud tomu tak není, je vývojář dotázán, zda chce projekt přeložit. Nyní je možné projekt přidat na plátno. To, zda je projekt přeložen, lze detekovat pomocí metod, které nabízí model zdrojů Lissom, tak jak je tomu v následujícím kódu. Inicializace v metodě `initialize` načte model hardwarového

projektu, ze kterého je možné získat informace o modelu, v tomto případě je klíčový seznam portů, jejich šířek a identifikátorů, které jsou vytvořeny a vloženy do nového CPU. U vytváření instancí IP Core je postup identický, pouze s rozdílem, že model není načítán pomocí metod modelu Lissom, ale s pomocí knihoven pro zpracování a načítání XML.

```
IProject proj = ResourcesPlugin.getWorkspace().getRoot().getProject("name");
IHardwareProject hwproject = CorePluginAPI.getModel().getHardwareProject(proj);
ICodasipFile output = hwproject.getModelOutput();
if (!output.getResource().exists())
{
	// ask to build project
	...
}
cpu.initialize();
```

Posledními úpravami v kódu diagramu jsou lehké kosmetické změny v zobrazování komponent na plátně, respektive v umístění portů v komponentách. Porty jsou reprezentovány jednoduchým čtvercem o velikosti strany asi 10 pixelů a jednotlivé typy portů jsou odlišeny barevně. GMF umožňuje specifikovat potomka uzlu jako tzv. *fixed child* a zvolit jednu ze stran rodiče (*WEST*, *EAST*, *NORTH*, *SOUTH*), na kterou budou umístěni. GMF pak podle svého algoritmu umísťuje prvky po okraji rodičovského objektu. Tento algoritmus je ale nevhodný pro umísťování portů, proto jsem musel některé metody tohoto algoritmu upravit tak, aby výsledek odpovídal požadavkům a byl vhodný pro reprezentaci komponentního systému, resp. abstrakce UML diagramu komponent. Vstupní porty jsou oranžové a jsou umístěny na západní straně rodičovské komponenty, výstupní jsou tmavě šedé a jsou umístěny na východě, na severu a jihu jsou černé porty pro vstupy clock a reset označené písmeny *C* a *R*. Navíc jsou u portů zobrazeny jejich identifikátory. Výsledné provedení je znázorněno na obrázku 19.



Obrázek 19: Ukázka grafické reprezentace portů

5.3.3 Integrace do vývojového prostředí

Upravený nástroj teď již docela odpovídá požadavkům, je ale nutné provést ještě několik kroků, které povedou k efektivnější práci s nástrojem. Grafický editor v jeho současné podobě nereflektuje změny ve Workspace, o kterých jsou informováni posluchači, které jsem zmínil v předchozí kapitole. Je tedy nutné implementovat reakce na změny přímo v souborech. Pokud například vývojář provede změny ve zdrojovém kódu hardwarového projektu, tedy v souboru s kódem v jazyce CodAL, a projekt následně přeloží, musí se provedené změny odrazit i na plátně diagramu s víceprocesorovým

systémem, ve kterém je tento hardwarový projekt umístěn. Stejná situace by měla nastat také při změnách v souboru popisujícím některou z IP Core komponent použitých v diagramu. Dále je nutné reagovat na mazání, přejmenovávání a přesouvání souborů a projektů. Eclipse Resources API chápe přejmenovávání jako přesouvání. Pokud například dojde ke změně názvu souboru, je toto chápáno jako přesun tohoto souboru do umístění, které je dáno původní cestou k rodičovskému adresáři a novým názvem souboru. Posluchač implementující rozhraní `IResourceChangeListener` je o změnách informován prostřednictvím metody `resourceChange`, která má jeden parametr – objekt typu `IResourceChangeEvent`, ve kterém jsou ukryty veškeré informace o změně, která v prostředí nastala. Díky tomuto objektu je možné projít všechny komponenty umístěné na plátně diagramu a porovnat, zda cesta k souboru resp. projektu, který reprezentují, odpovídá právě změněnému souboru resp. projektu, dále zdroji. Pokud ano je nutné nejprve zkontrolovat, zda byl zdroj odstraněn, přesunut (přejmenován) nebo došlo pouze ke změně uvnitř tohoto zdroje. Ad hoc řešení při odstranění zdroje by bylo smazat jeho instanci, také z plátna diagramu, nicméně se může jednat o poměrně klíčový prvek v celém modelovaném systému, který na sebe váže spoustu jiných komponent. Proto je vhodnějším řešením vývojáři pouze oznámit, že zdroj, který daná komponenta modeluje, byl odstraněn. Při přejmenování dojde pouze k aktualizaci atributu cesta v postižené komponentě. Pokud došlo ke změnám uvnitř zdroje, je nutné tuto komponentu znovu inicializovat pomocí metody `initialize`. Ta je využívána jednak při vytváření komponenty, ale i v případě, že daná komponenta již existuje, ale registrovaný listener byl informován o nějaké změně, která v ní nastala. Pro tento případ jsem navrhl algoritmus, který umí recyklovat stávající porty, takže porty se stejným identifikátorem jsou zachovány a s nimi i spojení, která v diagramu existují. Porty, které již reprezentovaná komponenta neobsahuje, jsou odstraněny a naopak jsou přidány nové porty, které v komponentě přibyly.

Druhá úroveň, na které bude editor integrován s vývojovým prostředím, je provázání se zdroji, které jednotlivé komponenty modelují. V praxi půjde o to, že pokud vývojář poklepe na některou z komponent na plátně diagramu, otevře se mu její zdrojový kód. V případě IP Core se otevře XML editor s IP-XACT popisem komponenty, v případě CPU se otevře zdrojový soubor s popisem procesoru v jazyce CodAL. Na jednotlivé objekty, které jsou zobrazovány v GMF diagramech, se instalují tzv. *politiky*, které spravují jednotlivé druhy chování objektu, tzv. role, kterých je celá řada. Jedná se o implementaci návrhového vzoru *Strategy* [23], tedy situace, kdy je potřeba aplikovat různé algoritmy tak, aby byly vzájemně zaměnitelné a uplatnitelné i v době běhu programu. Na jednu roli objektu je možno nainstalovat i více politik. Je tak možné ovlivnit například to, jakým způsobem se bude zacházet s potomky objektu, jak bude s objektem nakládáno při pokusu o smazání, ale také je možné přiřadit politiku, která se aplikuje při dvojkliku na objekt. Následující kód ilustruje instalaci takové politiky na objekt, který v diagramu reprezentuje CPU. Při poklepání na objekt (role *OPEN_ROLE*) dojde po nainstalování této politiky k jejímu uplatnění. To znamená, že je zavolána metoda `getCommand`, která vytvoří tzv. *příkaz* (viz návrhový vzor *Command* [23]). Ten je

zodpovědný za otevření nového editoru, konkrétně se jedná o třídu OpenEditorCommand, které je předána cesta k souboru, který má být otevřen. Prostředí následně samo vybere vhodný editor k otevření zvoleného souboru.

```
installEditPolicy(EditPolicyRoles.OPEN_ROLE, new OpenEditorEditPolicy(new
IOpenEditorCommandProvider()
{
  @Override
  public Command getCommand()
  {
    EObject element = getNotationView().getElement();
    if (element instanceof CPU)
    {
      CPU cpu = (CPU) element;
      // retrieve Eclipse project from cpu object by its name
      IProject project = ResourcesPlugin.getWorkspace().getRoot().
                         getProject(cpu.getType());
      // retrieve Codasip HW project from Eclipse project
      IHardwareProject hwproj = CorePluginAPI.getModel().
                                getHardwareProject(project);
      // retrieve main CodAl file and open it
      IResource resource = hwproj.getMainModelFile().getResource();
      if (resource != null && resource.exists())
        return new ICommandProxy(
               new OpenEditorCommand(resource.getFullPath().toString()));
    }
    return null;
  }
}));
```

5.3.4 Generování výstupů

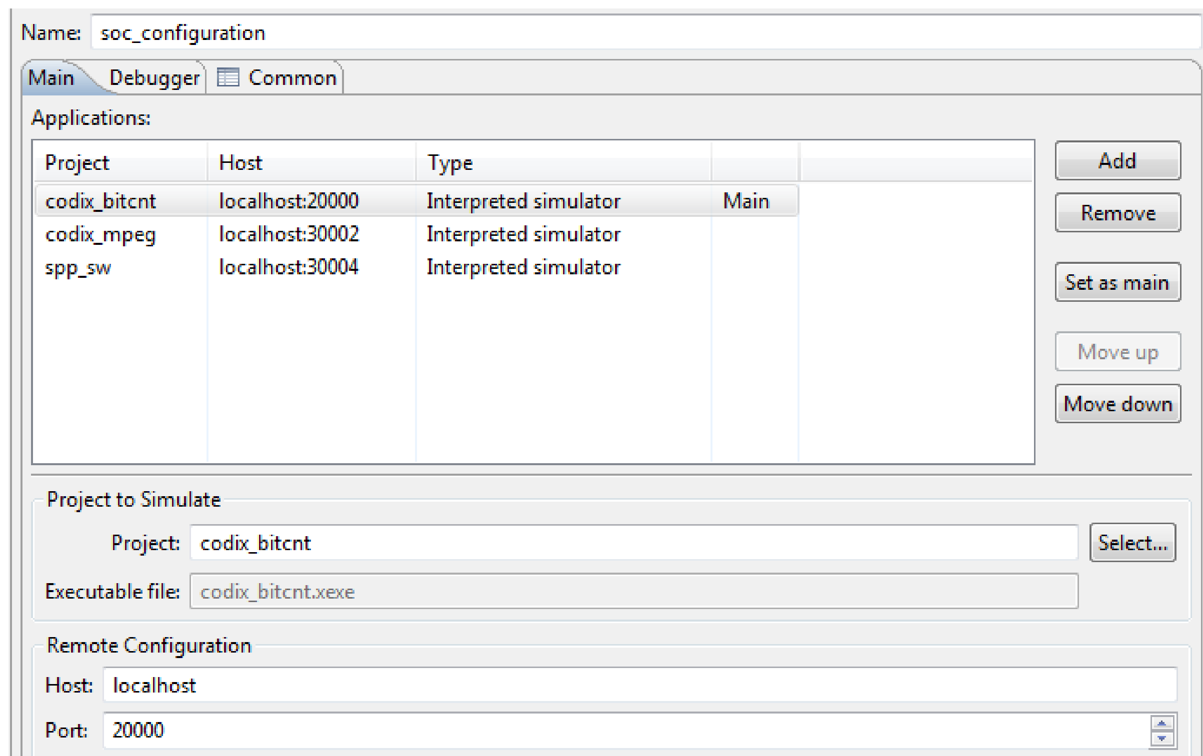
V momentě, kdy vývojář dokončí práci na modelování víceprocesorového systému, musí mít k dispozici nástroje jak svou práci zhodnotit a posunout se ve vývoji dále, tedy k simulování celého systému. Celý model obsahuje několik komponent, které by měly být nakonfigurovány k simulaci, nyní je potřebovat vytvořit konfiguraci pro debugovací perspektivu (viz kapitola 4.2). K tomuto účelu je již v Codasip Studio vytvořeno aplikační rozhraní, které umí takovou perspektivu vytvořit. Následující kód ilustruje použití tohoto rozhraní.

```
RemoteConfiguration rc = new RemoteConfiguration();
String output = null;
// find every cpu object at the canvas
for(EObject eo : canvas.eContents())
{
	if(eo instanceof CPU)
	{
		CPU cpu = (CPU) eo;
		// create configration and fill it with cpu properties
		SimulatorConfig simconf = new SimulatorConfig();
		simconf.setApplication(cpu.getApp());
		simconf.setHost(cpu.getHost());
		simconf.setPort(cpu.getPort());
		simconf.setProjectName(cpu.getType());
		simconf.setSoftwareProject(cpu.getApp());
```

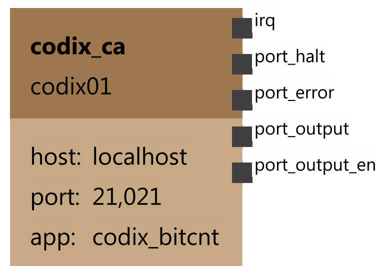
```
            simconf.setRelativeFilePath(cpu.getPath());
            rc.getSimulatorsConfig().add(simconf);
      }
}
// finaly generate output text representation
output = rc.serialize();
```

Postupně se projdou všechny procesory v modelu a pro každý z nich se vytvoří objekt `SimulatorConfig`, kterému se nastaví všechny potřebné atributy, a tento objekt je vložen do struktury představující celou konfiguraci. Nakonec je objekt serializován. Výstupem je v podstatě strukturovaný popis konfigurací jednotlivých CPU v modelu. Vývojář si může perspektivu vygenerovat přes kontextové menu plátna diagramu. Výsledná konfigurace je uložena do textového souboru a odpovídá následujícímu XML kódu. Odpovídající grafická reprezentace procesoru na plátně je na obrázku 21. Jedná se o procesor typu *codix_ca*, simulace poběží na stanici *localhost* (IP adresa 127.0.0.1) na portu 20000, simulovat se bude softwarový projekt s *codix_bitcnt*. Na 20 obrázku je stejná konfigurace v grafické podobě v uživatelském rozhraní Codasip Studio.

```
<SIMULATION>
   <SIMCNF>
      <PROJECT>codix_ca</PROJECT>
      <HOST>localhost</HOST>
      <PORT>20000</PORT>
      <APP>codix_bitcnt.xexe</APP>
      <FREQ>0</FREQ>
      <NONMW-OPTFILE_PATH>/codix_ca</NONMW-OPTFILE_PATH>
      <NONMW-IS_MAIN>false</NONMW-IS_MAIN>
      <NONMW-SW_PROJECT>codix_bitcnt</NONMW-SW_PROJECT>
   </SIMCNF>
</SIMULATION>
```



Obrázek 20: Ukázka výsledné konfigurace v grafickém rozhraní Codasip Studio



Obrázek 21: Grafická reprezentace CPU

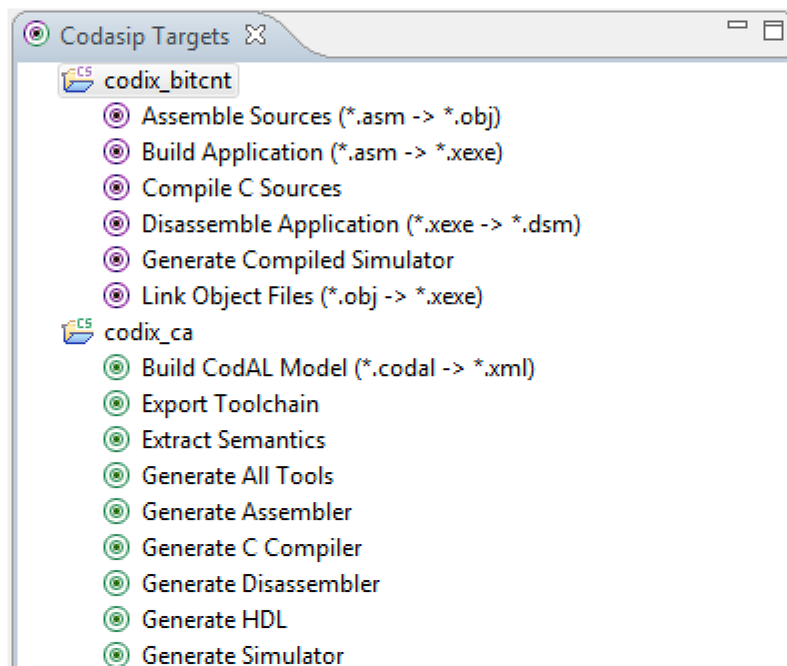
Druhým výstupem je popis celého modelu v jazyce XML. Pro tento účel jsem navrhl strukturu, díky které je možné celý model uchovat a následně použít při dalším zpracování (například export do HDL). V budoucnosti bude tato struktura nahrazena popisem ze standardu IP-XACT. Ke generování XML popisu jsem opět využil již dříve zmíněné knihovny pro práci s XML za použití DOM rozhraní (balíčky `javax.xml` a `org.w3c.dom`). V tomto bodě je potřeba vyřešit, jakým způsobem budou reprezentována jednotlivá propojení portů. Z tohoto důvodu jsem do metamodelu přidal textový atribut identifikátor komponent, který je v celém modelu jedinečný, protože je automaticky generován při vytvoření komponenty a uživatel nemá možnost jej změnit. Nyní je možné adresovat konkrétní port v komponentě, pomocí dvojice identifikátorů ve tvaru *port_id@komponenta_id*. Tímto jedinečným identifikátorem je označen vstupní port (včetně clock a reset), kam vede spojení z daného výstupního portu. V následujícím kódu je fragment vygenerovaného XML popisu modelu, z kterého je zjevné, jak probíhá párování spojených portů. Generování tohoto XML popisu je možno spustit opět přes kontextové menu plátna.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<soc>
  <name>soc</name>
    <cpu>
    <type>codix_ca</type>
    <name/>
    <id>_D_Yd4KqLEeKS6f_SLm5pZg</id>
    <ports>
      <oport>
<!–
     this output port is connected to a port "input_0" at component with id
     "_lo-bMKn9EeKznI0mroyA5A"
-->
        <id>port_output</id>
        <target>input_2@_lo-bMKn9EeKznI0mroyA5A</target>
        <width>16</width>
      </oport>
    </ports>
  </cpu>
  <ipcore>
    <type>sample</type>
    <name>comp_00</name>
    <id>_lo-bMKn9EeKznI0mroyA5A</id>
    <ports>
      <iport>
        <id>input_0</id>
        <width>16</width>
```

```
      </iport>
    </ports>
  </ipcore>
</soc>
```

5.4 SoC projekty

Persistence dat z diagramových nástrojů GMF je zajištěna pomocí dvou souborů, protože je nutné uchovat jednak samotná data modelu a dále pak informace o umístění jednotlivých objektů v diagramu, jejich rozměrech a spojeních. Oba soubory jsou textové a obsahují data zapsaná v jazyce XML. GMF vygeneruje také jejich přípony, v SoC diagramu jsou to *soc* a *soc_diagram*. V aplikacích postavených na platformě Eclipse mají jednotlivé projekty tzv. *nature*, což je informace o tom, k čemu projekt slouží, jaká data obsahuje, popisuje životní cyklus projektu apod. V modelu zdrojů Lissom lze vytvořit dva typy projektů – *hardware* a *software project*. SoC diagram představuje propojení několika projektů z modelu zdrojů Lissom, ať už se jedná o popis hardware nebo software. Model SoC ovšem z logických důvodů nepatří ani do jednoho z uvedených typů projektu. Proto byl ke stávajícím dvěma typům projektů přidán třetí, který je pojmenován SoC. Jedná se tedy o typ projektu, který zastřešuje více jiných projektů a spojuje je do jednoho víceprocesorového systému. Simulovat činnost projektů lze pouze s projekty, které jsou přeloženy do interní XML reprezentace, k tomuto účelu je v Codasip Studio k dispozici tzv. *Target view* (viz obrázek 22), tedy pohled, který pro jednotlivé projekty ve Workspace nabízí sadu cílů – *targetů*, které lze na projekty aplikovat (například překlad do interní XML reprezentace). SoC projekty mohou zastřešovat více projektů, z nichž všechny musí být pro úspěšnou simulaci činnosti přeloženy. Proto je jedním z budoucích rozšíření implementace hromadných targetů SoC projektu (viz kapitola 6.1).

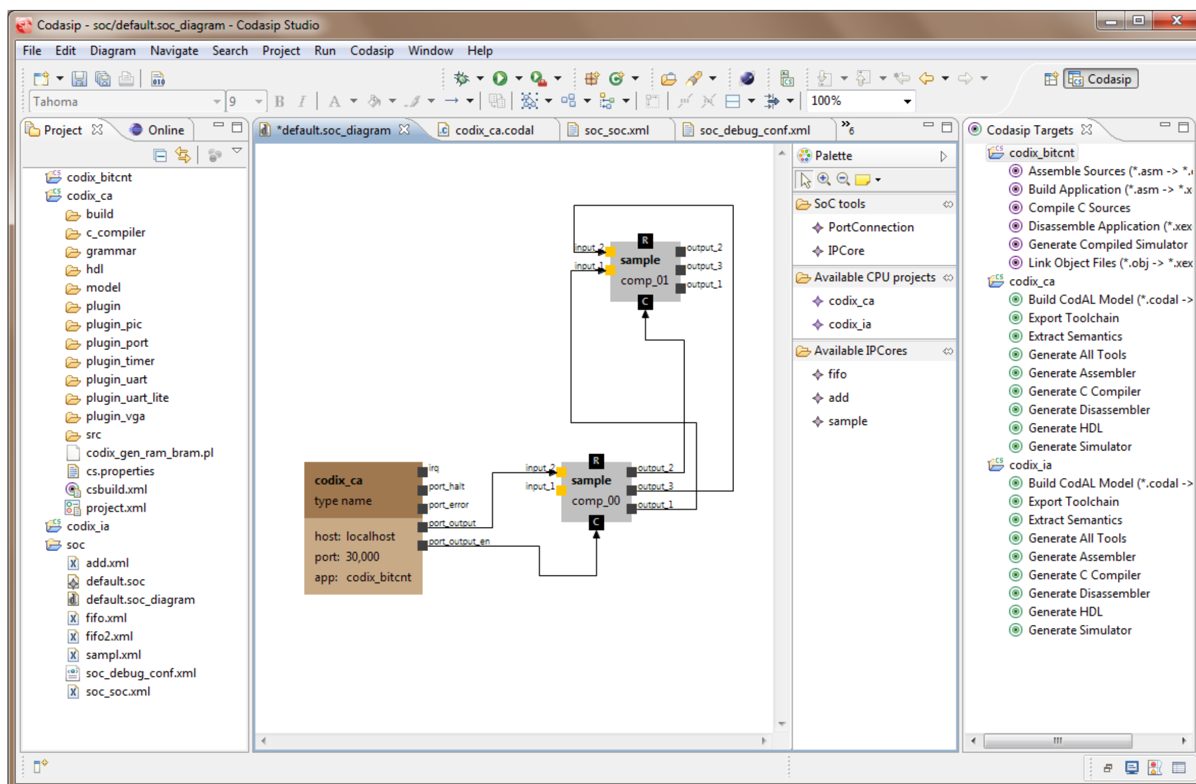


Obrázek 22: Ukázka Codasip Target view

6 Výsledné nástroje

Výsledkem mé diplomové práce je sada nástrojů pro modelování a vývoj víceprocesorových systémů plně integrovaná do prostředí Codasip Studio. Jádrem celého systému je nástroj pro grafické modelování víceprocesorových systémů na čipu v podobě diagramů. Na plátno diagramu je možné umísťovat instance hardwarových komponent (CPU a IP Core), jejichž popis se nachází v souborovém systému vývojového prostředí Eclipse, a vzájemně je propojovat skrze vstupní a výstupní porty. Jednotlivé komponenty jsou systémem obecně chápány jako černá skříňka. Důležitý je standardizovaný popis jejich rozhraní a na samotné vlastnosti komponenty nejsou kladeny výrazné nároky. Komponenty může vývojář získat několika způsoby:

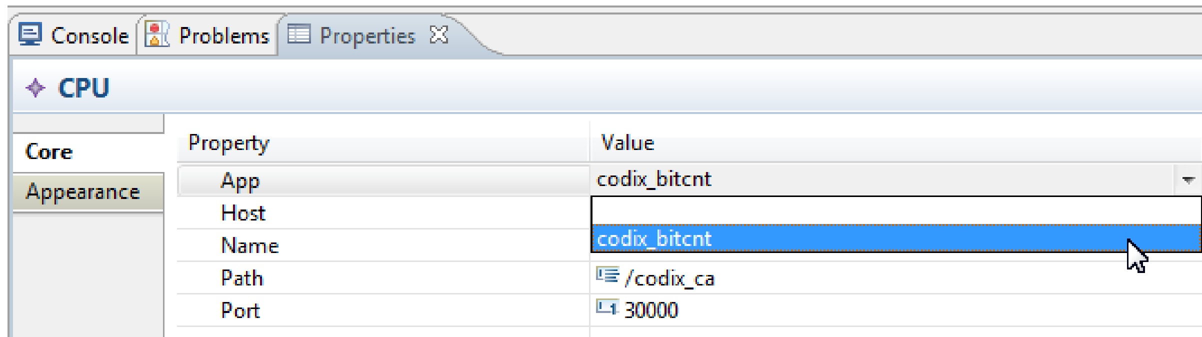
- Vytvořit model CPU nebo IP Core na zelené louce.
- Využít CPU Wizard k vygenerování kostry CPU v jazyce CodAl (včetně grafické reprezentace).
- Využít Single Compoment Wizard pro nakonfigurování a vygenerování IP Core v jazyce IP-XACT.
- Stáhnout si hotové řešení pomocí integrovaného prohlížeče online projektů z databáze projektu Lissom.



Obrázek 23: Výsledný nástroj

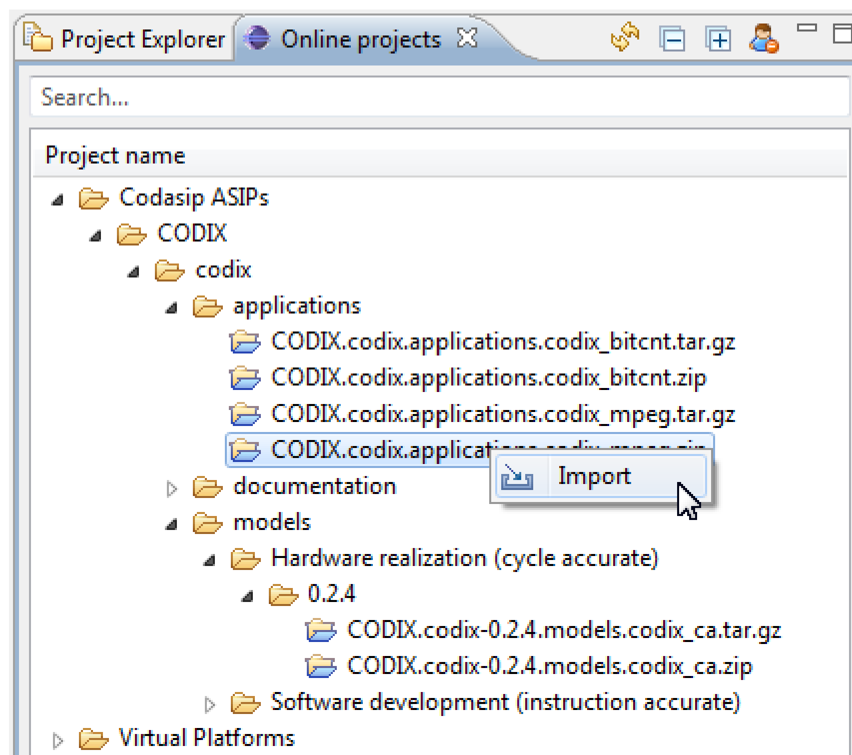
Na obrázku 23 je zobrazen celý výsledný nástroj. V okně aplikace je patrná nástrojová lišta, jejíž obsah se dynamicky mění podle toho, jaké komponenty jsou dostupné. V pravé části se nachází

zmíněný pohled Target View, kde je možno provádět vybrané akce nad projekty ve Workspace. Konfiguraci jednotlivých komponent je možné měnit přímo v editoru, nebo za pomocí pohledu vlastnosti. Na obrázku 24 je znázorněna konfigurace CPU, uživatel může procesoru přiřadit softwarový projekt z nabídky dostupných projektů ve Workspace.



Obrázek 24: Vlastnosti procesoru

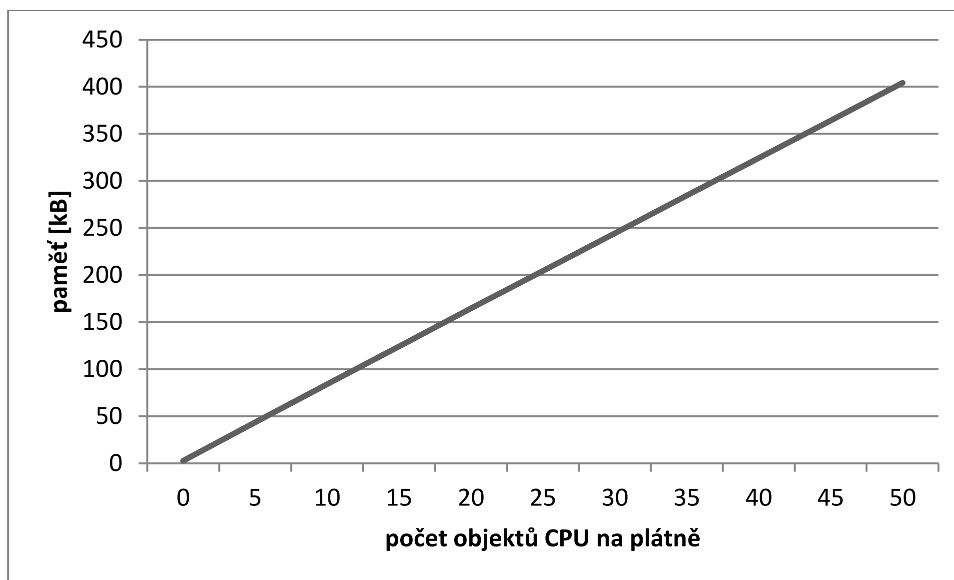
Projekty, které jsou dostupné online, může uživatel stahovat po zadání svých přihlašovacích údajů. Poté se mu zobrazí nabídka všech dostupných projektů, ke kterým má oprávnění (obrázek 25). V dostupných projektech je možné vyhledávat pomocí textového pole *Search*. Zvolený projekt je možné importovat skrze kontextové menu nebo dvojklikem. Pokud systém detekuje, že se ve Workspace již nachází projekt s daným jménem, je uživatel dotázán, zda jej chce přepsat. Ve stažených projektech může uživatel provádět libovolné změny a následně pak tyto projekty používat v modelu víceprocesorových systémů.



Obrázek 25: Online projekty

Výsledný model je možné exportovat skrze kontextové menu jako simulační konfiguraci nebo jako popis v jazyce XML. Vygenerování simulační konfigurace znamená pro uživatele podstatné zrychlení práce, veškerá nastavení jsou vyplněna automaticky a uživatel už je nemusí vyplňovat pro každý projekt zvláště. Diagramy frameworku GMF jsou navrženy tak, aby byly použitelné (rychlá odezva na uživatelské vstupy, stabilita, spolehlivost) i při větších počtech objektů na plátně, tedy v situacích, kdy uživatel modeluje rozsáhlé a komplexní systémy. Pro otestování stability jsem (konkrétně objekty CPU) provedl řadu měření paměti využité zásuvným modulem SoC diagramu pro různé počty objektů na plátně diagramu. K měření jsem použil nástroj VisualVM [27], který je součástí balíku pro Java vývojáře - JDK (*Java Development Kit*). Z obrázku 26 je zřejmé, že spotřeba paměti roste přímo úměrně počtu objektů na plátně. Přesná data jsou uvedena v následující tabulce.

objekty na plátně	**využitá paměť [kB]**
0	3
5	43
10	84
15	124
20	164
25	204
30	244
35	284
40	324
45	364
50	404



Obrázek 26: Graf závislosti využité paměti na počtu objektů diagramu

6.1 Možná rozšíření

Činnost projektu Lissom ani vývoj nástroje Codasip Studio nekončí, a proto existuje celá řada možností, jak výsledný nástroj ještě více integrovat a přizpůsobit uživatelskému komfortu. Některá rozšíření se naopak ubírají směrem ke standardizaci tak, aby byly jednotlivé modely použitelné i v nástrojích třetích stran.

6.1.1 Hromadné targety SoC projektů

Pro simulaci celého víceprocesorového systému je nutné, aby jeho jednotlivé projekty byly přeloženy a připraveny pro simulaci. Proto bude do zmíněného Target view implementována také podpora pro hromadné cíle celého SoC projektu. Jedním kliknutím tak bude vývojář schopen přeložit všechny projekty, které jsou v daném víceprocesorovém systému obsaženy.

6.1.2 Generování výstupu v jazyce IP-XACT

Standard IP-XACT je poměrně rozšířený nástroj pro popis elektronických komponent. Proto by bylo vhodné implementovat výstup víceprocesorových systémů do tohoto standardu tak, aby se modely vytvořené v Codasip Studio daly použít i v jiných nástrojích.

6.1.3 Grafický editor IP Core komponent

IP Core komponenty jsou po vygenerování pomocí průvodce popsány jazykem IP-XACT. Obecně rozsáhlejší komponenty se mohou stát v textové podobě velice těžko čitelnými. Z tohoto důvodu by bylo vhodné implementovat editor postavený na grafickém uživatelském rozhraní.

6.1.4 Online dokumentace

K online projektům je na serveru Lissom dostupná také dokumentace, obvykle ve formátu PDF. Pro platformu Eclipse existují zásuvné moduly, které umožňují zobrazovat tento typ dokumentů přímo v okně editoru. Dokumentace by se tak mohla stahovat a importovat do Workspace podobně jako online projekty. Jiným přístupem by mohla být dokumentace v podobě webových stránek, které je rovněž možné v Eclipse zobrazit. V tomto případě by měl uživatel po ruce vždy aktuální verzi dokumentace.

6.1.5 Vizuální průvodce

Pro začínající uživatele je tak komplexní systém jako Codasip Studio poměrně nepřehledným prostředím. Start s tímto prostředím by mohl být urychlen pomocí speciálního průvodce, který by uživatele provedl jednotlivými kroky pracovních postupů v tomto prostředí.

7 Závěr

Cílem prvních kapitol práce je seznámit čtenáře s aktuální situací na poli HW/SW codesign. Děje se tak skrze osvětlení této problematiky a seznámení s projektem Lissom. Dále jsou představeny frameworky GMF a EMF, které jsou využity k implementaci nástroje, jímž se tato práce zabývá. Ve druhé polovině práce je čtenář seznámen s analýzou požadavků a návrhem nástroje. Poznatky z této části jsou pak uplatněny v kapitole, která se věnuje implementaci nástroje. Zejména některým netriviálním problémům, se kterými jsem se během implementace setkal, a jejich řešením v podobě několika algoritmů.

Praktickým výsledkem mé diplomové práce je zejména nástroj pro grafický návrh víceprocesorových systémů a sady podpůrných nástrojů pro prostředí Codasip Studio. Výsledné nástroje jsou plně integrovány do celého prostředí a v rámci projektu Lissom budou použity k rychlému a efektivnímu vývoji. Implementace uvedených nástrojů přispěje ke zvýšení konkurenceschopnosti celého projektu a nabízí podobné funkce, které jsou v tomto odvětví považovány za uživatelský standard. Současnou podobu nástroje je možné dále rozvíjet a přizpůsobovat požadavkům HW/SW codesign vývojářů, kteří s nástrojem budou pracovat.

Při práci na nástrojích jsem využil předchozích zkušeností s prostředím Eclipse a frameworky GMF a EMF, které jsem získal při tvorbě bakalářské práce [9]. Zužitkoval jsem také zkušenosti z projektu Lissom a vědomosti o jazyce CodAL. Novinkou pro mě byla tvorba grafického uživatelského rozhraní pro prostředí Eclipse. Dále jsem se seznámil s několika standardy (IP-XACT, XML) a osvojil si praktickou implementaci některých návrhových vzorů (Observer, Strategy, Command, Factory).

Literatura

[1] HOFFMANN, A. H. MEYR a R. LEUPERS. *Architecture exploration for embedded processors with LISA*. Boston: Kluwer Academic Publishers, 2002. ISBN 14-020-7338-0.

[2] FUČÍK, O. *Prezentace k předmětu HSC: Úvod*. Brno: FIT VUT v Brně, 2011.

[3] *Project Lissom* [online]. 2006, verze 5.10.2012 [cit. 2013-01-02]. Dostupné z: http://www.fit.vutbr.cz/research/groups/lissom/project.html

[4] HRUŠKA, T. a K. MASAŘÍK. Structural Equivalence between Architectural Descriptive and Hardware Languages. In: *A proceedings volume from the 4th International Conference on Cybernetics and Information ...*. Florida, US: International Institute of Informacs and Systemics, 2007, s. 40-45. ISBN 1-934272-10-8.

[5] APS BRNO S.R.O. *CodAL Manual*. Brno: 2011.

[6] Codasip Studio [online]. verze 2013 [cit. 2013-05-04]. Dostupné z: https://www.codasip.com/products/studio/

[7] ILČÍK, O. *Nástroj pro grafické prototypování vestavěných systémů*. Brno: 2011. diplomová práce. FIT VUT v Brně.

[8] APS BRNO S.R.O. *Codasip Framework Manual*. Brno: 2011.

[9] NETOČNÝ, O. *Transformace grafické reprezentace procesoru do jazyka pro popis architektury*. Brno: 2011. bakalářská práce. FIT VUT.

[10] CDT. *Eclipse* [online]. verze 2.3.2013 [cit. 2013-04-03]. Dostupné z: http://www.eclipse.org/cdt/

[11] *Eclipse* [online]. verze 20.12.2012 [cit. 2013-01-03]. Dostupné z: http://www.eclipse.org/

[12] D'ANJOU, J. et al. *Java Developer's Guide to Eclipse*. 2nd ed. Boston: Adison-Wesley Professional, 2004. ISBN 987-0-321-30502-2.

[13] STEINBERG, D. et al. *EMF: Eclipse Modeling Framework*. 2nd ed. Boston: Pearson Education, Inc. 2009. ISBN 987-0-321-33188-5.

[14] Graphical Modeling Framework (GMF) Tooling. *Eclipse* [online]. verze 10.10.2012 [cit. 2013-01-03]. Dostupné z: http://eclipse.org/gmf-tooling/

[15] RUMBAUGH, J. I. JACOBSON a G. BOOCH. *The Unified Modeling Language Reference Manual*. Reading MA: Addison Wesley Longman, Inc. 1999. ISBN 0-201-30998-X.

[16] ARLOW, J. a I. NEUSTADT. *UML a unifikovaný proces vývoje aplikací*. Brno: Computer Press, 2003. ISBN 80-7226-947-X.

[17] XML Technology. *W3C* [online]. 2010 [cit. 2013-04-15]. Dostupné z: http://www.w3.org/

standards/xml/

[18] IEEE Std 1685-2009. *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating and Reusing IP within …*. IEEE. Třídící znak 18 2010.

[19] GEF (Graphical Editing Framework). *Eclipse* [online]. verze 28.9.2012 [cit. 2013-01-04]. Dostupné z: http://www.eclipse.org/gef/

[20] *XMOS* [online]. verze 11.12.2012 [cit. 2013-01-03]. Dostupné z: http://www.xmos.com/

[21] SWT: The Standard Widget Toolkit. *Eclipse* [online]. verze 14.9.2012 [cit. 2013-01-03]. Dostupné z: http://www.eclipse.org/swt/

[22] HRUŠKA, T. a R. BURGET. *Internetové aplikace (WAP) II.: část SGML, HTML, CSS, DOM*. Brno: VUT, 2012.

[23] GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Massachusetts: Addison-Wesley, 1995. ISBN 02-016-3361-2.

[24] HTTP Authentication: Basic and Digest Access Authentication. *RFC* [online]. 1999 [cit. 2013-04-14]. Dostupné z: http://tools.ietf.org/html/rfc2617

[25] Apache HttpComponents. *Apache* [online]. 2005, verze 2013-03-24 [cit. 2012-04-13]. Dostupné z: http://hc.apache.org/

[26] How You've Changed!: Responding to resource changes in the Eclipse workspace. *Eclipse* [online]. 2002, verze 2004 [cit. 2013-04-21]. Dostupné z: http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html

[27] *Project Kenai* [online]. verze 13.11.2012 [cit. 2013-05-12]. Dostupné z: http://visualvm.java.net/

A Obsah CD

Samotný systém Codasip IDE včetně Codasip Studio je komerčním produktem a nemůže být na tomto CD umístěn, zájemcům bude prezentován na požádání autora. Přiložené CD má následující obsah:

- **doc** - technická zpráva,
- **src** - zdrojové kódy zásuvných modulů pro Codasip Studio, které byly během této práce vytvořeny nebo vygenerovány a upraveny.