VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

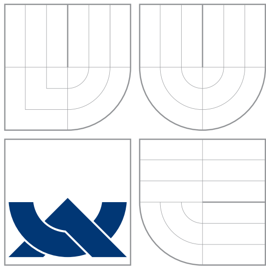
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO GRAFICKÉ PROTOTYPOVÁNÍ VESTAVĚNÝCH SYSTÉMŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

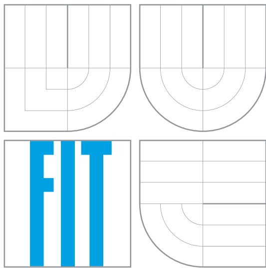
AUTOR PRÁCE Bc. ONDŘEJ ILČÍK
AUTHOR

BRNO 2011



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 VESTAVĚNÝCH SYSTÉMŮ

TOOL FOR GRAPHICAL PROTOTYPING OF THE EMBEDDED SYSTEMS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. ONDŘEJ ILČÍK

AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. TOMÁŠ HRUŠKA, CSc.

SUPERVISOR

BRNO 2011

Abstrakt

Práce se věnuje problematice grafického modelování vestavěných systémů pomocí dialektů jazyka UML a předkládá stručný souhrn existujících profilů. Dále se věnuje nástrojům pro podporu modelování na platformě Eclipse. V poslední části popisuje implementaci grafického modelování pro účely projektu Lissom.

Abstract

This study is focused on grafical modeling of embedded systems using dialects of UML. It provides a brief description of existing profiles. Furthemore it deals with modeling frameworks for the Eclipse platform and describes an implementation of such modeling tool as a part of project Lissom.

Klíčová slova

Hardware/software codesign, vestavěné systémy, grafické modelování, SoC, MDD, UML, SysML, MARTE, Lissom, Eclipse, EMF, GMF.

Keywords

Hardware/software codesign, embedded systems, graphical modeling, SoC, MDD, UML, SysML, MARTE, Lissom, Eclipse, EMF, GMF.

Citace

Ondřej Ilčík: Nástroj pro grafické prototypování
vestavěných systémů, diplomová práce, Brno, FIT VUT v Brně, 2011

Nástroj pro grafické prototypování vestavěných systémů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.

....
Ondřej Ilčík
25. května 2011

Poděkování

Děkuji Ing. Karlu Masaříkovi, PhD. za odborné konzultace a cenné připomínky.

© Ondřej Ilčík, 2011.
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 Grafické modelování vestavěných systémů **4**
2.1 Úvod do problému . . . 4
2.2 UML 2 . . . 5
2.2.1 Nástroje pro rozšíření UML . . . 5
2.3 SysML . . . 6
2.3.1 Popis struktury a behaviorální popis . . . 6
2.3.2 Popis požadavků . . . 7
2.3.3 Popis alokací . . . 7
2.4 MARTE . . . 7
2.5 UML for SoC . . . 8
2.6 UML for SystemC . . . 9
2.7 Související technologie a standardy . . . 9
2.7.1 IP-XACT . . . 9
2.7.2 AADL . . . 10
2.8 Existující nástroje . . . 10
2.8.1 No Magic Magic Draw . . . 10
2.8.2 CoFluent Studio . . . 11
2.8.3 Eclipse Papyrus UML . . . 11
2.8.4 IBM Rational Rhapsody . . . 12

3 Podpora modelování v Eclipse **13**
3.1 EMF . . . 13
3.1.1 Vztah EMF a jiných standardů . . . 14
3.1.2 Popis modelu . . . 15
3.1.3 Validace modelu . . . 16
3.1.4 Práce s modelem . . . 16
3.2 GEF . . . 17
3.3 GMP . . . 18
3.3.1 Koncept vývoje komponent editoru . . . 18

4 Projekt Lissom **21**
4.1 Lissom IDE . . . 22

5 Požadavky na vytvářený nástroj **25**

6 Analýza požadavků a návrh řešení **26**

7 Realizace modelovacího nástroje 28
7.1 Vytvoření modelu a jeho grafického editoru 28
7.1.1 Doménový model 28
7.1.2 Grafická podoba 30
7.1.3 Validace modelu 32
7.1.4 Úpravy editoru 34
7.2 Integrace editoru do vývojového prostředí 35
7.2.1 Propojení s jednotlivými částmi prostředí 36

8 Výsledné řešení 42
8.1 Možná rozšíření 43
8.1.1 Interní vazby modelovaného systému 43
8.1.2 Podpora více aplikací pro jednu platformu 43
8.1.3 Knihovna znovupoužitelných bloků v paletě nástrojů 44
8.1.4 Interní vazby modelovaného systému 44
8.1.5 Validace v reálném čase 44

9 Závěr 45

A Použité zkratky 49

B Obsah CD 51

C Návod na instalaci 52

D Formáty souborů 53
D.1 Formát modelu konfigurace 53
D.2 Formát interní konfigurace projektu 53
D.3 Formát simulační konfigurace projektu 54

Kapitola 1

Úvod

V poslední době pozorujeme výrazný nárust složitosti vestavěných systémů. Ten je umožněn technickým pokrokem na jedné straně a podporován poptávkou po vyšším výkonu z různých oblastí použití na straně druhé. Jde např. o spotřební zařízení (HDTV, chytré telefony), automobilový průmysl nebo bezpečnostní prvky (biometrická zařízení). Tato zařízení často vyžadují pro svůj běh víceprocesorové systémy a komplexní obslužný software.

Důsledkem zvyšující se složitosti je stále komplikovanější a zdlouhavější návrh a analýza vestavěného systému a tím se snižující produktivita. Objevuje se potřeba nového přístupu k vývoji vestavěných systému, který by dokázal eliminovat zmíněný problém.

Zdá se, že řešením by mohla být relativně mladá metodologie pro *vývoj řízený modelem* (MDD – Model Driven Development), do té doby používaná spíše pro oblast softwarových systémů, a jazyk UML jako praktický nástroj pro realizaci této metodologie.

Tato práce se snaží zmapovat současné možnosti vývoje vestavěných systémů pomocí MDA a UML a vytvořit nástroj umožňující prototypování vestavěných systémů pro projekt Lissom, probíhající na Fakultě informačních technologií VUT.

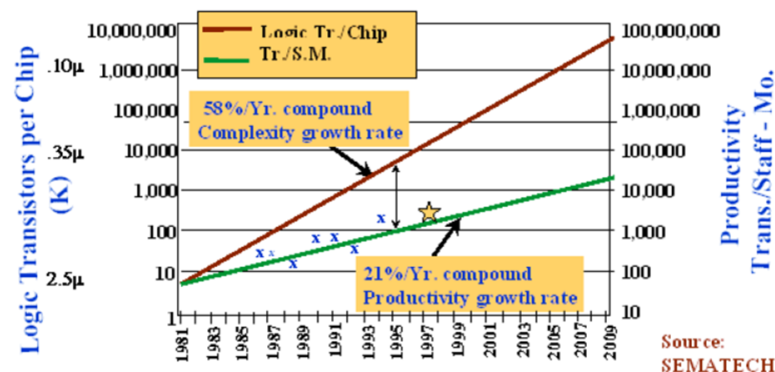
Dílo je členěno na osm kapitol. První shrnuje současný stav na poli grafického modelování vestavěných systémů. Druhá kapitola popisuje možnosti modelování na platformě Eclipse pomocí rozhraní EMF a GMF. Třetí kapitola seznamuje s projektem Lissom a jeho potřebami pro rychlé prototypování systémů. Čtvrtá kapitola shnuje konkrétní požadavky implementovaný grafický modelovací nástroj. Další kapitoly provázejí analýzou, návrhem a implementací takového nástroje. Předposlední kapitola shrnuje dosažený výsledek a popisuje možná rozšíření v budoucnosti.

Kapitola 2

Grafické modelování vestavěných systémů

2.1 Úvod do problému

Jak již bylo naznačeno v úvodu, vývoj vestavěných systémů se v současné době potýká s problémem označovaným jako *productivy gap* [4][6], jak ukazuje obr. 2.1. Složitost vestavěných systémů se zvojnásobí každých 10 měsíců[26]. Podle Fujitsu [14] připadalo v roce 2005 na verifikační proces více než 50% celkového času vývoje čipu. Okolo 65% čipů si vyžádalo tzv. re-spin, tedy nový návrh. Dá se předpokládat, že při zachování konvenčních metod vývoje bude toto číslo v současnosti ještě vyšší.



Obrázek 2.1: Nepoměr mezi narůstající složitostí a produktivitou [3]

Ve snaze vyřešit tento problém se hledají takové způsoby návrhu a analýzy vestavěných systémů, které umožní dosáhnout vyšší úrovně abstrakce. Jednou z možností je vydat se stejnou cestou jako už dříve čistě softwarové systémy – zavést vývoj řízený modelem (MDD - Model Driven Development).

Vývoj řízený modelem umožňuje oddělit specifikaci systému ve formě modelu od jeho fyzické realizace. Vyšší abstrakce umožňuje snazší specifikaci systému a sdílení informací založené na formálních modelech [22]. Formalizací MDD je standard MDA (Model Driven Architecture) [15] organizace OMG[1] z roku 2001.

Nástrojem pro modelování systémů v MDA je jazyk UML, který bude – včetně svých specifických dialektů – popsán v následující části této kapitoly.

V závěru kapitoly jsou zmíněny některé další standardy související s modelováním vestavěných systémů.

2.2 UML 2

Unified Modeling Language je obecný grafický modelovací jazyk vytvořený a spravovaný skupinou OMG, který vznikl spojením konceptů Booch, OMT a OOSE[18]. Poskytuje 13 různých diagramů pro popis struktury, chování a interakce. Zásadních změn doznal ve druhé verzi, která umožnila použít UML jako skutečný programovací jazyk (modelování s podporou automatického generování kódu), zejména díky podpoře funkční sémantiky – díky ní lze vyjádřit akce UML objektů[5].

Cílem UML je pouze poskytnout notaci, samo o sobě nedefinuje ani nenařizuje žádnou metodologii pro vývoj. Přestože je používáno hlavně pro vývoj softwarových systémů, je možné jej díky dostatečně výrazovosti a zejména rozšiřitelnosti použít i pro vývoj vestavěných systémů [13].

Pro tuto oblast jsou významné zejména následující diagramy:

- **Diagram tříd** , zobrazující statickou strukturu systému, jeho vnitřní podobu a relace mezi částmi systému.
- **Diagram komponent**, který zobrazuje modulární části systému.
- **Diagram vnitřní struktury**, zobrazující interní strukturu systému nebo jeho části. Důležitou vlastností jsou zde porty a rozhraní, které dokáží zachytit styčná místa mezi objektem a vnějším světem.
- **Sekvenční diagram**, zachycující vzájemnou komunikaci objektů. Může být použit pro tvorbu testů.
- **Diagram nasazení**, zobrazující běhovou konfiguraci softwarových a hardwarových částí systému.

Přestože je UML jazyk obecný, je u něj poznat jeho orientace na softwarové systémy. Pro oblast vestavěných systémů tak např. chybí nebo je nedostatečná přímá podpora analytického modelování, modelování časových vlastností systému nebo podpora modelování ne-funkčních vlastností (NFP). Další překážkou je nejasná sémantika, což je způsobeno příliš širokým záběrem jazyka. Naštěstí UML podporuje rozšiřování pomocí tzv. profilů, které tyto problémy eliminují, o čemž pojednává následující podkapitola.

2.2.1 Nástroje pro rozšíření UML

Jazyk UML zahrnuje od verze 2.0 mechanismus pro rozšíření, který upravuje UML pro specifickou problémovou doménu. Tento mechanismus podporuje úpravy původní sémantiky striktně aditivním způsobem tak, aby nemohly odporovat původnímu významu[2].

Profil je kolekce následujících rozšíření:

- **Stereotyp** , který definuje, jak lze rozšířit existující metatřídu. Umožňuje tak použít doménově specifickou terminologii.
- **Tag definition** nebo také *metaproperty* jsou vlastnosti stereotypu.

[1]Object Managament Group – http://www.omg.org/

- **Constraint** definuje podmínku nebo omezení, kterou musí patřičný element splňovat. Ta může být zapsána různými způsoby – přirozeným jazykem, matematicky, programovacím jazykem nebo pomocí jazyka OCL (Object Constraint Language)[18].

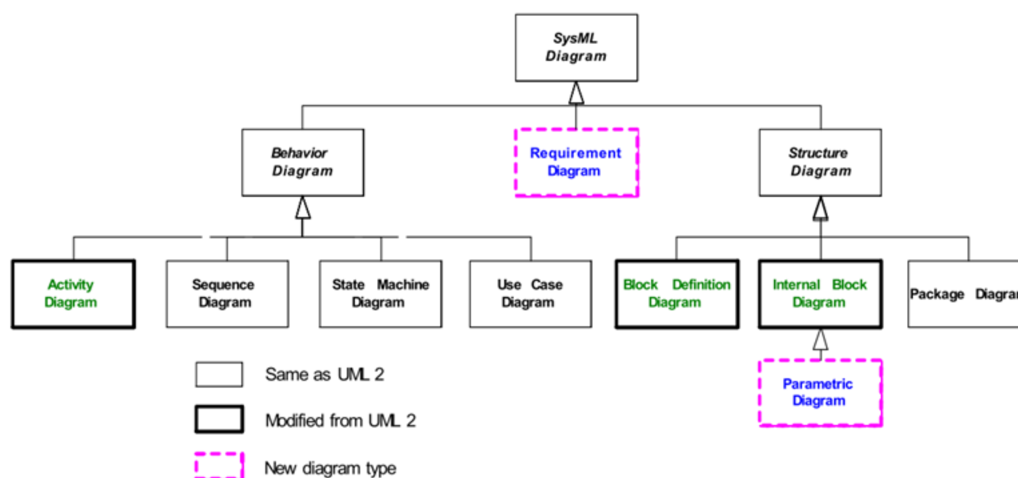
Tvorba vlastního profilu zahrnuje tři fáze[26]. V první fázi jsou odstraněny ty části metamodelu, které pro budoucí profil nejsou přínosné. V dalším kroku je metamodel rozšířen pomocí stereotypů a metavlastností. V poslední fázi je přidána grafická notace.

2.3 SysML

Profil SysML[17] definuje doménově nezávislý jazyk pro systémové inženýrství. Podporuje specifikaci, analýzu, návrh, verifikaci a validaci široké škály systémů od hardwarových, přes softwarové, informační až po procesní nebo lidské.

Přestože jde o jazyk spíše obecný, má své využití i v oblasti vestavěných systémů díky společným zájmům – jak systémové inženýrství tak vestavěné systémy potřebují nástroj pro přesné modelování požadavků, specifikaci heteregenních systémů, simulaci, verifikaci a validaci.

SysML používá podmnožinu UML a rozšiřuje ji o prvky potřebné v systémovém inženýrství. Taxonomie profilu je zobrazena na obr 2.2.



Obrázek 2.2: Taxonomie SysML [17]

Vlastnosti SysML[26]:

2.3.1 Popis struktury a behaviorální popis

SysML zjednodušuje UML diagramy používané pro reprezentaci struktury systému. Zavádí koncept tzv. *bloku*, stereotypu, který popisuje systém jako strukturu propojených částí. *Block* poskytuje doménově neutrální modelovací element, který může být použit pro libovolný druh systému. V kontextu vestavěných systémů může jít jak o hardware, tak o software.

SysML přináší několik vylepšení Activity diagramu z UML, jde např. o možnost zrušit již spuštěnou akci.

2.3.2 Popis požadavků

Jedno ze zásadních vylepšení SysML oproti UML je podpora pro reprezentaci požadavků a možnost jejich svázání z modelem a testovacími procedurami. Požadavky jsou tak přímou součástí návrhu systému, narozdíl od UML, kde je zvykem požadavky modelovat pomocí Use Case diagramů.

2.3.3 Popis alokací

Koncept alokací je více abstraktní formou *nasazení* z UML. Definuje vztah mezi elementy za pomoci mapování zdroje na cíl. Může být použito např. na mapování požadavků na části systému, mapování chování na implementující strukturu nebo asociaci hardware a obslužného software.

Profil přináší nebo upravuje tyto diagramy:

- **Requirement diagram** je nový diagram, který umožňuje vývojáři modelovat požadavky a vztahovat je k elementům, které je splňují nebo verifikují.
- **Parametric diagram** je také nový diagram, určený pro modelování parametrů systému.
- **Block definition diagram** vychází z diagramu tříd z UML a definuje vlastnosti bloku a relace mezi bloky.
- **Internal block diagram** vychází z diagramu struktury a zobrazuje propojení částí systému a informační toky mezi bloky.
- **Activity diagram** vychází ze stejnojmenného diagramu z UML.

Nevýhodou jazyka z pohledu vývoje vestavěných systémů je nedostatečná sémantika jako daň za široký záběr jazyka[26]. Stejně jako v případě UML není součástí metodologie pro vývoj. SysML podporuje generování kódu z modelu, existují např. nástroje pro vytvoření SystemC reprezentace z modelu SysML. Protože však SysML samo o sobě nestačí k modelování všech aspektů vestavěného systému, je třeba jej použít společně s některým více specializovaným profilem.

2.4 MARTE

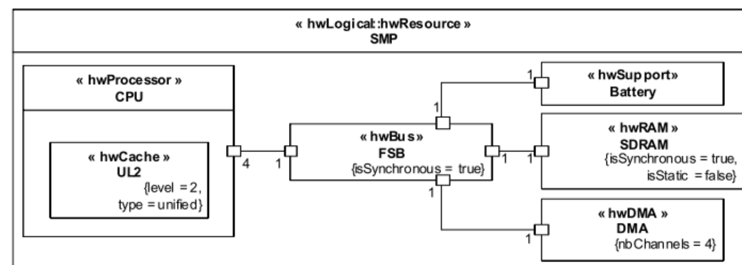
Profil MARTE[19] (Modeling and Analysis of Real-Time and Embedded systems) je určen pro modelování a analýzu real-time vestavěných systémů pomocí MDA metodologie. Pokrývá hardwarové i softwarové aspekty vývoje. Nahrazuje profil UML Profile for Schedulability, Performance and Time.

Jedním z hlavních cílů tohoto profilu je definovat pomocí různých pohledů a modelů aplikaci (včetně funkčních a ne-funkčních vlastností), hardwarovou architekturu a alokaci aplikace na dostupné hardwarové zdroje. Dostatečně přesná sémantika by v takovém případě umožnila automatizovat generování kódu pro simulaci na různých úrovních abstrakce nebo syntetizovat části hardware.

MARTE tvoří 4 balíky:

1. **Foundations**, definující zakladní elementy, jako je kauzální model, modelování nefunkčních vlastností systému, modelování času, modelování zdrojů na systémové úrovni a modelování alokací.
2. **Design model**, přinášející komponentní model pro kompatibilitu s jinými standardy (SysML, AADL, CCM), podporu souběžnosti a detailní modelování hardware i obslužné aplikace.
3. **Analysis model**, který definuje anotační modely, jež by umožnily spolupráci s existujícími analyzačními nástroji.
4. **Annexes** jsou přílohy standardu.

Některé zdroje [7] vidí MARTE méně užitečný díky chybějící podpoře pro *design space exploration*, syntézu, modelování požadavků, formální analýzu. Nedostatky lze částečně vyřešit současným využitím i jiných profilů, např. SysML.



Obrázek 2.3: Ukázka MARTE modelu SMP architektury, popsaného pomocí HRM (Hardware Resource Modeling)

2.5 UML for SoC

Profil UML for SOC je iniciativou firem Rational (nyní IBM) a Fujitsu, od roku 2006 dostupný jako standard OMG[20]. Jak název napovídá, je určen k popisu systému na čipu pomocí jazyka UML, a to od úrovně RTL (Register Tranfer Level) po TLM (Transactional Level Modeling). Snahou je maximální možné znovupoužití standardního UML s minimem rozšíření. Cílem profilu je poskytnout následující schopnosti:

1. **Hierarchická reprezentace modulů a kanálů**, které jsou základními elementy profilu.
2. **Role modulů**
3. **Informace přenášené mezi moduly**

Některé stereotypy definované v profilu:
Pro grafickou notaci je použit rozšířený diagram struktury. Profil podporuje automatické generování SystemC kódu.
Nevýhodou UML for SoC je, že řeší návrh vestavěných systémů pouze z pohledu formalizace specifikace a pokrytí testy. Mezi omezení profilu patří nemožnost modelovat ne-funkční prvky systému a použití proprietárního jazyka CWL pro popis rozhraní[7].

Tabulka 2.1: Některé stereotypy profilu UML for SoC

SoC Element	Stereotyp	UML metatřída
Module	SoCModule	Class
Process	SoCProcess	Operation
Controller	Controller	Class
Channel	SoCChannel	Class
Port	SoCPort	Port/Class
Connector	SoCConnector	Connector
Clock port	SoCClock	Port
Reset port	SoCReset	Port

2.6 UML for SystemC

System C je otevřený standard pro návrh SoC na systémové úrovni[23]. Je prosazován OSCI[2]a standardizován jako IEEE 1666-2005. Na poli návrhu a analýzy SoC je významným hráčem. Jde o sadu tříd a maker pro jazyk C++, které umožňují událostně řízenou simulaci[27]. Umožňuje simulovat souběžné procesy.

Pro tento jazyk byl vytvořen UML profil, který zachycuje jak strukturální tak behaviorální vlastnosti jazyka. Tento profil umožňuje specifikovat, analyzovat, navrhnout, vizualizovat a dokumentovat softwarové i hardwarové části vyvíjeného SoC. Profil vychází z vrstev jazyka SystemC, tzv. *SystemC core layer* a *SystemC layer of predefined channels, ports and interfaces.*

Skládá se z těchto tří částí:

1. **Structure and Communication** definuje stereotypy pro základní stavební bloky jazyka. Slouží k reprezentaci hierarchické struktury a komunikačních bloků vytvořených z modulů, rozhraní, portů a kanálů.

2. **Behavior and Synchronization** definuje stereotypy související se stavovými stroji, které umožňují popsat chování SystemC procesů

3. **Data types** definuje datové typy specifické pro SystemC.

2.7 Související technologie a standardy

S modelováním vestavěných systémů souvisí i standardy, které nevychází z UML. Z hlediska této práce se jedná spíše o okrajové téma, ale vzhledem k jejich rozšířenosti a důležitosti je vhodné se o nich alespoň zmínit.

2.7.1 IP-XACT

IP-XACT[12] je IEEE standard pro popis elektronických komponent a jejich návrhů pomocí jazyka XML. Byl vytvořen konsorciem SPIRIT[3]. Cílem je sjednocení popisu komponent pomocí nezávislého formátů, který umožní výměnu knihoven návrhů mezi jednotlivými nástroji.

[2] Open SystemC Initiative

2.7.2 AADL

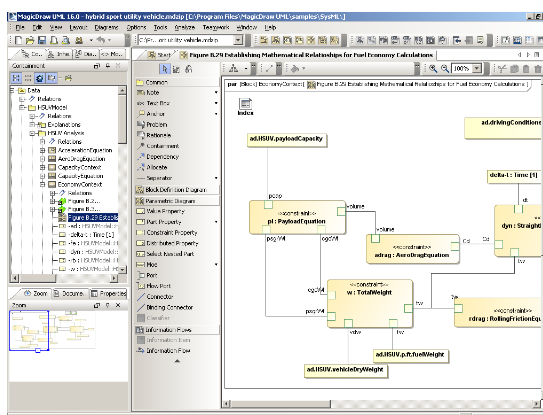
Architecture Analysis And Design Language je jazyk pro popis architektury standardizovaný organizací SAE[4]. Vychází z jazyku MetaH od firmy Honeywell. Používá se pro modelování harware a software vestavěných systémů. Slouží pro dokumentaci, analýzu a generování kódu.

2.8 Existující nástroje

Následující podkapitola si klade za cíl stručně seznámit s některými existujícími nástroji pro grafické modelování vestavěných systémů. Jsou zastoupeny jak komerční, tak volně dostupné varianty.

2.8.1 No Magic Magic Draw

Magic Draw[5]společnosti No Magic, Inc. je obecný modelovací nástroj s podporou modulů pro rozšíření funkcionality. Tyto moduly mohou být dodávány třetími stranami a podporují jak vývoj řízený modelem obecně, tak modelování vestavěných systémů a hardware. Umožňuje modelovat systémy v jazycích UML (a jeho dialektech jako SysML), BPMN a UPDM. Nástroj DSL Customization Engine, který je součástí Magic Draw, umožňuje rozšířit toto prostředí o podporu doménově specifických jazyků.



Obrázek 2.4: Modelování SysML v prostředí MagicDraw (zdroj: http://www.magicdraw.com/screenshots/)

Z hlediska modelování vestavěných systémů je zajímavý podporou SysML a MARTE. Podpora SysML dle specifikace 1.2 zahrnuje všechny jeho diagramy a umožňuje tak specifikace, analýzu, návrh a validaci širokého spektra systémů. Velice zajímavé jsou v tomto ohledu moduly třetích stran. Např. ParaMagic vyvíjený společností InterCAX LLC dramaticky rozšiřuje možnosti simulace systémů popsaných v SysML díky propojení s MS Excel, Matlab/Simulink a Mathematica. MagicDraw podporuje i příbuzné standardy UPDM a DoDAF, povinně používané pro modelování armádních systémů.

[4]Skupina výrobců a uživalů nástrojů pro automatizaci návrhu elektronických systémů. Členy konsorcia je mnoho významných firem z oblasti vestavěných systémů, např. ARM Holdings, Cadence Design Systems, Freescale Semiconductor, Mentor Graphics, Synopsys, Texas Instrumets a další.

[4]Profesní sdružení odborníků z oblasti leteckého, automobilové a dopravního průmyslu.

V podpoře standardů a integraci s existujícími nástroji je v současné době nejspíše bezkonkureční.

2.8.2 CoFluent Studio

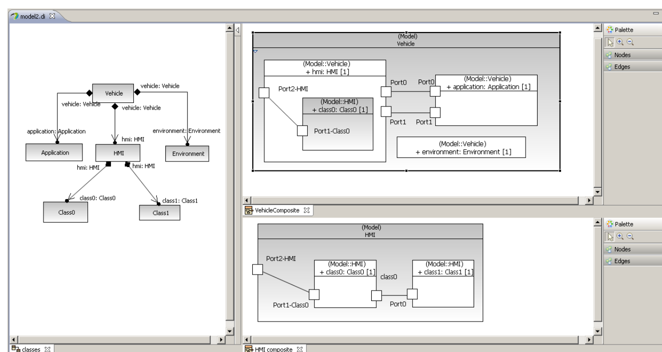
Produkt společnosti CoFluent Design[6]nabízí novou metodologii vývoje řízeného aplikací, založenou na kombinaci standardů SysML, MARTE a vlastního profilu UML. Snahou této metodologie je definovat vhodnou architekturu během prvních 20 procent životního cyklu projektu. Klíčem k tomu je vytvořit oddělené modely pro aplikace a platformu a umožnit jejich kombinování s cílem získat poznatky o výsledném systému ještě před jeho detailním návrhem a integrací[8].

Modely vytvořeného pomoci CoFluent Studia jsou přeloženy na TLM úroveň v podobě kódu pro SystemC. Je možné analyzovat a simulovat i víceprocesorové nebo vícejádrové systémy a pozorovat tak chování komplexní systémů a analyzovat jejich výkonnostní parametry. CoFluent Studio má silnou podporu pro tzv. *design space exploration*: umožňuje definovat různé hardwarové platformy (jedno- nebo víceprocesorové, s různými stupně vzájemného propojení ap.), nabízí generické znovupoužitelné komponenty a umožňuje zkoumat chování systémů různým mapováním aplikací na platformy bez nutnosti úpravy kódu. V modelech je možné zanést i ne-funkční parametry systémů.

2.8.3 Eclipse Papyrus UML

Papyru[7]poskytuje prostředí pro editaci modelů založených an EMF, zejména těch, které vycházejí z UML 2.2, jako jsou SysML nebo v budoucnosti MARTE. Umožnuje snadnou implementaci vlastních DSL jazyků postavených nad UML s důrazem na přizpůsobení libovolné části editoru. Editor může být jak grafický, tak textový.

Papyrus se v roce 2010 stal jedním z podprojektů Eclipse, což mj. vyústilo v přepis velké části jeho kódu. Některé jeho původní části jsou v současnosti ještě nedokončené. Jde např. o podporu profilu MARTE. Do budoucna se však jedná o zajímavý nástroj, přinášející na platformu Eclipse možnost snadného vytvoření vlastního dialektu UML a příslušných editorů.



Obrázek 2.5: Prostředí Papyrus (zdroj: http://wiki.eclipse.org/PapyrusUserGuide/)

[5]http://www.magicdraw.com/
[6]http://www.cofluentdesign.com
[7]http://www.eclipse.org/modeling/mdt/papyrus/

2.8.4 IBM Rational Rhapsody

Rational Rhapsody[8]je grafické vývojové prostředí pro systémové inženýry a vývojáře software vytvářející real-time nebo vestavěné systémy. Používá grafické modely ke generování software v jazycích, jako je C, C++ nebo Java. Umožňuje vyvíjet aplikace na úrovni modelu nebo kódu, přičemž obě dva způsoby popisu se vzájemně synchronizují. Podporuje standardy UML, SysML, DoDAF, MODAF a další.

[8]http://www-01.ibm.com/software/awdtools/rhapsody

Kapitola 3

Podpora modelování v Eclipse

Tato část práce se věnuje platformě Eclipse, která se díky svému otevřenému vývoji, robustnosti a možnosti snadného rozšíření těší v posledních letech velkému nárůstu zájmu mezi firmami[1]a institucemi, věnujícími se problematice modelování systémů. Eclipse je od třetí verze postaveno na průmyslovém standardu OSGi R4[21]. Jde o specifikaci modulárního dynamického systému, který umožňuje průběžnou instalaci, spuštění i zastavení aplikací, definuje životní cyklus aplikací a poskytuje infrastrukturu pro spolupráci mezi nimi. Eclipse je nejvíce známo jako prostředí pro vývoj v Javě, ale ve skutečnosti podporuje i desítky jiných jazyků (např. Ada, C++, Perl, Python, Ruby, . . .). Umožňuje také tvorbu RCP aplikací. Mezi nezmíněné, ale důležité vlastnosti patří podpora mnoha operačních systémů.

Nárůst zájmu o grafické modelování reflektuje i dostupnost poměrně velkého množství rozhraní, knihoven a projektů, které se modelováním v prostředí Eclipse zabývají. Ty jsou soutředěny pod zastřešujícím projektem Eclipse Modeling Project[2]. Protože praktická část této diplomové práce na nich staví, budou ty nejvýznamnější krátce popsány v následujících podkapitolách. V textu jsem čerpal zejména z knihy Eclipse Modeling Framework Second Edition[25] a webových stránek jednotlivých projektů[9].

3.1 EMF

Eclipse Modeling Framework[3]je modelovací nástroj určený pro vytváření aplikací na základě definice modelu. Je jádrem a nejdůležitější částí EMP. Zatímco standardní postup tvorby modelu jako součásti aplikace by nejspíše zahrnoval vytvoření UML diagramu, napsání odpovídajícího kódu v Javě, a popis podoby persistence dat (např. pomocí XML Schema). To znamená tři různé způsoby reprezentace toho samého. EMF naproti tomu poskytuje prostředí, které umožňuje provádět validaci, uložení a editaci modelu, který je tak popsán pouze jednou. EMF model lze chápat jako sjednocení Javy, UML a XML.

EMF tak umožňuje smazat rozdíl mezi modelováním a programováním. Namísto oddělení vysokoúrovňového modelování a nízkoúrovňového programování spojuje tyto koncepty dohromady. U rozsáhlých aplikací je toto rozdělení někdy nutné, nicméně míra separace je volbou programátora.

EMF se skládá ze tří částí:

[2]Mezi členy Eclipse Foundation, kteří se angažují ve vývoji modelovacích nástrojů, patří např. IBM nebo SAP

[2]http://www.eclipse.org/modeling/

[3]http://eclipse.org/emf/

1. **EMF**, který obsahuje metamodel *Ecore* pro popis modelů, podporu pro provoz a zasílání zpráv (notifikací), podporu persistence a reflexivní API pro manipulaci objektů
2. **EMF.Edit**, který obsahuje generické třídy pro tvorbu editorů EMF modelů. Jde o třídy, které umožňují zobrazení modelu pomocí standardních UI prvků, rozhraní pro práci s objekty, využívající návrhový vzor *command*, a podpora *undo* a *redo*.
3. **EMF.Codegen**, umožňující tvorbu kompletního editoru modelu (včetně GUI).

Jsou podporovány tři úrovně generování kódu:

- **Model**, poskytuje rozhraní a výchozí implementaci objektů modelu v jazyce Java, včetně implementace *factory* tříd.
- **Adapters**, který je rozhraním mezi objekty modelu a jejich vizuální reprezentací.
- **Editor**, umožňující vytvořit výchozí editor připravený pro úpravy.

3.1.1 Vztah EMF a jiných standardů

MDA

MDA, jak bylo zmíněno dřív, je standard asociace OMG. Jednou z jeho hlavních myšlenek je poskytnout uživateli nástroj pro vytvoření jednotného modelu, který lze transformovat a použít pro tvorbu různých nástrojů. EMF tuto myšlenku naplňuje díky podpoře generování nástrojů pro práci s modelem a persistenci jeho dat.

UML

Nejpoužívanějším jazykem v oblasti objektového modelování je UML. Je často využíván pro specifikaci a popis softwarových systémů, a díky dialektům jako SysML nebo MARTE, zmíněných v předešlých kapitolách, se používá i v oblasti systémového inženýrství a modelování vestavěných systémů. UML samo o sobě je však příliš abstraktní a komplexní, jediným společným aspektem je tak modelování tříd (EMF model je ve skutečnosti podmnožinou diagramu tříd). Je však k dispozici implementace UML metamodelu pro EMF.

XMI

XMI (XML Metadata Interchange) je standard pro výměnu metadat pomocí XML. Lze jej použít pro libovolný metamodel, který je možné vyjádřit pomocí MOF. EMF používá XMI jako kanonickou formu Ecore modelů[16].

MOF

MOF (Meta-Object Facility) je další ze standardů OMG pro modelem řízený vývoj. Původně vzniklo jako metamodelovací architektura pro popis UML. Skládá se ze 4 vrstev, od nejnižší po nejvyšší:

1. **M3**, je vrstva meta-meta modelu pro popis metamodelů vrstvy M2. Meta-meta model může být popsán sám sebou.
2. **M2**, konkrétní metamodel, např. jazyk UML

3. **M1**, vlastní model, obvykle DSL (doménově specificky jazyk)

4. **M0**, instance modelu

Ecore jazyka EMF odpovídá EMOF (Essential MOF), jde tedy o úroveň M3.

OCL

Jazyk OCL (Object Constraint Language) umožňuje definovat v metamodelu MOF (tedy např. UML) integritní omezení a invarianty. Lze tak popsat vlastnosti, které není možné vyjádřit pomocí diagramu.

Např. zápis

```
notation::Diagram.allInstances()->unique(name)
```

vynucuje, aby všechny diagramy v modelu měly unikátní jméno.

3.1.2 Popis modelu

Pro popis modelu slouží v EMF metamodel *Ecore*. Ten je popsán sebou samým, je tedy sobě metamodelem a z hlediska cizího modelu metametamodelem.

Definuje 4 základní typy objektů:

- **EClass**, který reprezentuje třídy modelu. Obsahuje identifikátor, a může obsahovat atributy a reference.
- **EAttribute** symbolizující atribut. Je popsán jménem a typem.
- **EDataType**, reprezentující jednoduchý datový typ asociovaný s primitivním datovým typem v Javě.
- **EReference**, který reprezentuje asociaci mezi třídami, přesněji řečeno jeden její konec. Stejně jako ostatní prvky má svůj identifikátor a navíc typ. Obsahuje zároveň další parametry, které upřesňují "sílu" vztahu (např. *asociace* vs. *agregace*).

Model v EMF může být vytvořen 3 různými způsoby:

1. **pomocí UML nástroje** (import z jiných nástrojů)
2. **pomocí vestavěného editoru Ecore**
3. **anotací rozhraní v jazyce Java**

Pro každou třídu v modelu je vygenerováno rozhraní v jazyce Java a příslušná výchozí implementace. Implementující třídy mají společného předka *EObject*, což je obdoba java.lang.Object pro objekty v EMF, díky které třída může informovat o změnách stavu nebo být uložena. Každé rozhraní obsahuje *get* a *set* metody, reprezentující atributy třídy a reference na jiné objekty. Součástí *set* metod je právě notifikace o změně atributu.

Podobu vygenerovaného modelu je v případě potřeby možné upravit pomocí šablon JET – Java Emitter Templates[1].

Adaptace objektů

Pro rozšíření funkcionality objektů bez potřeby dědičnosti nabízí EMF koncept adaptérů, které lze k danému objektu připojit. Klientský objekt jednoduše požádá továrnu (adapter factory) o adaptaci jiného objektu na typ klientem požadovaný. Ten je poté k němu připojen.

Adaptéry se používají i pro registraci pozorovatelů událostí (observers), vyvolaných daným objektem.

Reflexivní API

EMF umožňují použít vlastní reflexivní přístup k atributům třídy, což umožňuje např. pracovat i s modelem vygenerovaným za běhu. Zřejmou nevýhodou je negativní vliv na výkon.

3.1.3 Validace modelu

Podpora validace v EMF rozšiřuje možnosti, jak zajistit, aby byl model po celou dobu konzistentní a ve správném stavu. Je možné s její pomocí definovat invarianty (vlastnost systému, která se za žádných okolností nemění) a omezení (tvrzení, pravdivé v určitém čase nebo kontextu). Ty jsou popsány v jazycích Java nebo OCL. Invarianty jsou implementovány jako speciální operace třídy, které se týkají. Tyto operace vrací vždy typ *EBoolean*, udávající, byl-li invariant splněn.

EMF nabízí dva způsoby validace – dávkový a živý. Dávkový, jak už název napovídá, je volán volán explicitně, naproti tomu živý naslouchá změnám v modelu a ověřuje neporušení specifikovaných omezení v reálném čase.

3.1.4 Práce s modelem

V této fázi, kdy už máme k dispozici model, je na řadě zřejmě vytvoření nástroje pro jeho editaci. K tomuto účelu lze s výhodou použít rozhraní EMF.Edit, poskytující sadu tříd nezávislých na modelu pro tvorbu editorů EMF modelů. Jde zejména o:

- tzv. *content* a *label providers*. Content provider je bránou zobrazovacího elementu do vlastního modelu. Když má být zobrazena některá část modelu, zobrazovací element se spojí s content providerem a zjistí, o kterou část jde. Ten má zároveň na starost upozornit grafický element na změnu modelu a umožnit mu tak aktualizaci obsahu. Pro samotnou reprezentaci obsahu slouží label provider. Jeho cílem je vrátit text a ikonu reprezentující zobrazovanou část modelu. Tyto dva nástroje umožňují zobrazení modelu prostřednictvím grafických prvků knihovny JFace a okna Property View.
- rozhraní pro příkazy, které zavádí do modelu transakční zpracování a schopnost návratu o krok zpět o posunu dopředu.
- generátor výchozího editoru modelu, který lze dále upravovat.

Po implementaci, resp. úpravě všech poskytovatelů (*item providers*) máme k dispozici UI prvek pro prohlížení modelu. K tomu, aby bylo možné model i editovat, je třeba přidat podporu příkazů (*commands*). Mezi schopnosti příkazu patří:

- spuštění příkazu
- vrácení stavu zpět (undo) a posunutí dopředu (redo)

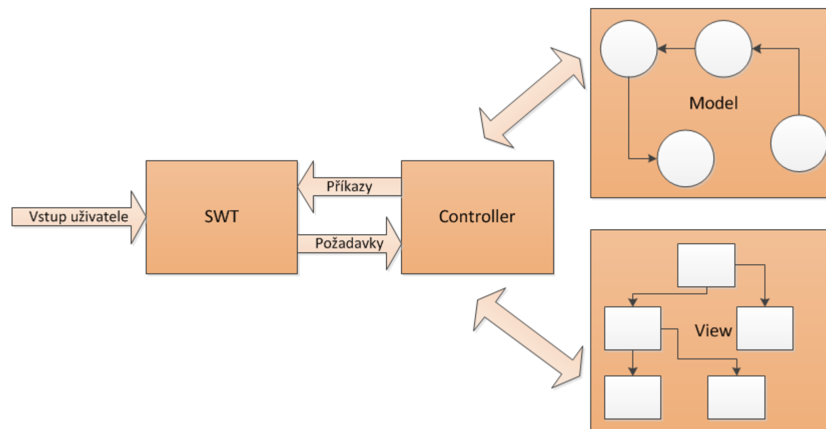
- ověření možnosti provedení operace (např. spuštění)
- navrácení výsledku (umožňuje skládání příkazů)
- vrácení postižených objektů

Příkazy jsou ukládány do zásobníku s podporou posunutí o krok vzad a před (ony zmíněné operace undo a redo).

Více o problematice práce s modelem se lze dočíst v knize Eclipse Modeling Framework[9].

3.2 GEF

Graphical Modeling Framework [4]je framework nad architekturou MVC, který umožňuje snadné vytváření grafických editorů. Poskytuje uživateli předdefinované elementy ve formě palety, se kterými je možné manipulovat a vzájemně je propojovat. GEF přitom v reálném čase snímá změny a promítá je do datového modelu na pozadí. Lze jej použít např. k tvorbě editorů diagramů tříd, nástrojů pro tvorbu uživatelského rozhraní a dokonce WYSIWYG textových editorů.



Obrázek 3.1: Architektura rozhraní GEF

Celé rozhraní je rozděleno na dvě části:

1. Draw2d, nástroj pro kreslení a práci s plátnem v grafické knihovně SWT
2. GEF, MVC rozhraní nad Draw2d

Ústředním prvkem Draw2d jsou grafické komponenty nazývané *figures*. Každá taková komponenta je tvořena jednoduchým objektem v Javě a má pravoúhlé hranice, které vymezují její tvar. Komponenta je asociována s instancí plátna v SWT. Draw2d se stará o přeposílání událostí vygenerových v SWT a překreslení plátna.

Návrh dle MVC rozděluje datový model, uživatelské rozhraní a řídicí logiku na samostatné části. Rozdělení umožňuje měnit každou z částí s minimálním dopadem na zbývající. Jednolivé části jsou v GEF implementovány takto:

- GEF nevyžaduje konkrétní podobu **modelu**. Musí však podporovat ukládání a zasílání změn.

- **Zobrazení** je realizováno buď pomocí grafických komponent (*figures*) nebo stromových struktur *TreeItems*.
- Každý vizualizovaný objekt má svůj vlastní **kontroler**. V GEF je kontroler nazývaný *EditPart* a je vazbou mezi částí modelu a jeho zobrazením. Obsahuje tzv. *EditPolicies*, které upřesňují způsob editace objektu a odezvy při práci s grafickou reprezentací.

GEF nabízí dva druhy pohledů na model – grafický pro zobrazení a úpravu grafických komponent a stromový pro zobrazení v hierarchické podobě, podobně jako nástroje knihovny JFace.

3.3 GMP

Nabízí se otázka, zda-li by bylo možné využít EMF a GEF k automatickému generování grafického editoru popsaného modelu. Tento problém řeší aplikační framework Graphical Modeling Project [5].

K popisu diagramů využívá model, který je rozdělen na sémantickou část a notaci (grafickou reprezentaci). Poskytuje sadu znovupoužitelných komponent jako jsou nástroje pro tisk, sady tlačítek nebo export obrázků. Podporuje rozšiřitelnost vytvořených editorů pomocí poskytovaných rozhraní.

Tvoří jej 3 základní části:

1. **GMF Tooling**, sloužící ke generování grafického editoru na základě dodaného modelu.
2. **GMF Runtime** je běhové prostředí.
3. **GMF Notation**, který řeší persistenci vizuálních dat (rozmístění komponent v diagramu ap.).

3.3.1 Koncept vývoje komponent editoru

Životní cyklus vývoje grafického editoru je zobrazen na následujícím diagramu. Kombinuje v sobě všechny potřebné kroky od vytvoření doménového modelu v Ecore až po generování grafického prostředí, jak je zobrazeno na následujícím diagramu. Pro vytvoření všech těchto částí je v k dispozici tzv. *Dashboard*, který provází uživatele tvorbou editoru. Pro každou část je k dispozici průvodce, takže v některých případech (ačkoliv to nebývá časté) je možné se úplně vyhnout ruční úpravě generovaného kódu.

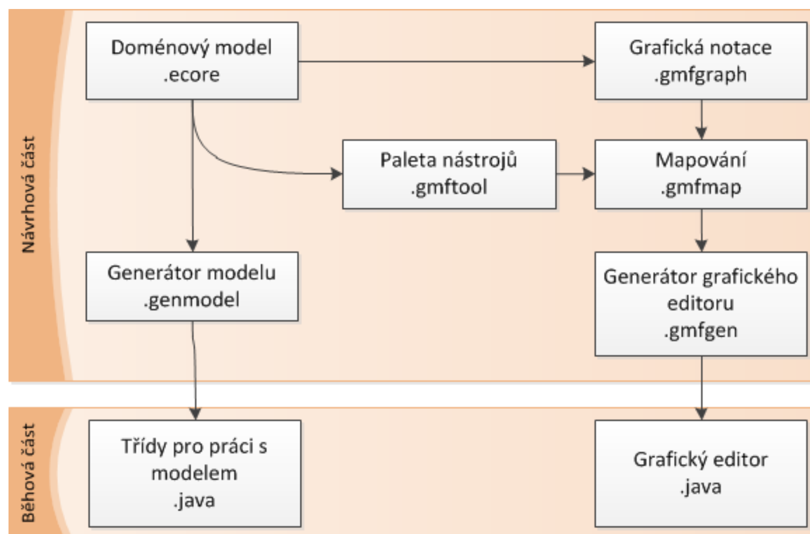
Jednotlivé části, týkající se přímo GMF, jsou popsány níže.

Definice grafické reprezentace

Definuje grafickou reprezentaci. Umožňuje popsat způsob vizualizace elementů doménového modelu. Je možné definovat např. barvu popředí a pozadí, tloušťku čar, šířku okrajů ap. V případě, že by tyto možnosti k popsání reprezentace elementu nestačily, je možné implementovat vlasní implementaci zobrazení.

Dále definuje typy uzlů a spojů, které se mohou v diagramu vyskytnout. Je nutné, aby jeden z elementů doménového modelu reprezentoval plátno diagramu a agregoval všechny

[4] http://www.eclipse.org/gef/
[5] http://www.eclipse.org/modeling/gmp/



Obrázek 3.2: Životní cyklus vývoje grafického editoru v Eclipse

takové elementy, které bude možné do diagramu vložit jako uzly. Uzly i spoje mohou obsahovat libovolné množství nápisů a jsou propojeny s jejich grafickými elementy.

V tomto souboru jsou také popsány *Compartments* – sekce v uzlech, které slouží jako kontejnery pro ostatní uzly.

Paleta nástrojů

Definuje nástroje pro tvorbu nových uzlů a jejich propojení. Nástroje jsou dostupné prostřednictvím palety, kterou lze upravovat, a mohou být kategorizovány do předdefinovaných skupin. Obecně není nutné, aby jeden uzel nebo hrana odpovídaly jednomu nástroji v paletě. Je dokonce možné, aby jeden nástroj v paletě tvořil více různých uzlů nebo propojení – to, který má být vytvořen, zjistí GMF z kontextu.

Mapování

Propojuje doménový metamodel, jeho grafickou reprezentaci a paletu nástrojů. Upravuje, které uzly a spojení reprezentují které třídy v metamodelu, a kterými nástroji budou tvořeny. Upravuje i mapování nápisů, které jsou součástí uzlů nebo náleží k hranám, na atributy elementů v modelu. U nich lze nastavit, zda bude možné je v editoru upravovat.

Konfigurace mapování může definovat různá omezení při vytváření diagramu. Tato omezení mohou být vyhodnocována např. při vytváření uzlů, spojů, nebo na explicitní žádost.

Výrazy, popisující omezení, mohou být popsány v jednom z těchto jazyků:

1. OCL
2. Regulární výraz
3. Negovaný regulární výraz
4. Java

Dělí se do následujících skupin:

1. Audit, reprezentuje skupin pravidel, které musí splňovat instance diagramu.

2. Metrics, numerický výraz, který může omezovat dolní a horní limit vlastnosti.

3. Link Constraints, omezení spojů, které se spouští při tvorbě nového spoje.

Použitím omzení spojů může GMF zabránit spojení elementů, které by nedávalo smysl – při pokusu o jeho vytvoření je nedovolí GMF propojit a tento problém vizuálně indikuje uživateli.

Takto např. vypadá v OCL omezení pro jednoduchou dědičnost elementů, tedy zabraňuje třídě mít více než jednoho rodiče, rozšiřovat sebe, případně tvořit cyklus[11]:

```
if oppositeEnd.oclIsUndefined() then
 self.eSuperTypes->isEmpty()
else
 self <> oppositeEnd and not oppositeEnd.eAllSuperTypes->
  includes(self)
endif
```

Generátor editoru

Z definice mapování je nutné vytvořit konfigurační soubor pro generování editoru. V něm lze také nastavit některé specifické parametry výsledného editoru, jako je asociovaná přípona, použití validačních dekorátorů atd. Výstupem generování je plugin pro prostředí Eclipse.

Kapitola 4

Projekt Lissom

Projekt Lissom[1]je vyvíjen na Fakultě informačních technologií VUT. Je zaměřený na dvě základní oblasti. První je vývoj jazyka pro popis MPSoC a vestavěných systémů (jazyk ISAC). Druhou je generování nástrojů pro vývoj software (kompilátor jazyka C, simulátor atd.) z popisu hardware v jazyce ISAC. Cílem je vytvořit produkt, který integruje všechny potřebné nástroje k souběžnému vytvoření zákaznických obvodů a aplikací na nich provozovaných.

V rámci projektu jsou vyvíjeny čtyři aplikace:

1. **Kompilátor jazyka ISAC** , který transformuje popis hardware do interního modelu ve formátu XML. Zároveň vykonává syntaktickou a sémantickou kontrolu zápisu.

2. **Rekonfigurovatelný kompilátor a dekompilátor jazyka C**, které umožňují kompilaci a ladění programů pro libovolnou vytvořenou architekturu.

3. **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.

Poslední částí je vývojové prostředí, navržené jako třívrstvá aplikace, skládající se z těchto částí:

1. **Middleware**, střední vrstva, která propojuje uživatelské prostředí a vývojové nástroje. Umožňuje vytvářet generické nástroje. Při distribuované simulaci se stará o vzájemnou synchronizace simulátorů.

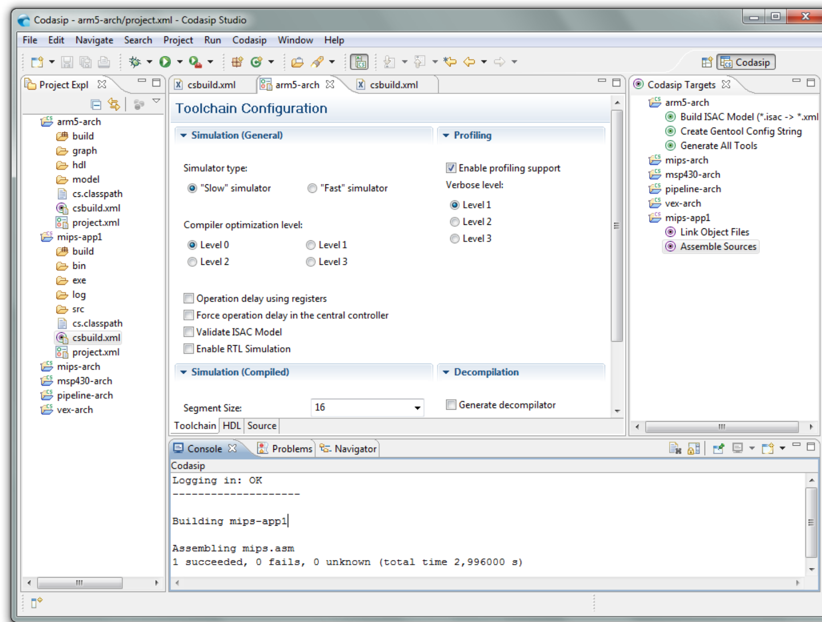
2. **Simulační rozhraní**, která slouží jako nástroj pro správu běžících simulátorů. Poskytují rozhraní pro ladění spuštěných aplikací, které mohou běžet zároveň.

3. **Uživatelské rozhraní**, poskytující vývojáři všechny funkce potřebné pro celý vývoj vestavěného obvodu. Aktuálně jsou dostupné dva typy rozhraní: konzolová aplikaci a IDE postavené na platformě Eclipse.

[1]http://www.fit.vutbr.cz/research/groups/lissom/index.html

4.1 Lissom IDE

Lissom IDE je integrové vývojové prostředí pro nástroje ze stejnojmenného projektu. Jde o RCP (Rich Client Platform) aplikaci vytvořenou nad platformou Eclipse. Protože samotnou komunikaci s nástroji realizuje střední vrsta (Middleware), se kterou IDE komunikuje přes proprietární protokol po TCP/IP, jde o tenký klient. Lissom IDE je aktuálně tvořeno jádrem Eclipse (platformou OSGi R4), 40 vlastními a zhruba 500 přejatými pluginy. Fakticky tak jde o poměrně komplexní aplikaci.



Obrázek 4.1: Vývojové prostředí projektu Lissom

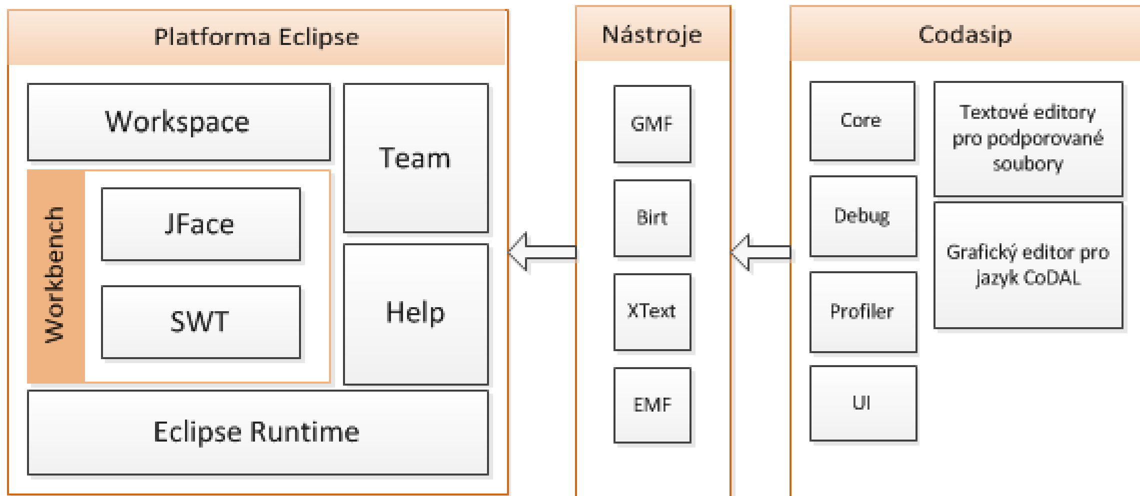
Jejím cílem je poskytnout vývojáři prostředí, integrující všechny potřebné nástroje pro souběžný vývoj, testování a profilování vestavěných zařízení.

V současné době prostředí umožňuje:

1. vytvářet a spravovat projekty
2. editovat zdrojové soubory
3. vytvářet a používat generické nástroje
4. automatizovat úlohy pomocí skriptů
5. simulovat napsané aplikace, včetně podpory krokování a zobrazení zdrojů simulované platformy
6. profilovat aplikace a zjistit tak úzká hrdla vyvíjeného systému

Pro práci se zdrojovými soubory je k dispozici sada editorů. Pro popis hardware je k dispozici komplexní editor jazyka ISAC vytvořený pomocí technologie XText[28]. Podporuje zvýraznění a automatickou kontrolu syntaxe a automatické doplňování. Pro popis platformy slouží i grafický editor jazyka ISAC[24], postavený na stejných rozhraních, jako nově

vytvářený editor pro prototypování. Úpravu souborů jazyka symbolických adres umožňuje jednoduchý editor, podporující lexikální analýzu specifickou pro danou platformu. Posledním editorem je editor konfigurace projektů, který umožňuje intuitivní změny parametrů, zasílaných v příkazech střední vrstvě.



Obrázek 4.2: Moduly vývojového prostředí Lissom IDE

Jednotlivé vrstvy navzájem komunikují přes TCP/IP pomocí vlastního aplikačního protokolu. Toto rozdělení umožňuje na jedné straně centralizované ovládání nástrojů přes jednu instanci middleware pro více vývojářů, a na druhé straně distribuované zpracování úloh, kdy každý ze spuštěných simulátorů může pro lepší škálování výkonu běžet na své vlastní stanici.

Vývojové rozhraní aktuálně pracuje se dvěma typy projeků:

- **HW - Definice platformy**, který slouží pro popis vyvíjené platformy pomocí jazyka ISAC. Zároveň umožňuje generovat specifické nástroje, VHDL reprezentaci. Každý takový projekt je konfigurovatelný, a je tak možné ovlivnit např. sílu optimalizací generovaného překladače jazyka C nebo podporu profilovacích nástrojů.
- **SW - Obslužná aplikace**, která umožňuje vytvoření aplikace, jež poběží na vyvíjené platformě. Aplikaci je možno psát v jazyce symbolických adres, v nejbližší době přibude i podpora pro jazyk C.

Jak je vidět, vývojové prostředí pro Lissom má relativně dobrou podporu pro vývoj vestavěných systémů na nižší úrovni. Systémy jsou ale čím dál více složitější, obsahují více čipů, komunikují s množstvím periferií a senzorů. Zároveň se objevuje potřeba více konfigurací pro jeden typ systému, které by umožnily snazší a rychlejší optimalizaci návrhu systému. Další požadovanou vlastností je možnost snadného znovupoužití IP (Intellectual Property), která by umožnila rychlé protypování vestavěných systémů.

Pro splnění těchto požadavků je potřeba systémová úroveň modelování pro zachycení struktury vestavěného systému jako integritního celku. Tu projekt Lissom aktuálně nepodporuje. Pokud bychom shrnuli předchozí odstavec, měl by modelovací nástroj na této úrovni zejména:

1. Zachytit vazby mezi komunikujícími hardwarovými komponentami

2. Umožnit modelovat součinnost částí obslužné aplikace
3. Nastavit alokaci aplikací na dostupné zdroje
4. Umožnit snadné znovupoužití IP a softwarových komponent

Kapitola 5

Požadavky na vytvářený nástroj

Předchozí kapitola představila projekt Lissom a shrnula současný stav vyvíjených nástrojů, zejména vývojového prostředí. To aktuálně poskytuje díky vhodným editorům možnost efektivního vývoje platformy a přidružených aplikací v jazyce symbolických adres. Nevýhodou současného řešení je však vzájemná provázanost těchto dvou fundamentálních částí. Vazba aplikace na platformu je konfigurovatelná, neumožňuje však snadnou změnu. Lze sice otestovat běh aplikace na více různých platformách, vyžaduje to však překonfigurování projektu a hlavně nutnost vždy přeložit zdrojové soubory aplikace, protože ty jsou uchovávány pouze pro jednu – aktuální – platformu. To zároveň znemožňuje simulovat více aplikací zaráz, což je nutné tehdy, pokud platformu tvoří více čipů a na každém z nich běží vlastní aplikace.

Na druhou stranu oproti původním předpokladům se neukazuje jako nutné zachytit vazby mezi jednotlivými platformami a aplikacemi (ve smyslu vazba *platforma - platforma* a *aplikace - aplikace*), protože tyto jsou zachyceny na úrovni popisu hardware resp. ve zdrojových kódech aplikací a byly by redundantní.

Shrnuto, od praktické části této práce se očekává, že rozšíří vývojové prostředí tak, aby umožňovalo:

- vizuálně modelovat vazby mezi aplikacemi a platformami (vazba *aplikace - platforma*). Je třeba umožnit uživateli definovat více konfigurací alokace a umožnit mu mezi těmito konfiguracemi přepínat.
- zachytit alokace aplikací a simulátorů platforem na dostupné hardwarové zdroje a podpořit tak možnosti 3vrstvého návrhu pro distribuovanou simulaci. Součástí je zároveň převod takových konfigurací do formátu akceptovaného střední vrstvou, která alokace provede.
- uchovávat objektové a spustitelné soubory aplikací pro každou platformu zvlášť, aby nebylo nutné při změně konfigurace provést opětovný překlad.

Zmíněné změny

- přinesou možnost rychle vytvořit a zkoumat více kandidátních řešení
- podpoří tak rychlejší a snadnější tzv. **design space exploration**
- připraví prostředí pro budoucí rozšíření o nabídku znovupoužitelných komponent

Popsané měny by měly být hlavním přínosem praktické části této práce.

Kapitola 6

Analýza požadavků a návrh řešení

Z informací z předešlé kapitoly plynou dva zásadní fakty:

1. Požadavky na schopnosti modelovacího nástroje se omezují na dva aspekty grafického modelování vestavěných systémů – vazby mezi komponentami a alokace na dostupné zdroje. Není vyžadována podpora pro analýzu, syntézu, speficikaci požadavků nebo zachycení chování.

2. Naopak více do hloubky jdou požadavky na těsnou integraci s vývojovým prostředím. Z hlediska uživatele je stěžejní efektivní práce s modelem a okamžité přizpůsobení vývojového prostředí novému modelu. To se týká jak oblasti vývoje, kde jde především o nastavení a výběr správného vygenerovaného nástroje, tak oblasti simulace, kde je třeba sestavit odpovídající konfiguraci pro střední vrstvu a umožnit ladění zvolených aplikací.

Z toho plyne, že více než o modelování samotných architektur a aplikací jde spíše o *modelování vazeb mezi nimi*. V porovnání s existujícími profily, jako je MARTE nebo UML for SoC, jde o více abstraktní model, který nijak přímo nezasahuje do podoby ať už hardware nebo obslužné aplikace. Na druhou stranu je jisté, že požadavky na modelovací nástroj budou v budoucnosti růst směrem ke schopnosti zachytit konkrétní vlastnosti komponent pro vývoj MPSoC a je potřeba dopředu počítat s tím, že model se bude upravovat a rozšiřovat.

V úvahu přicházejí v zásadě dva různé způsoby realizace. První spočívá ve výběru a použití již hotového nástroje pro modelování vestavěných systémů (pokud takový existuje), druhý v implementaci vlastního modelovacího prostředí. Při rozhodování je třeba zvážit následující parametry:

1. Náročnost implementace a/nebo přizpůsobení

2. Dodržení standardů

3. Licence, dostupnost zdrojových kódů

4. Dostupnost dokumentace

První zmíněná varianta se samozřejmě jeví jako výhodnější. Navíc, pokud bychom se rozhodli jít cestou existujících standardů, je to také možnost jediná - specifikace těchto standardů, ať už jde o MARTE, SysML nebo UML for SoC jsou mnohasetstránkové formální dokumenty a vytvoření odpovídajícího modelovacího nástroje by vyžadovalo neúměrně mnoho času. Pro platformu Eclipse jsou v rámci projektu Papyrus, zmíněného dříve,

vyvíjeny profily SysML a MARTE. Jejich použití však v praxi brání to, že tento projekt byl v době psaní práce z velké části přepisován a např. model MARTE není v současnosti ještě pro novou verzi k dispozici. Jiné nekomerční implementace v tuto chvíli zřejmě neexistují, proto se touto cestou aktuálně nelze vydat.

Druhý způsob zahrnuje vytvoření vlastního modelovacího nástroje postaveného nad rozhraním EMF a GMF. Mezi jeho výhody patří menší závislost na produktech třetích stran (a tím menší rizika s tím spojená) a pravděpodobně snazší integrace do existujícího vývojového řešení, což je v tomto případě stěžejní. Nevýhodou samozřejmě je, že půjde o proprietární řešení – historie ukazuje, že ignorovat standardy se nevyplácí. V současné chvíli se však toto řešení jeví jako jediné realizovatelné.

Z hlediska interakce s uživatelem by vývojové přestředí pro splnění požadovaných cílů mělo:

1. umožnit definovat v rámci vývojového prostředí několik různých modelů vazeb aplikací na platformy a alokací na výpočetní zdroje. Tyto modelu budou dále označovány jako *konfigurace*.

2. nabídnout vývojáři možnost, jak mezi konfiguracemi přepínat, a aby jedna z nich byla tzv. aktivní – dle ní by se vývojové prostředí řídilo. Mělo by být vždy patrné, která konfigurace je aktuální.

3. podporovat možnost exportu nastavení uloženého v libovolné konfiguraci do formátu používaného střední vrstvou jako konfigurace simulačního prostředí.

4. automaticky reagovat na změny projektů a zdrojových souborů – tedy buď konfiguraci příslušně upravit nebo naznačit uživateli, že se nenechází v konzistentním stavu (např. při odebrání projektu).

Kapitola 7

Realizace modelovacího nástroje

Kapitola popisuje rozšíření vývojového nástroje projektu Lissom o modelovací část a jeho úpravy tak, aby byly splněny požadavky zadání. Cílem však není nahrazovat manuál nebo referenční příručku k rozhraním Eclipse, ani zatěžovat čtenáře implementačními detaily, proto implementace a integrace popsána spíše na úrovni architektury. Ukázky zdrojových kódů jsou použity pouze za účelem demonstrace zajímavých částí projektu a specifik modelovacích prostředí nebo Lissom IDE.

Realizaci nástroje lze rozdělit na vytvoření doménového modelu a jeho editoru a následně jejich integrace do Lissom IDE. Každé části je věnována příslušná podkapitola.

7.1 Vytvoření modelu a jeho grafického editoru

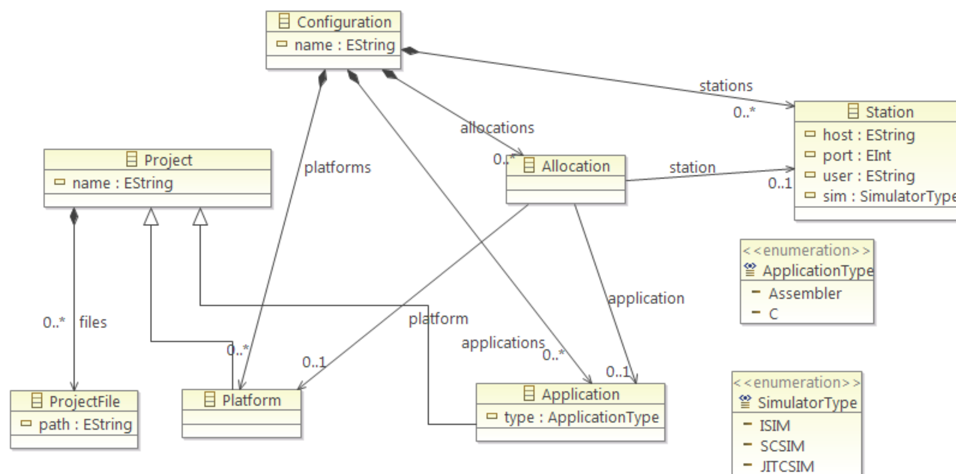
K vytvoření doménového modelu a jeho editačního nástroje byly využity frameworky EMF a GMF. Vývoj vychází z životního cyklu popsaného ve druhé kapitole. Výstupem této části je generovaný doménový model a příslušející textový i grafický editor ve formě sady pluginů pro prostředí Eclipse.

7.1.1 Doménový model

Doménový model vychází z požadavků na zachycení vazeb mezi projekty a jejich alokací na dostupné pracovní stanice. Základem je třída *Configuration*, kořenový prvek modelu. Obsahuje kolekci *projektů*, *stanic* a *alokací*. *Project* reprezentuje fyzický projekt ve vývojovém prostředí. Z něj vychází dvě specializace: *Platform* pro projekt popisující hardwarovou platformu a *Application* pro aplikační projekt. *Station* je definuje přístupný výpočetní zdroj, používaný při simulaci. Obsahuje zároveň parametry, jako je typ simulátoru. *Allocation* je vazba mezi právě jednou platformou, aplikací a žádnou nebo jednou stanicí. Je zde jako objekt nahrazující v EMF neexistující ternární asociaci. Projekty mohou obsahovat soubory (vztah agregace), které budou v editoru sloužit pouze jako odkaz na některé významné části projektu (u projektů pro popis platformy může jít například o soubory ISAC) a nebudou při uložení modelu serializovány. Dále jsou definovány dva výčty: pro typ aplikačního projektu a typ simulátoru.

Ecore metamodel byl vytvořen pomocí grafického editoru Ecore Tools a je znázorněn na obrázku 7.1. Z něj byl nástroji EMF vytvořen vstup pro generátor modelu a posléze samotný model.

Pro představu jak vypadá vygenerovaný model, je vložen následující fragment kódu s metodou *setPlatform* v třídě *AllocationImpl* (slouží tedy k nastavení platformy v alokaci).



Obrázek 7.1: Ecore metamodel

V kódu je vidět, že po ihned po změně reference je provedena notifikace. Její parametry obsahují jak typ změny a o kterou vlastnost jde, tak její původní a novou hodnotu.

```
public void setPlatform(Platform newPlatform) {
  Platform oldPlatform = platform;
  platform = newPlatform;
  if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, Notification.SET,
      ConfigurationPackage.ALLOCATION__PLATFORM,
        oldPlatform, platform));
}
```

Jiná ukázka pro změnu zobrazuje komentář k rozhraní *getFile* v rozhraní *Project*. Ukazuje způsob automatického generování komentáře. Ten je důležitější, než se na první pohled zdá, protože obsahuje anotace definující vlastnosti modelu. Např. v tomto přídadě parametry anotace *@model* určují, že jednak soubory jsou k projektu ve vztahu agregace (namísto výchozí asociace, tedy silnější druh vazby – soubory samy o sobě nemohou existovat) a jednak jsou přechodné, tedy nejsou serializovány. Důležitá je ještě anotace *@generated*, používaná v EMF i GMF. Ta určuje (a upozorňuje), že se jedná o vygenerováný kód, který může být přepsán. Chceme-li tomu zabránit (obvykle, protože jsme výchozí implementaci upravili a nechceme o ni přijít), je nutné doplnit anotaci o parametr *NOT*. Tento problém není třeba řešit u samotného komentáře, který je rozdělen na generovanou a vývojářem upravovanou část.

```
/**
 * Returns the value of the '<em><b>Files</b></em>'
 *              containment reference list.
 * The list contents are of type
 *              {@link configuration.ProjectFile}.
 * <!-- begin-user-doc -->
 * <p>
 * If the meaning of the '<em>Files</em>'
```

```
 *                containment reference list isn't clear,
 * there really should be more of a description here...
 * </p>
 * <!-- end-user-doc -->
 * @return the value of the '<em>Files</em>'
 *                containment reference list.
 * @see configuration.ConfigurationPackage#getProject_Files()
 * @model containment="true" transient="true"
 * @generated
 */
EList<ProjectFile> getFiles();
```

Testování modelu

Zejména po ručních úpravách kódu modelu je vhodné disponovat nástrojem, který umožní jeho testování. Spouštět kvůli testování celé prostředí, které model používá, by bylo velice nepraktické, proto generátor modelu v EMF umožňuje vytvořit automaticky i kostru pro unit a integritní testy pro celý model. Ty je pak možné dávkově spustit a ověřit funkčnost modelu.

7.1.2 Grafická podoba

Druhou částí je vytvoření grafického modelu pomocí prostředí GMF. Postup bude odpovídat životnímu cyklu popsanému obrázkem 3.2, jehož části bude dále stručně popsány.

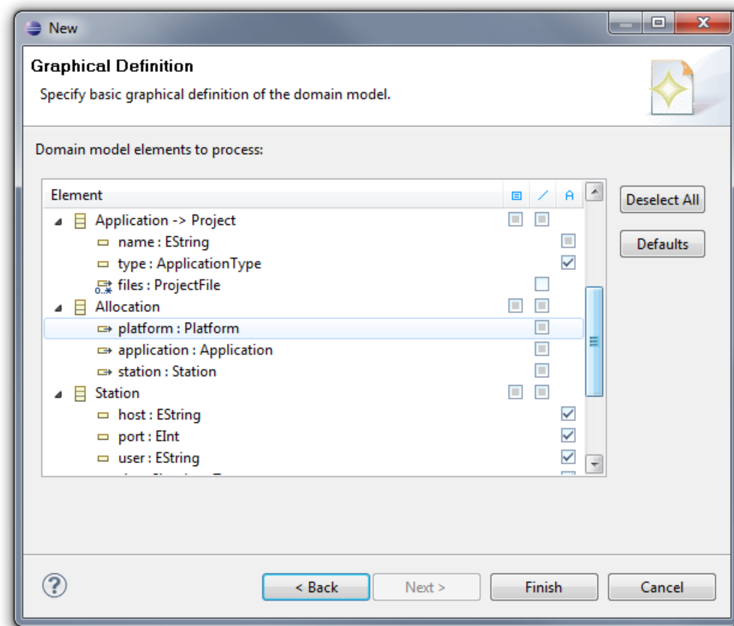
Popis grafické reprezentace

Soubor pro popis grafické reprezentace je možné vytvořit pomocí průvodce a pak pouze upravit, jak ukazuje obrázek 7.2. Průvodce umožňuje snadno nastavit, jestli bude daná část doménového modelu představovat uzel, spoj nebo nápis. Je však podmínkou, aby jedna třída modelu reprezentovalo plátno diagramu. Všechny ostatní třídy, které pak mají reprezentovat uzly a spoje v diagramu musí tato třída agregovat.

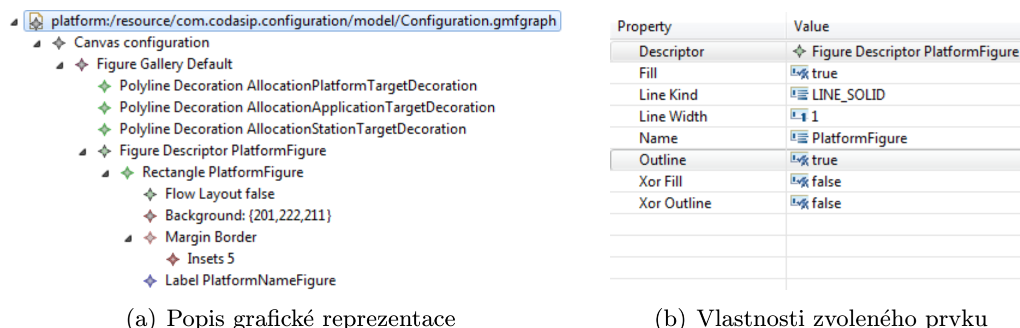
Po vygenerování je třeba soubor upravit dle vlastních požadavků. Skládá se ze dvou částí. První je galerie tvarů (*Figure Gallery*), ta definuje různé grafické podoby, zatím bez přiřazení k jednotlivým uzlům nebo spojům. Na obrázku 7.5(a) je např. vidět způsob vizualizace nazvaný *PlatformFigure*. Je definován jako čtverec s pevně danou barvou pozadí, odstupem okraje 5 px a nápisem, který má opět svůj vlastní způsob zobrazení. Na sousedním obrázku 7.5(b) jsou vidět některé další vlastnosti, například druh a šířka okraje nebo způsob výplně.

Druhou je seznam uzlů (*Node*), spojů (*Connection*), oddělení (*Compartment*) a nápisů (*Diagram Label*). Tyto elementy je následně třeba spojit s některým tvarem z dřív zmíněné galerie.

Pokud chceme některý uzel zobrazit uvnitř jiného, je pro něj potřeba definovat oddělení. V případě implementovaného editoru se to týká seznamu souborů, které se mohou zobrazit v projektu.



Obrázek 7.2: Průvodce tvorbou grafické reprezentace



(a) Popis grafické reprezentace

(b) Vlastnosti zvoleného prvku

Obrázek 7.3: Úprava .gmfgraph

Definice palety nástrojů

Pro generování palety je možné opět použít podobného průvodce. Ten se od průvodce pro generování grafické reprezentace liší v zásadě jen tím, že neobsahuje sloupec pro nápisy, protože ty uživatel v GMF nikdy netvoří sám, vždy jsou součástí některého z uzlů (případně náleží k některému spoji). Pro položky v paletě je možné vytvořit kategorie, které se v editoru projeví jako odlišné seznamy nástrojů. U každého nástroje v paletě je možné definovat jeho název, výchozí ikonu a popis.

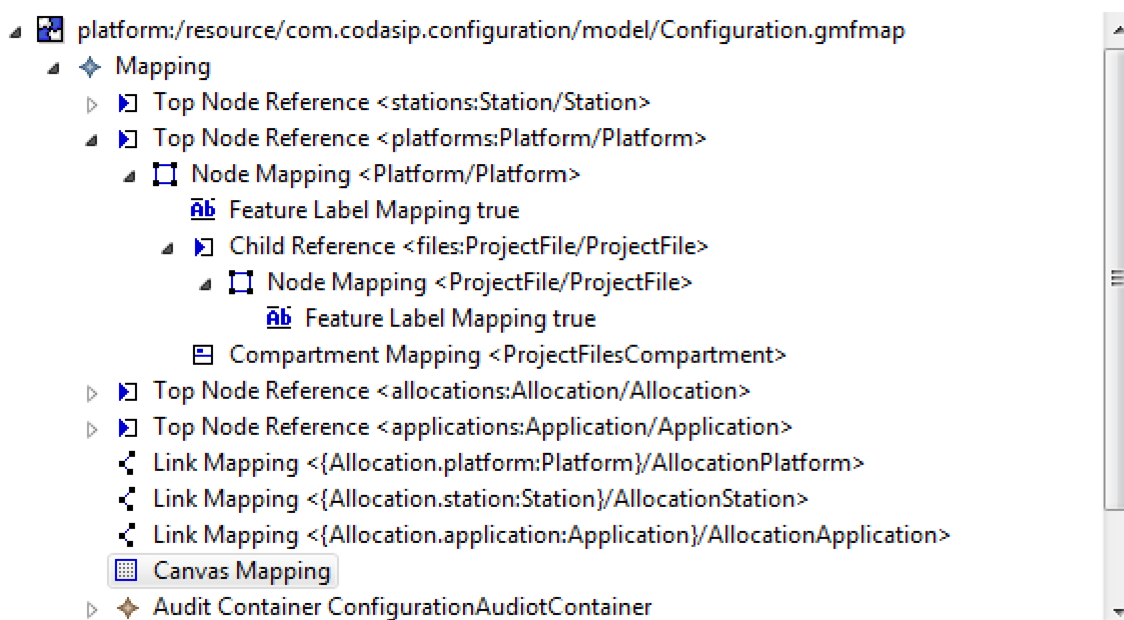
V editoru pro Lissom jsou nástroje rozděleny na dvě skupiny. První je skupina uzlů, do které náleží nástroj pro tvorbu aplikačního projektu, projektu platformy, stanice a alokace. Mezi spoji je jediný nástroj, umožňující asociovat prvky mezi sebou. Tento nástroj bude ve skutečnosti tvořit 3 různé druhy asociací, protože model rozlišuje vazbu podle zdrojových a cílových elementů. Asociace *Allocation - Platform* je tak odlišná od *Allocation - Application* a *Allocation - Station*.

Vytvoření mapování

Nyní je třeba provést mapování metamodelu, grafické podoby a palety nástrojů. K tomu slouží další soubor, *.gmfmap*. Skládá ze opět z několik částí: mapování uzlů, spojů a plátna. Pro každý uzel je třeba nastavit příslušnou třídu z metamodelu, grafickou reprezentaci a nástroj pro jeho tvorbu.

V editoru pro Lissom není záměrně definován nástroj pro *ProjectFile*, protože ten uživatel nebude moci vytvářet. Dále bylo třeba nastavit, aby všechny spoje byly tvořeny jedním nástrojem.

Na přiloženém obrázku 7.6 je vidět mapování oddělení (*Compartment*) pro soubory jako součást platformy.



Obrázek 7.4: Definice mapování

Vygenerování samotného editoru

Pokud máme k dispozici mapování, zbývá vygenerovat konfigurační soubor pro generování samotného editoru. Ten ještě umožňuje nastavit obecné parametry, které nejsou specifické pro konkrétní model nebo editor. Jde např. o příponu asociovanou s tímto editorem v prostředí Eclipse, zapnutí/vypnutí vizuálního upozornění na chyby zjištěné validací nebo zástupce editoru v dialogu pro tvorbu nových souborů.

Po tomto kroku máme základní editor k dispozici.

7.1.3 Validace modelu

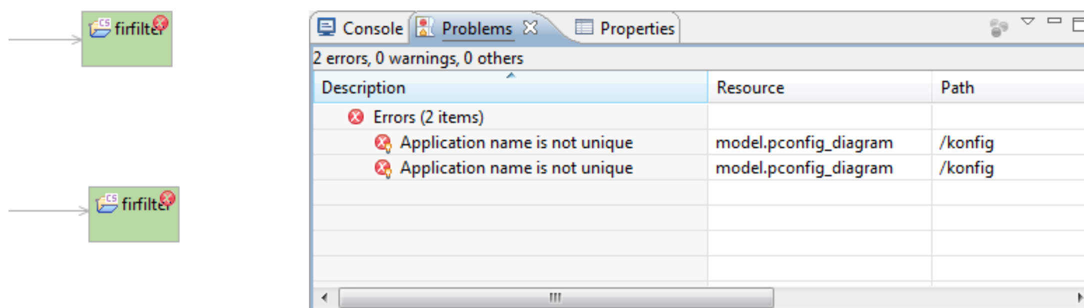
Metamodel modelovacího nástroje pro Lissom je poměrně jednoduchý, takže nebylo nutné specifikovat dodatečná omezení – sám o sobě se model vytvářený grafickým editorem nemůže dostat do nesprávného stavu. Protože však model reprezentuje fyzické projekty, které mohou být např. smazány, je třeba řešit nekonzistenci vůči aktuálnímu pracovnímu prostoru (*workspace*).

První možností je naslouchat změnám v pracovním prostoru a všechny modely s ním jednostranně sychronizovat. To však není vhodné řešení z hlediska uživatelského komfortu – smazáním projektu by např. uživatel mohl přijít o nastavení alokace popsané v modelu, která byla během synchronizace odebrána. Přitom možná chtěl pouze nahradit tento projekt nějakou novější jinou verzí ap.

Druhou možností je nabídnout uživateli možnost validace modelu a v případě, že je nalezen problém, o něm informovat. V editoru mohou nastat v zásadě dva typy problémů:

1. Projekt v modelu fyzicky neexistuje nebo je nepřístupný (např. byl uzavřen).
2. V modelu existuje více projektů se stejným jménem.

Validaci je možné provést jak pomocí nástrojů EMF[10], tak GMF[11]. V editoru pro Lissom byla zvolena druhá varianta. Ta spočívá v rozšíření popisu mapování o novou položku *Audit Container*, které definuje skupinu pravidel, jež musí být v modelu platit. Každému pravidlu je přiřazen jeden element diagramu, který je označen, je-li pravidlo porušeno. Pravidla mohou být popsána v jazycích Java a OCL.



(a) Část diagramu chybného modelu

(b) Zobrazení chyby v Problem View

Obrázek 7.5: Ukázka validace chybného modelu

Takto vypadá pravidlo testující, zda pro aplikační projekty existují odpovídající fyzické projekty v pracovním prostoru. Z uzlu diagramu je zjištěn odpovídají objekt modelu. Jeho kontejnerem musí být v každém případě *Configuration*. Ten obsahuje seznam všech aplikačních projektů, u který už pak lze jednoduše podle jména vyhledat odpovídající projekt pomocí Lissom Resource modelu. Pokud není nalezen nebo nejde o aplikační projekt, je vrácena chyba.

```
/**
 * @generated NOT
 */
public IStatus validate(IValidationContext ctx) {
  Node context = (Node) ctx.getTarget();
  EObject eobj = context.getElement().eContainer();
  if (eobj instanceof Configuration) {
    Configuration config = (Configuration) eobj;

    for (Application app : config.getApplications()) {
      String appName = app.getName();
```

```
      IBaseProject project = CorePluginAPI.getModel().
        getProjectByName(appName);

      if (project == null || !(project
        instanceof ISoftwareProject))
        return ctx.createFailureStatus(app);
    }
  }

  return ctx.createSuccessStatus();
}
```

7.1.4 Úpravy editoru

Ve finálním editoru bylo provedeno několik úprav rozšiřujících a upravujících funkcionalitu editoru. Tři z nich jsou popsány v následující podkapitole.

Rozdělení palety na nové a existující projekty

Editor v daném stavu sice umožňuje vkládat do modelu projekty a vytvářet mezi nimi vazby, práce s ním je ale nekomfortní, protože každou akci je třeba provést dvakrát. Např. chce-li uživatel do vyvíjeného řešení vložit aplikační projekt, musí spustit patřičného průvodce a následně vložit do diagramu uzel reprezentující tento projekt a pojmenovat ho stejně jako fyzický projekt – jedno v jakém pořadí tyto operace provede. Krom toho, že jde o zbytečné zdržení, je zde i riziko překlepu a následné nekonzistence modelu.

Řešení tohoto problému spočívá v rozšíření editoru ve dvou směrech. Zaprvé je do palety nástrojů přidána nová skupina. Ta obsahuje již existující projekty v pracovním prostoru. Pro každý z nich je vytvořen v paletě jeden nástroj. Implementace je řešena nasloucháním na změny v Lissom Resource modelu, který zasílá události týkající se jen Lissom projektů. Je tak zajištěno, že při smazání projektu bude odpovídající volba z palety odebrána.

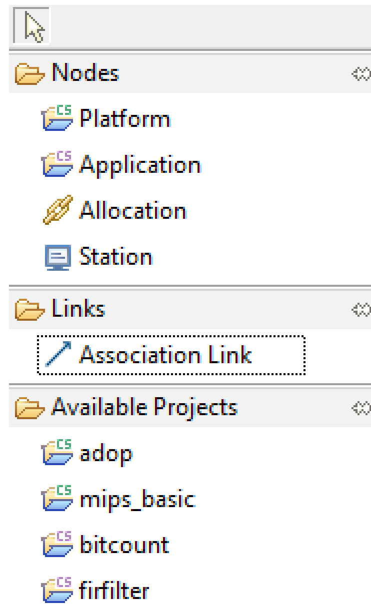
Za druhé, původní nástroje jsou rozšířeny o spuštění průvodce, který vytvoří s novým projektem v modelu konfigurace i projekt fyzický. Pokud uživatel průvodce předčasně ukončí, nevytvoří se projekt ani v modelu konfigurace.

Výsledkem je, že chce-li uživatel rozšířit konfiguraci o nový projekt, je zároveň spuštěn průvodce a projekt je vytvořen i fyzicky. Naopak chce-li vložit existující projekt, stačí jej přetáhnout z palety nástrojů. Důsledkem je, že již nutné ručně měnit jméno projektu v modelu, takže je tato možnost pro snížení pravděpodobnosti, že bude model nekonzistentní s modelem zdrojů, zakázána.

Popisované rozšíření by mohlo v budoucnosti sloužit jako základní stavební kámen podpory znovupoužitelnosti bloků. Paleta by mohla být rozšířena o knihovnu IP bloků, které by se při vložení do modelu nainstalovaly do pracovního prostoru.

Zobrazení ISAC souborů v reálném čase

Editor zobrazuje v uzlech reprezentujících platformy ISAC soubory, které ji popisují. Tato část modelu je vytvářena v reálném čase a neúčastní se persistence. Protože má konfigurační model sloužit jako vrstva abstrakce nad modelem ISACu, nabízí se možnost zobrazit tyto soubory a umožnit tak snadný přechod mezi dvěma modely. Aktuálně je zobrazení pouze



Obrázek 7.6: Detail palety obsahující i skupinu existujících projektů

informativní, tato funkcionalita bude povolena po integraci editoru pro model ISAC, který byl již dříve implementován Ing. Srnou[24]. Bude tak možné přesunout se "dvojklikem" o "jednu vrstvu níže".

Tato funkce je realizována následovně: při startu zásuvného modulu editoru je zaregistrován pozorovatel (*observer* nebo také *listener*), který naslouchá změnám v Eclipse Resource modelu. Pokud v některém projektu přibude nebo je odstraněn soubor ISAC, jsou patřičně upraveny všechny otevřené modely (i když je aktivní pouze jedna konfigurace, otevřených jich může být neomezeně mnoho).

Zobrazení adresy stanice

Základním údajem o stanici (dostupný výpočetní zdroj, na který lze nainstalovat simulátor) je její adresa v síti. GMF ve vygenerovaném editoru vytvoří pro každý atribut třídy vlastní nápis v uzlu. Uživatelsky přívětivější by však bylo zobrazení adresy ve tvaru URL. Proto byly výchozí nápisy odebrány a místo nich byl vytvořen nový, které zobrazuje název stanice ve formátu *user@domain:port*. Element *Station* od svého vzniku sleduje změny v atributech tvořících adresy, a automaticky nápis aktualizuje.

7.2 Integrace editoru do vývojového prostředí

Nyní je třeba hotový model a jeho editor začlenit do existujícího vývojového prostředí. Specifikem oproti jiným editorům v Lissom IDE je to, že tento neslouží pouze k úpravě vlastního souboru, ale vzhledem k požadované funkcionalitě je třeba jej těsně integrovat do existujícího prostředí.

Hlavním cílem implementovaného rozšíření je umožnit vývojáři snadno modelovat vazby mezi různými projekty a zkoumat vlastnosti různých variací systému, kterém budeme zjednodušeně nazývat *konfigurace*, jak bylo zmíněno dříve. Je třeba, aby bylo možné tyto konfi-

gurace mezi sebou přepínat, což zavádí do vývojového prostředí kontext v podobě aktuální konfigurace. Ta určuje propojení komponent v daném čase.

Propojení hraje v pracovním prostoru (*workspace*) důležitou roli:

- Nastavuje, který kompilátor a dekompilátor bude použit při práci se zdrojovými kódy aplikace.
- Určuje podobu simulačních parametrů a typ simulátoru použitý pro danou aplikaci.
- Definuje lexikální analyzátor pro editor jazyka symbolických adres.

Při integraci modelovacího prostředí tedy jde zejména o nahrazení původního způsobu definování vazeb mezi projekty a zavedení kontextu aktuální konfigurace. To ovlivní zejména následujících pět částí Lissom IDE:

- **Správa projektů**: je třeba vytvořit nový projekt, který zapouzdří konfiguraci a poskytne prostor pro ukládání výsledků kompilace, což je další z požadavků na modelovací nástroj – přeložené aplikace nesmějí být přepsány při přepnutí konfigurace.
- **Lissom Resource model**: jde o abstraktní vrstvu nad Eclipse Resource modelem, jež poskytuje informace o projektech, jejich souborech a konfiguracích v pracovním prostoru. Jeho význam je popsán v samostatné podkapitole.
- **Vývojová perspektiva**: Zavedení kontextu si žádá rozšíření perspektivy o nástroj pro přepínání aktivní konfigurace. Taktéž je třeba, aby se kontext projevil v podobě filtrování projektů v Target View, aby nebylo možné pracovat s projekty mimo aktuální kontext.
- **Kompilační skripty**: Skripty je třeba upravit s ohledem na nový způsob ukládání výsledků kompilace.
- **Simulační perspektiva**: Pro konfiguraci simulace lze získat všechny potřebné informace z modelu, není tedy třeba, aby je uživatel musel zadávat ještě jednou.

Nevýhodou popisovaných úprav je, že prostředí si nezachová zpětnou kompatibilitu s předešlými verzemi. Řešením by mohlo být nabídnout uživateli možnost konverze starého projektu na novější verzi.

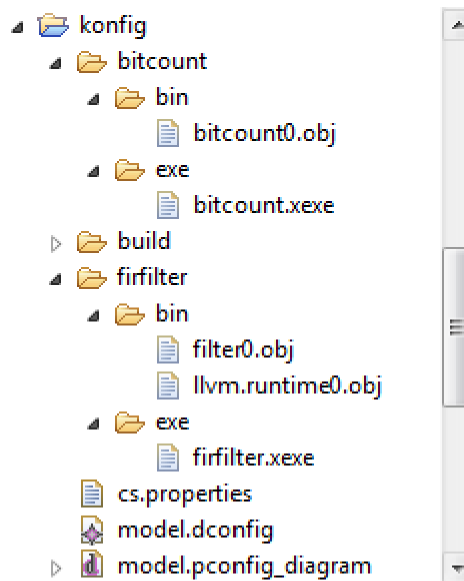
7.2.1 Propojení s jednotlivými částmi prostředí

Tato kapitola podrobněji rozebírá nutné úpravy Lissom IDE pro integraci modelovacího nástroje.

Nový typ projektu

Součástí změn v Lissom IDE je vytvoření nového typu projektu vedle stávajících dvou, sloužících pro popis platformy a tvorbu softwarové aplikace. Tento nový typ, nazývaný konfigurační projekt, slouží jednak k uchování právě jedné instance konfiguračního modelu a jednak pro uložení objektových a spustitelných souborů, které vznikly jako výsledek kompilace aplikací v dané konfiguraci. Tím je zajištěno, že při přepnutí na jinou konfiguraci nebudou tyto soubory přepsány.

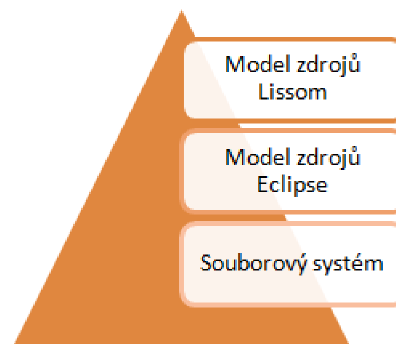
Pro uložení souborů aplikace je v projektu vytvořena struktura podprojektů v podobě složek, odpovídajících jednotlivým aplikacím.



Obrázek 7.7: Detail struktury nového typu projektu.

Model zdrojů

Lissom Resource model je ústředním prvkem vývojového prostředí, využívaný téměř všemi zásuvnými modely. Jde o abstrakci nad modelem zdrojů v Eclipse, který je opět abstrakcí nad fyzickou podobou dat, obvykle souborovým systémem operačního systému, jak je vidět na obr. 7.8. Model zdrojů Lissom zpracovává události zasílané modelem v Eclipse.



Obrázek 7.8: Hierarchie modelů zdrojů

Plní několik rolí:

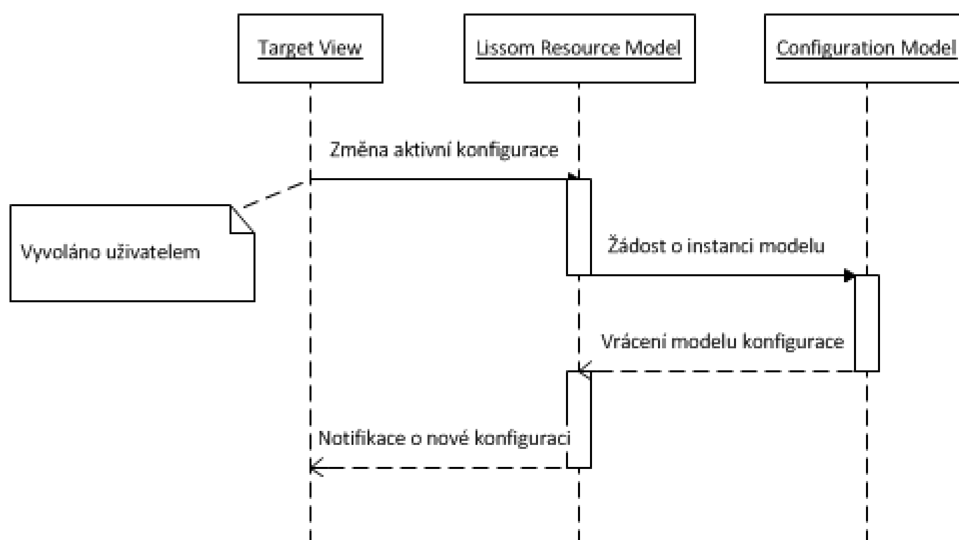
1. Informuje o změnách v projektech Lissom
2. Automaticky aktualizuje konfigurace projektů v paměti při změnách příslušných souborů
3. Poskytuje aktuální strukturu projektů včetně vazeb mezi projekty
4. Slouží k vytváření projektů

5. Umožňuje vyvářet tzv. *markery* – metainformace zdrojů (souborů, složek ap.)

Novým úkolem modelu zdrojů je udržovat informaci o aktuální konfiguraci a při její změně aktualizovat vazby mezi projekty a informovat naslouchající objekty. Žádost o přepnutí konfigurace je zasílána z okna Target View, kde je umístěn UI prvek pro nastavení volby aktuální konfigurace, ale obecně ji mohou vyvolat i jiné objekty.

Zpráva musí obsahovat referenci na nový konfigurační projekt. Model zdrojů nastaví konfigurační model tohoto projektu jako aktivní a zahájí rekonfiguraci. Ta zahrnuje načtení modelu konfigurace do paměti a jeho sychronizace s modelem zdrojů – pro každý aplikační projekt je nastavena odpovídající platforma a tato informace je uložena do konfiguračního souboru každého projektu. Po dokončení operace je provedena notifikace objektů, naslouchajících změnám v modelu zdrojů.

Průběh komunikace je zachycen na následujícím stavovém diagramu 7.10. Protože se během této operace může nacházet model zdrojů v nekonzistentním stavu, je zakázáno provádění skriptů.



Obrázek 7.9: Sekvenční diagram volby konfigurace

Model zdrojů Lissom používá i vytvořený grafický editor. Ten pomocí něj mj. zjišťuje existující projekty Lissom a vkládá je do palety nástrojů. Dojde-li ke změně, je naslouchající objekt upozorněn zprávou, obsahující objekt, který byl změněn a typ změny.

Např. přepnutí aktivní konfigurace vypadá v modelu takto:

```
@Override
public void setActiveConfigurationProject(
  IConfigurationProject activeConfigurationProject) {
  this.activeConfigurationProject = activeConfigurationProject;

  reconcileModel();

  addEventDelta(new CodasipElementDelta(this,
    ICodasipElementDelta.ACTIVE_CONFIGURATION_CHANGED));
  fireAllQueuedEvents();
}
```

Vývojová perspektiva

Perspektivou se v prostředí Eclipse rozumí kontext nebo způsob pohledu na danou úlohu. Z pohledu uživatele jde zejména o rozvržení uživatelského prostředí a vhodné reakce na události specifické pro daný problém. Např. perspektiva pro ladění tak nabízí okno zobrazující obsah zásobníku. Při poklepání na některou z položek zásobníku se otevře odpovídající zdrojový kód. Vývojová perspektiva prostředí Lissom, která je součástí modulu Lissom UI, nabízí tyto pohledy:

1. Project Explorer, který zobrazuje strukturu projektů
2. Okno editoru
3. Console view, které slouží jako vstup a výstup spuštěných skriptů
4. Problem view, zobrazující seznam nalezených problémů (např. chyby při kompilaci)
5. Target view, zobrazující systémové a uživatelem definované kompilační skripty.

Ve vývojové perspektivě byly provedeny dvě změny:

1. Bylo zaintegrováno okno Properties view, jež při otevřeném grafickém editoru umožní uživateli měnit atributy modelovaných elementů
2. Byla provedena reimplementace Target view.

Okno Target view je obdobou podobného okna z perspektivy pro C++ (Makefile) nebo Javu (Ant). Umožňuje uživateli spustit skript pro daný projekt. Bylo rozšířeno o seznam dostupných konfigurací, mezi kterými lze přepínat. Při přepnutí dojde zaslání žádosti Lissom Resource modelu o překonfigurování.

Poté, co okno obdrží od modelu zprávu o změně konfigurace, provede filtrování seznamu projektů tak, aby bylo možné pracovat jen s těmi v aktuální konfiguraci. Tato změna může být iniciována nejen přepnutím konfigurace, ale i změnou v modelu konfigurace aktuální.

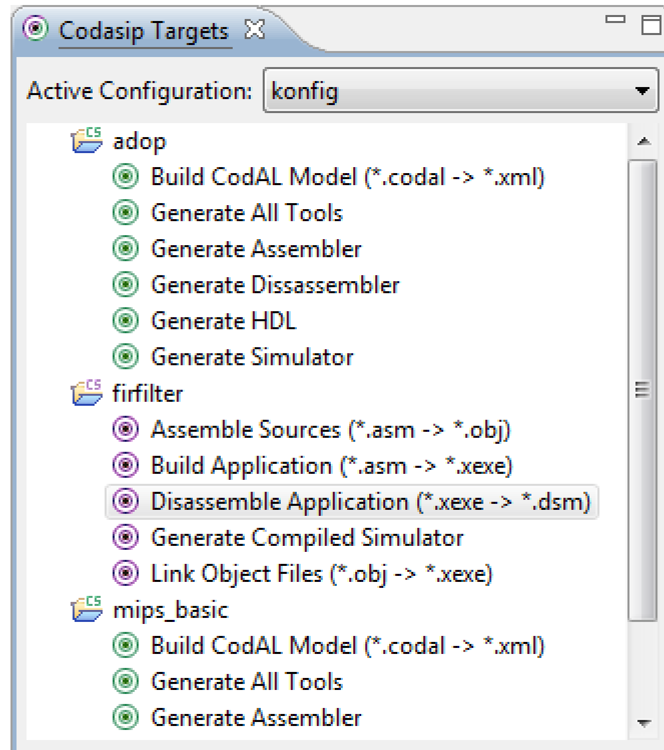
Kompilační skripty

V původní verzi vývojového prostředí se výsledek kompilace zdrojových kódů (tedy objektové a spustitelné soubory) ukládal do patřičných složek v aplikačním projektu. Toto řešení se neukazuje jako praktické, pokud chceme uživateli umožnit vyvíjet aplikaci pro více platforem. Jeho neblahým důsledkem je nutnost rekompilace aplikace po změně platformy, pro kterou má být aplikace vyvíjena.

Řešením je ukládat tyto soubory namísto do aplikačního projektu do projektu konfiguračního. To vyžaduje jednak patřičné úpravy struktury projektů a jednak úpravu té části vývojového prostředí, která zabezpečuje zpracování uživatelem zadaných příkazů, jako je přeložení popisu hardware nebo kompilace zdrojových kódů aplikace. Pro spuštění každého takového příkazu je v prostředí dostupný tzv. *target*. Jde o obdobu cílů ze souborů Makefile. Protože podoba příkazů se v praxi často mění a zároveň je třeba nabídnout uživateli možnost vlastních příkazů, jsou tyto realizovány prostřednictvím skriptů v jazyce Groovy. Integrace modelovacího prostředí tak vyžaduje úpravu zhruba desítky z nich.

Skripty intenzivně využívají aktualizovaný model zdrojů z modulu Lissom Core, díky tomu jsou úpravy snadné.

Takto např. vypadá část skriptu pro slinkování objektových souborů:



Obrázek 7.10: Detail okna Target View, zobrazující aktivní konfiguraci

1. Získání aktivního konfiguračního projektu:

```
def activeProj = model.getActiveConfigurationProject();
```

2. Získání složky, odpovídající aplikačnímu projektu:

```
def subProjFolder = activeProj.getResource().
  getFolder(prj.getName());
```

3. Získání složky, obsahující objektové soubory

```
def binFolder = subProjFolder.getFolder("bin");
```

4. Vyhledání objektových souborů

```
def objFiles = activeProj.getFiles(binFolder, true,
  ICodasipFile.OBJECT_FILE);
```

Simulační perspektiva

Vytvářený model zachycuje i alokace projektů na dostupné výpočetní zdroje, aby bylo možné jej využít pro konfiguraci simulace. Ta probíhá před jejím vlastním zahájením a spočívá v předání parametrů ve formě XML souboru střední vrstvě. Tento soubor obsahuje umístění zdrojů v síti, volné komunikační porty a typy simulátorů, které na ně budou nainstalovány. Všechny tyto informace jsou v modelu dostupné.

Namísto původního způsobu nastavení, kde uživatel musel tyto údaje sám dodat, je možné asociovat simulaci s jednou instancí modelu (konfigurací). Ten bude před zahájením simulace zpracován a konvertován do specifikovaného formátu.

Ukázka diagramu konvertovaného do podoby konfigurace simulace je zobrazena v příloze.

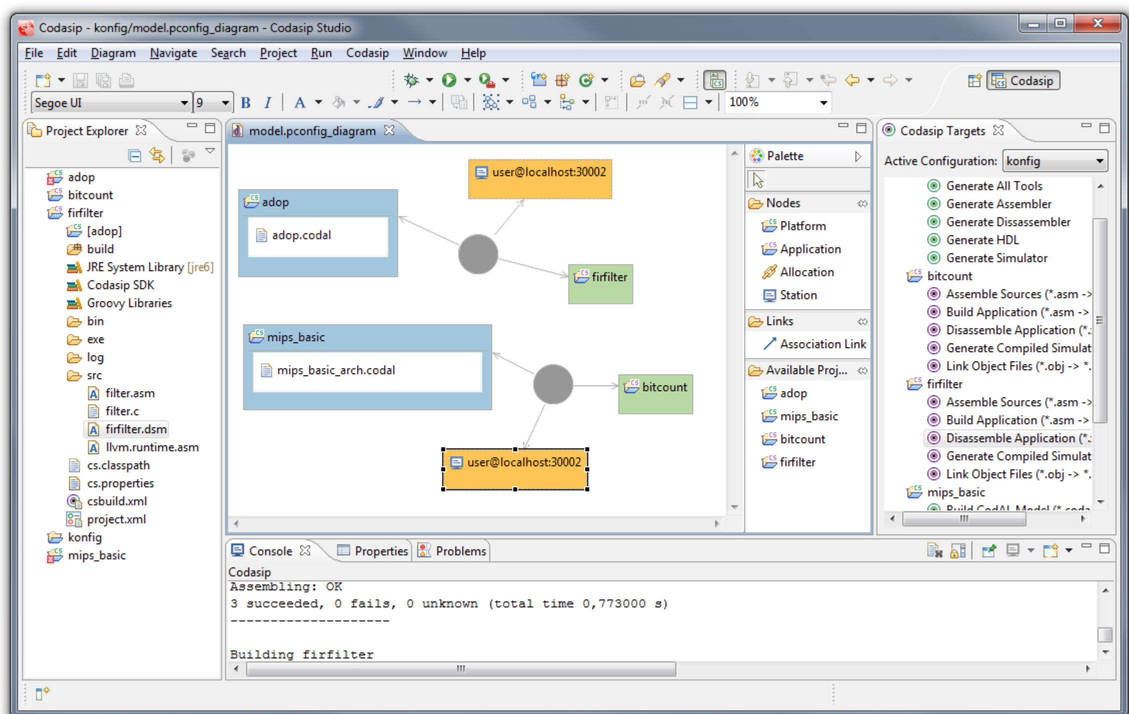
Kapitola 8

Výsledné řešení

Kapitola popisuje výsledné řešení a hodnotí jej vzhledem k zadání diplomové práce. Naznačuje možnosti dalších rozšíření a budoucí směr vývoje.

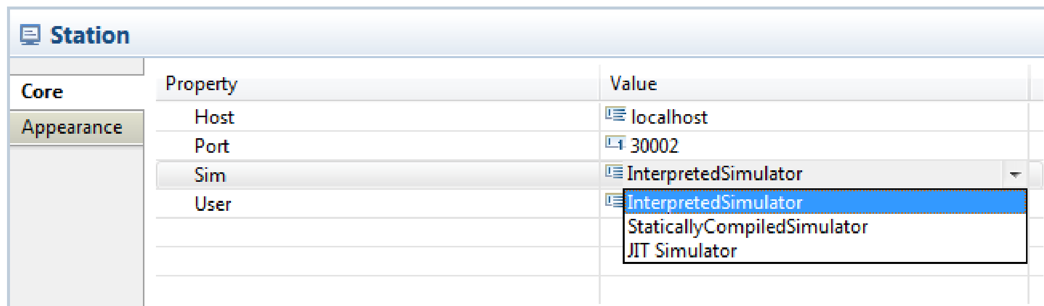
Práce může sloužit jako úvod do problematiky modelování vestavěných systémů pomocí jazyka UML a příslušných dialektů. Zároveň popisuje modelovací rozhraní EMF a GMF a ukazuje základní princip tvorby doménového metamodelu a příslušného grafického editoru pro platformu Eclipse.

Praktická část práce spočívala v implementaci nástroje ve formě sady zásuvných modulů a provedení potřebných úprav v existujícím vývojovém prostředí projektu Lissom. Tyto úpravy, které budou součástí nadcházející verze Lissom IDE, přinášejí vývojáři možnost modelovat, zkoumat a porovnávat různé variace jednoho systému, díky čemuž bude moci rychleji rozhodnout, která konfigurace je pro řešení daného problému nejvýhodnější.



Obrázek 8.1: Výsledná podoba modelovacího nástroje

Vytvořené modelovací prostředí je úzce integrováno do vývojového prostředí Lissom. Pro konfigurace vazeb mezi aplikacemi a platformami byl vytvořen nový typ projektu. Takových projektů může být libovolně mnoho a uživatel má možnost mezi nimi přepínat. Při práci s modelem je ověřována konzistence s aplikačními projekty a projekty popisujícími platformy.



Obrázek 8.2: Editace atributů u třídy *Station*

Součástí modelu je i popis alokací na výpočetní prostředky určený k simulaci vyvíjeného systému. Uživatel tak už nemusí ručně upravovat nastavení simulace – při jejím zahájení je model automaticky konvertován do potřebného konfiguračního formátu a odeslán střední vrstvě. Parametry simulace je možné intuitivně nastavit v grafickém editoru.

Velkou výhodou řešení do budoucnosti je použití rozhraní EMF. To umožňuje aplikovat principy vývoje řízeného modelem a tak používat pro všechny části vývojového prostředí jedinou implementaci modelu – jak pro model v paměti, tak jeho vizualizaci a persistenci. Při změnách modelu tak stačí upravit jen minimum kódu.

Příslibem do budoucna je framework GMF, umožňující spojit výhody EMF a GEF do jednoho. Citelným nedostatek je však zásadní nedostatek dokumentace. Mnoho informací je tak nezbytné hledat na fórech a blozích vývojářů.

Další směry vývoje implementovaného nástroje naznačuje následující podkapitola.

8.1 Možná rozšíření

Výsledný editor, stejně jako další provedené úpravy vývojového prostředí, umožňují v budoucnosti relativně snadná rozšíření. Ta se mohou týkat jak uživatelského prostředí, tak (a to zejména) vlastního doménového modelu.

8.1.1 Interní vazby modelovaného systému

Lze očekávat, že se v budoucnosti objeví jako vhodné rozšířit prostředí o modelování interních vazeb mezi částmi systému, resp. příslušných aplikací. To má však význam teprve tehdy, budou-li tyto změny automaticky promítnuty do zdrojového popisu platformy nebo aplikace (a naopak, bude-li model schopen na tyto změny reagovat).

Modelovací nástroj by tak na základě vytvořeného diagramu vygeneroval část popisu platformy v jazyce ISAC nebo připravil kostru aplikace.

8.1.2 Podpora více aplikací pro jednu platformu

Z pohledu modelovacího nástroje jde o snadnou úpravu modelu, která by však umožnila zásadní rozšíření schopností celého vývojového prostředí, a to spouštět na jednom procesoru

více aplikací – typicky jde o operační systém, podpůrné ovladače a vlastní aplikaci. Tato funkcionalita bude přidána po dokončení podpory na straně nástrojů a střední vrstvy.

8.1.3 Knihovna znovupoužitelných bloků v paletě nástrojů

Znovupoužitelnost se stává klíčovým parametrem určujícím rychlost vývoje hardware. Důležitou roli zde hraje standard IP-XACT. Jedním z možných rozšíření by mohla být integrace tohoto standardu a rozšíření modelovacího nástroje o paletu znovupoužitelných IP bloků.

8.1.4 Interní vazby modelovaného systému

Možným a užitečným rozšířením je nastavení parametrů projektů pro každou konfiguraci zvlášť. To by umožnilo snadno zkoumat právě vliv těchto parametrů, jako je optimalizace kompilátoru. Bylo by tak možné spustit aplikaci v režimu umožňujícím ladění a profilování a ihned poté zkusit "produkční" nastavení. Toto rozšíření by však vyžadovalo součinost se střední vrstvou, protože ta aktuálně uchovává pouze jednu variantu generovaných nástrojů.

8.1.5 Validace v reálném čase

Další úprava by mohla směřovat validaci modelu v reálném čase. Nyní probíhá dávkově, např. při uložení modelu.

Kapitola 9

Závěr

Práce se zabývá tvorbou nástroje pro modelování vestavěných systémů. Cílem první kapitoly je především sumarizace současné situace na poli vývoje vestavěných systémů za pomoci grafických nástrojů. Byl popsán jazyk UML jako nástroj pro aplikaci vývoje řízeného modelem. Zároveň byly popsány jeho dialekty používané v oblasti vestavěných systémů a existující nástroje na tomto poli. Druhá kapitola se věnuje podpoře modelování a vizualizace na platformě Eclipse, nad kterou je postaveno vývojové prostředí projektu Lissom. Následující část seznamuje s tímto projektem a jeho nástroji pro HW/SW co-design. Je popsána architektura a aktuální stav vývojového prostředí a jsou naznačeny požadavky na další vývoj. Ty jsou podrobněji rozpracovány v šesté kapitole. Sedmá kapitola si klade za cíl provést čtenáře realizací praktické části této práce. Závěrečná kapitola seznamuje s výsledným řešením a navrhuje možná rozšíření.

Teoretická část práce se opírá zejména o články a sborníky konferencí, věnující se této problematice, a specifikace příslušných standardů. Druhá část vychází z dostupných publikací o platformě Eclipse a modelovacích nástrojích EMF a GMF.

Výsledkem praktické části je implementace grafického modelovacího nástroje pro prototypování a rychlejší vývoj vestavěných aplikací a jeho integrace do existujícího vývojového prostředí. Přínos pro projekt Lissom spočívá ve zjednodušení modelování vztahů mezi jednotlivými komponentami, tvořícími integritní celek, a možnosti vytvořit a zkoumat více variant systému. To vývojář využije jak při vývoji systému, tak jeho simulaci a ladění.

Výhodou popisovaného řešení je dobrá a relativně snadná možnost dalšího rozvoje a údržby realizovaného nástroje, zejména díky použití modelovacích prostředí EMF a GMF, aplikujících principy MDD.

Literatura

[1] *Eclipse development using the graphical editing framework and the eclipse modeling framework*. Riverton, NJ, USA: IBM Corp., 2004, ISBN 0738453161.

[2] Alhir, S. S.: *Guide to Applying the UML*. Springer, první vydání, 2002.

[3] Association, S. I.: International Technology Roadmap for Semiconductors: 1999 edition. [online], cit [2011-01-09].
URL ftp://ftp.cordis.europa.eu/pub/ist/docs/ka4/pdesign_gap.pdf

[4] Aulagnier, D.; Koudri, A.; Lecomte, S.; aj.: SoC/SoPC development using MDD and MARTE profile. *Model Driven Engineering for Distributed Real-time Embedded Systems*, 2009.
URL http://hal.inria.fr/inria-00468650/en/

[5] Beneš, I. M.: *Přehled OO metodik a notací*, diplomová práce. 2008.

[6] Boutekkouk, F.; Benmohammed, M.; Bilavarn, S.; aj.: UML2.0 Profiles for Embedded Systems and Systems On a Chip (SOCs). *Journal of Object Technology*, ročník 8, č. 1, Leden 2009: s. 135–157, ISSN 1660-1769, doi:10.5381/jot.2009.8.1.a1.
URL http://www.jot.fm/contents/issue_2009_01/article1.html

[7] Boutekkouk, F.; Benmohammed, M.; Bilavarn, S.; aj.: UML2.0 Profiles for Embedded Systems and Systems On a Chip (SOCs). *Journal of Object Technology*, ročník 8, č. 1, 2009: s. 135–157.

[8] Design, C.: Model and Simulate a Virtual System (Application + Platform). [online], cit [2011-02-12].
URL http://www.cofluentdesign.com/index.php/en_US/Products_Services/cofluent-studio/system-architecting.html

[9] EclipseFoundation: Eclipse Modeling Project. [online], cit [2011-04-15].
URL http://www.eclipse.org/modeling/

[10] EclipseFoundation: EMF Validation Overview. [online], cit [2011-03-02].
URL http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.Validation.html

[11] EclipseFoundation: GMF Constraints. [online], cit [2011-03-20].
URL http://wiki.eclipse.org/GMFConstraints

[12] IEEE: IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. *IEEE Std 1685-2009*, 18 2010: s. C1 –360, doi:10.1109/IEEESTD.2010.5417309.

[13] Indrusiak, L.: A pragmatic perspective on UML for system-on-chip design. In *NORCHIP Conference, 2005. 23rd*, 2005, s. 169 – 171, doi:10.1109/NORCHP.2005.1597016.

[14] Oishi, Q. Z. R.; Nakata, T.: Integrating UML into SoC design process. In *In: Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE*, 2005, s. 836–837.

[15] OMG: Model Driven Architecture. [online], cit [2011-01-09].
URL http://www.omg.org/mda/

[16] OMG: MOF 2 XMI Mapping. [online], cit [2011-03-25].
URL http://www.omg.org/spec/XMI/2.4/Beta2/PDF/

[17] OMG: OMG Systems Modeling Language. [online], cit [2011-01-09].
URL http://www.omg.org/spec/SysML/1.2/

[18] OMG: OMG Unified Modeling LanguageTM (OMG UML), Infrastructure, Version 2.3. [online], cit [2011-01-09].
URL http://www.omg.org/spec/UML/2.3/

[19] OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. [online], cit [2011-01-09].
URL http://www.omg.org/spec/MARTE/1.0/PDF

[20] OMG: UML Profile for System on a Chip (SoC). [online], cit [2011-01-09].
URL http://www.omg.org/cgi-bin/doc?formal/06-08-01

[21] OSGiAlliance: OSGi Technology. [online], cit [2011-04-12].
URL http://www.osgi.org/About/Technology

[22] Poole, J. D.: Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[23] Riccobene, E.; Scandurra, P.; Rosti, A.; aj.: A UML 2.0 profile for SystemC: toward high-level SoC design. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-091-4, s. 138–141, doi:http://doi.acm.org/10.1145/1086228.1086254.
URL http://doi.acm.org/10.1145/1086228.1086254

[24] Srna, I. P.: *Nástroj pro návrh čipu v UML*, diplomová práce, Brno, FIT VUT v Brně. 2010.

[25] Steinberg, D.; Budinsky, F.; Paternostro, M.; aj.: *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, druhé vydání, 2009, ISBN 0321331885.

[26] Vanderperren, Y.; Mueller, W.; Dehaene, W.: UML for electronic systems design: a comprehensive overview. *Design Automation for Embedded Systems*, ročník 12, 2008: s. 261–292, ISSN 0929-5585, 10.1007/s10617-008-9028-9.
URL http://dx.doi.org/10.1007/s10617-008-9028-9

[27] Wikipedia: System C. [online], cit [2011-01-09].
URL http://en.wikipedia.org/wiki/SystemC

[28] Šuška, I. B.: *Syntaxí řízený editor*, diplomová práce, Brno, FIT VUT v Brně. 2009.

Příloha A

Použité zkratky

- **UML** Unified Modelling Language
- **MARTE** Modeling and Analysis for Real-Time and Embedded systems
- **SysML** Systems Modeling Language
- **MDA** Model Driven Architecture
- **MDD** Model Driven Development
- **OMG** Object Management Group
- **NFP** Non-Functional Properties
- **AADL** Architecture Analysis and Design Language
- **SOC** System-on-a-chip
- **MPSOC** Multiprocessor System-on-Chip
- **RTL** Register Tranfer Level
- **TLM** Transactional Level Modeling
- **DSL** Domain Specific Language
- **EMF** Eclipse Modeling Framework
- **GMF** Graphical Modeling Framework
- **GMP** Graphical Modeling Project
- **GEF** Graphical Editing Framework)
- **MVC** Model-view-controller
- **IP** Intellectual Property Core
- **XMI** XML Metadata Interchange
- **MOF** Meta-Object Facility

- **OCL** Object Constraint Language
- **SWT** The Standard Widget Toolkit
- **RCP** Rich Client Platform
- **JET** Java Emitter Templates

Příloha B

Obsah CD

Přiložené CD má následující obsah:

- **doc**: technická zpráva
- **src**: zdrojové kódy zásuvných modulů Lissom IDE, které byly během této práce upraveny
- **bin**: Lissom IDE v binární formě, ve variantách pro OS Windows a Linux a 32/64 bitů

Příloha C

Návod na instalaci

Lissom IDE je na doprovodném CD dodáno v binární formě. Pro spuštění je třeba mít:

1. Java Runtime Environment v aktuální verzi.
2. Licenční server

Aplikace lze provozovat na těchto platformách.

- MS Windows od verze 2000, 32/64 bitů
- Linux, 32/64 bitů

Před spuštěním je vhodné nastavit parametry pro alokování paměti v souboru *lissom-studio.ini*. Ty jsou ve výchozím stavu na nízkých hodnotách, aby bylo možné aplikaci spustit na co největším spektru počítačů, což však může komplikovat běh aplikace.

Pro 64bitové Windows 7 se jako rozumné jeví následující hodnoty:

```
--launcher.XXMaxPermSize
1024M
--launcher.XXMaxPermSize
1024m
-vmargs
-Xms40m
-Xmx1536m
```

Pro práci s Lissom IDE je třeba mít přístup k Middleware serveru, který není součástí doprovodného CD. Pro získání licenčního serveru a případného přístupu ke střední vrstvě kontaktujte autora.

Příloha D

Formáty souborů

Tato kapitola popisuje formáty textových souborů, se kterými modelovací prostředí přímo nebo nepřímo pracuje.

D.1 Formát modelu konfigurace

Serialovaný model je uložen ve formátu XMI. Např. model, který je prezentován v kapitole Výsledné řešení, má následující podobu:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration:Configuration xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:configuration="http://configuration/1.0">
  <platforms name="adop"/>
  <platforms name="mips_basic"/>
  <allocations platform="//@platforms.0"
   application="//@applications.0" station="//@stations.1"/>
  <allocations platform="//@platforms.1"
   application="//@applications.1" station="//@stations.0"/>
  <stations/>
  <stations/>
  <applications name="firfilter"/>
  <applications name="bitcount"/>
</configuration:Configuration>
```

D.2 Formát interní konfigurace projektu

Jde o soubor typu .properties, známý z prostředí jazyka Java. Slouží k interní konfiguraci projektu. Lissom Resource model do tohoto souboru nastavuje např. platformu asociovanou s danou aplikací, kterou zjistí z modelu konfigurace.

```
#Mon May 23 15:09:45 CEST 2011
codasip.file.mainmodel=mips_basic_arch.codal
codasip.path.build=build
codasip.path.grammar=grammar
codasip.path.graph=model\\graph
```

```
codasip.path.hdlgen=hdl\\gen
codasip.path.hdlmap=hdl\\map
codasip.path.model=model
codasip.path.package=build\\scripts
codasip.sw.prj.hwprj=mips_basic
eclipse.preferences.version=1
```

D.3 Formát simulační konfigurace projektu

Jde o XML formát dle interní specifikace. Slouží k nastavení parametrů simulace. Následující ukázka opět odpovídá zvolené konfiguraci z obrázku v kapitole Výsledné řešení.

```
<SIMULATION>
   <SIMCNF>
      <PROJECT>adop</PROJECT>
      <HOST>localhost</HOST>
      <PORT>30002</PORT>
      <APP>firfilter.xexe</APP>
      <FREQ>0</FREQ>
      <TYPE>ISIM</TYPE>
      <OPTFILE></OPTFILE>
   </SIMCNF>
   <SIMCNF>
      <PROJECT>mips_basic</PROJECT>
      <HOST>localhost</HOST>
      <PORT>30002</PORT>
      <APP>bitcount.xexe</APP>
      <FREQ>0</FREQ>
      <TYPE>ISIM</TYPE>
      <OPTFILE></OPTFILE>
   </SIMCNF>
</SIMULATION>
```