Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Nástroj pro modelování vlastností software

Plzeň, 2013 Jan Pikl

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. května 2013

Jan Pikl

Poděkování

Na tomto místě bych především rád poděkoval vedoucímu mé diplomové práce Ing. Ondřeji Rohlíkovi, Ph.D. za jeho ochotu, cenné rady a připomínky při tvorbě této práce. Dále bych chtěl poděkovat mé rodině a přátelům za jejich podporu během studia.

Abstract

Feature Modeling Tool

This work focuses on feature modeling and its tool support. The first part of the work evaluates state of the art in the field of feature modeling tools. It begins with a brief introduction to feature modeling and its historical and current development. It also offers overview of the currently available software tools. The second part defines requirements for a modern feature modeling tool, based on the review introduced earlier, and describes its implementation. This work also contains collection of sample feature models illustrating capabilities of the implemented tool.

Obsah

1 Úvod

Modelování vlastností je činnost, která nám umožňuje zachytit variabilitu v rámci složitého systému. Základním prvkem této činnosti je tvorba tzv. modelu vlastností, jež určuje shodnosti a různorodosti mezi jednotlivými variantami popisovaného objektu. Tyto varianty jsou mezi sebou rozlišeny pomocí jejich důležitých charakteristik, označovaných jako vlastnosti. Principy modelování vlastností jsou obecné a dají se použít v různých oblastech, jako je například průmyslová výroba či vývoj software, tedy zejména tam, kde může docházet ke vzniku různých variant vytvářeného produktu.

Tato práce se zabývá problematikou modelování vlastností, zejména pak jeho podporou ze strany softwarových nástrojů. Cílem této práce je návrh a implementace nástroje pro modelování vlastností, který by splňoval stanovené požadavky.

Čtenář je v úvodní části nejprve seznámen s principy modelování vlastností, jeho historickým a současným vývojem a jeho uplatněním v oblasti softwarového inženýrství. Dále jsou mu představeny některé dostupné modelovací nástroje a je provedeno jejich srovnání a zhodnocení. V následujících kapitolách jsou pak diskutovány požadavky kladené na modelovací nástroj, reflektující současný vývoj v oblastí modelování vlastností. Zbývající část práce se zabývá návrhem a popisem implementace nástroje, splňujícím vybranou podmnožinu těchto požadavků. Součástí práce je také sada příkladů demonstrujících použitelnost výsledného řešení.

2 Modelování vlastností

Cílem této kapitoly je seznámit čtenáře se základními principy modelování vlastností a jeho souvislostí se softwarovým inženýrstvím. Kapitola pak dále také mapuje historický a současný vývoj v oblasti modelování vlastností.

2.1 Motivace

V oblasti výroby, at' už například průmyslové či jiné, se často setkáváme s pojmem *rodina produktů* (z angl. *product family*), nazývaným někdy také jako *produktová řada* (z angl. *product line*)[1]. Tento pojem označuje skupinu produktů, které jsou postaveny na společném základu, ale liší mezi sebou v některých vlastnostech. Příkladem takové rodiny produktů může být například výroba automobilů, kdy výrobce dodává na trh několik variant téhož modelu odlišujících se v různých detailech (výkon, barva karoserie, příslušenství). Jiným příkladem může být výroba hardware, kdy jeden produkt může mít opět několik variant (viz tab. 2.1).

Označení	**Základní frekvence (GHz)**	**Velikost cache (MB)**	**Počet jader**	**Integrovaná GPU**	**AES**
i5-2320	3	6	4	Ano	Ano
i5-2550K	3,4	6	4	Ne	Ano
i5-660	3,33	4	2	Ano	Ano
i5-680	3,6	4	2	Ano	Ano
i5-760	2,8	8	4	Ne	Ne

Tabulka 2.1: Ukázka rodiny produktů CPU Intel Core i5.

Z uvedeného příkladu je vidět, že vlastnosti (sloupce), ve kterých jednotlivé varianty (řádky) odlišují, mohou mít různý charakter. Typicky mohou být vlastnosti v dané variantě přítomny či nepřítomny (integrovaná GPU), nebo mohou být vyjádřeny například číselnou hodnotu (frekvence CPU). Důležité je si uvědomit, že jednotlivé vlastnosti nemusí být nezávislé, ale mohou být navzájem provázány (například podpora AES může vyžadovat frekvenci CPU vyšší než 3GHz). Prostředek, který dokáže popsat jednotlivé vlastnosti a jejich vzájemné vazby, je tzv. model vlastností.

2.2 Doménové inženýrství

Ještě než formálně definuje pojem model vlastností, uveďme jej nejprve do kontextu softwarového inženýrství.

V předchozím textu jsme uvedli několik příkladů rodin produktů z oblasti výroby, avšak tento princip se dá stejným způsobem aplikovat i na vývoj software. Vezměme si

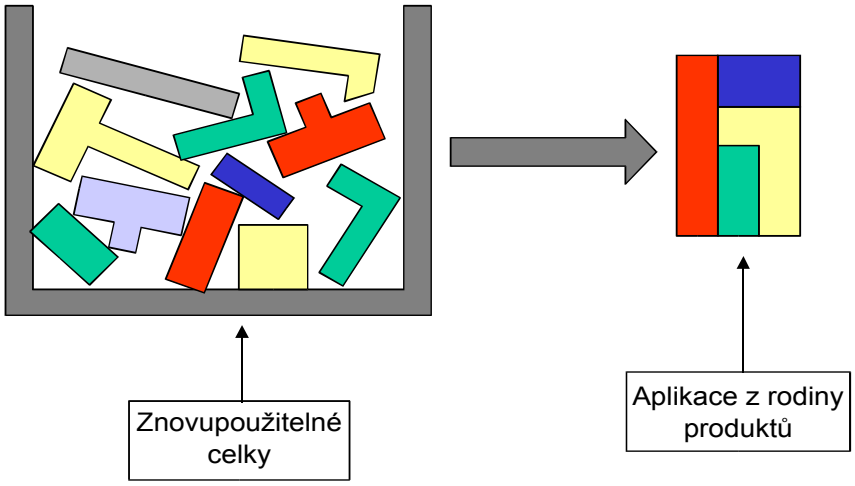
[1] Ačkoliv bývají tyto pojmy mezi sebou často zaměňovány, nemusejí vždy vyjadřovat to samé. Příkladem by mohla být produktová řada skládající se z několika odlišných rodin produktů.

například nějaký komplexní informační systém. Takový systém se bude pravděpodobně skládat z většího počtu komponent, kdy každá komponenta implementuje různou funkcionalitu a nějakým způsobem závisí i na ostatních komponentách. Přidáním nebo odebráním použitých komponent docílíme různých variant původní aplikace, které se liší v některých vlastnostech. Soubor systémů, které se dají tímto způsobem zkonstruovat, pak nazýváme jako *rodina softwarových produktů* (z angl. *software product family*) či někdy také jako *produktová řada software* (z angl. *software product line*, zkráceně SPL).

Proces, jehož účelem je vývoj právě takových rodin produktů, se nazývá *doménové inženýrství* (z angl. *domain engineering*[2]). Přesnou definici tohoto pojmu nalezneme například v [Cza00]:

> Doménové inženýrství je proces shromažďování, organizování a uchovávání poznatků získaných při vytváření systémů nebo jejich částí v dané oblasti (doméně) ve formě znovupoužitelných prvků (např. komponent), stejně tak poskytnutí adekvátních prostředků (např. vyhledání, kvalifikace, rozšiřování, úpravy, sestavení) pro znovupoužití těchto prvků při vytváření systémů nových.

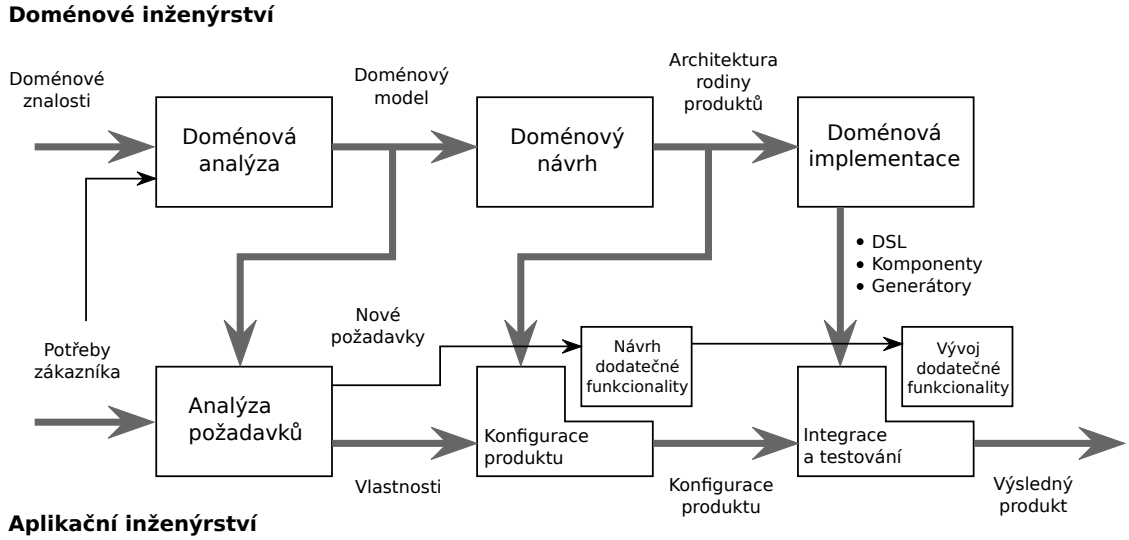
Hlavním rozdílem oproti konvenčnímu přístupu, kdy se snažíme na základě požadavků zákazníka vytvořit (jediný) cílový systém, je zde snaha vytvořit znovupoužitelné celky pro tvorbu celé rodiny takových systémů v dané doméně. Tento přístup můžeme ilustrovat obrázkem 2.1, převzatým z [Pas05].



Obrázek 2.1: Tvorba rodiny SW produktů ze znovupoužitelných celků.

Abychom lepé pochopili vztah mezi doménovým inženýrstvím a modelováním vlastností, popíšeme detailněji proces vývoje rodin SW produktů, tak jak je uveden v [Cza00]. Pro ilustraci celého procesu nám poslouží obrázek 2.2, převzatý ze zmíněné publikace.

[2]Označovaný také jako *product line engineering* či *product family engineering*



Obrázek 2.2: Vývoj software založený na doménovém inženýrství.

Doménové inženýrství Jak již bylo zmíněno, v tomto procesu dochází k vytvoření znovupoužitelných celků určených pro tvorbu aplikací v dané doméně. Tento proces, označovaný také jako *engineering for reuse*, se skládá celkem ze 3 částí:

- *Doménová analýza*: Je vybrána a zmapována cílová doména. Jejím výstupem je doménový model popisující danou doménu (její rozsah, data, vlastnosti atd.).
- *Doménový návrh*: Je navržena společná architektura pro systémy v dané doméně a sestrojen plán produkce.
- *Doménová implementace*: Navržená architektura je implementována v podobě znovupoužitelných celků (např. komponent, DSL[3], generátorů kódu atd.). Je vytvořena celková infrastruktura a proces produkce.

Aplikační inženýrství Zde dojde k sestavení konkrétní aplikace dle požadavků zákazníka z celků vytvořených během doménového inženýrství. Analogicky k předchozímu je tento proces označován jako *engineering by reuse*.

- *Analýza požadavků*: Jsou analyzovány požadavky zákazníka a určeny vlastnosti, které má mít výsledný produkt.
- *Konfigurace produktu*: Na základě požadovaných vlastností jsou vybrány celky pro implementaci produktu. Případně je i proveden návrh chybějící celků.
- *Integrace a testování*: Z vybraných celků je sestaven výsledný produkt. Chybějící funkcionalita je doprogramována.

[3]Doménově specifický jazyk (*domain specific language*). Na rozdíl od univerzálního programovacího jazyka je zaměřený na omezenou, konkrétní doménu.

V popsaném procesu je pro nás zajímavá část doménové analýzy, kde dochází k vytvoření doménového modelu, jehož součástí je právě i model vlastností, zachycující různorodosti a shodnosti mezi jednotlivými aplikacemi v rodině SW produktu.

2.3 Modely vlastností

Poprvé byly modely vlastností a jejich použití ve spojitosti s doménovým inženýrstvím popsány v roce 1990 v [Kang90]. Od té doby byla tato původní verze modelů vlastností různými autory rozšiřována, neboť se postupně objevovaly skutečnosti, které původní model nedokázal zachytit. Pro přehlednost ale nejprve uveďme původní model vlastností, tak jak byl definován ve zmíněné publikaci.

2.3.1 Vlastnost

Vlastnost (z angl. *feature*) můžeme dle [Kang90] definovat jako parametr systému, který přímo ovlivňují koncového uživatele. Jako příklad si můžeme představit zákazníka vybírajícího konkrétní variantu daného modelu automobilu. Vlastnosti, jež může brát zákazník v úvahu, jsou například:

- Výkon motoru.
- Typ motoru (benzínový, či dieselový).
- Barva karoserie (1 z N nabízených možností).
- To zda bude mít vůz automatické, či ruční řazení.
- To zda bude mít vůz klimatizaci či ne.

Ekvivalentně bychom dokázali uvést i příklady vlastností z oblasti vývoje SW (např. to zda bude/nebude daný SW podporovat zabezpečené přihlášení apod.). Jiní autoři, jako například [Cza00], uvádějí poněkud obecnější definici vlastnosti, a to jako důležitou charakteristiku systému, která je relevantní všem zainteresovaným osobám a jež je určena pro zachycení shodností a různorodostí mezi jednotlivými systémy z dané rodiny. Na rozdíl od té původní zde již není kladen důraz na viditelnost vlastnosti pro koncového zákazníka.

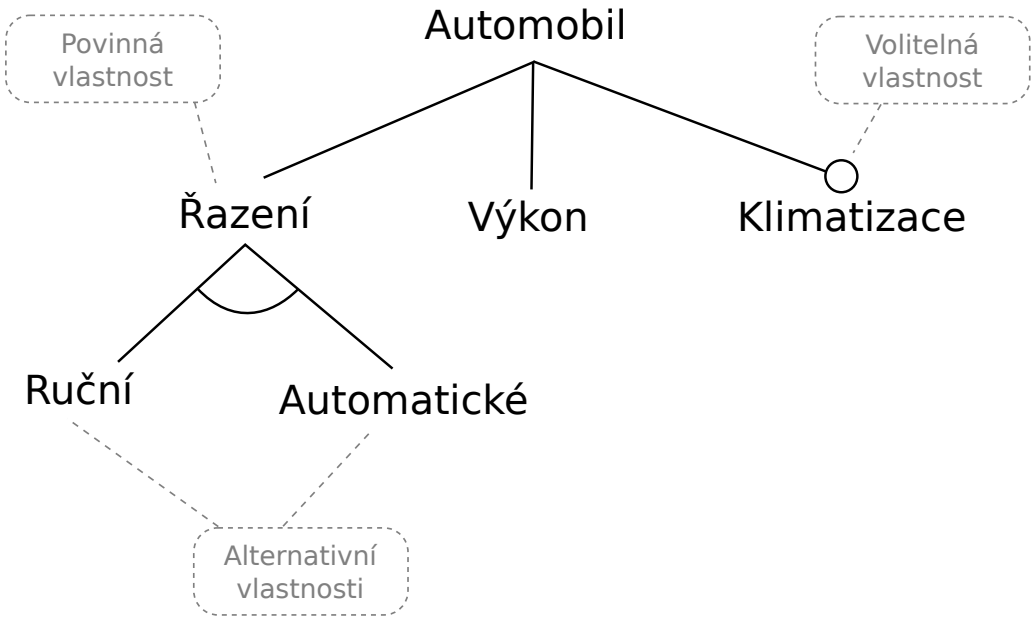
Na uvedeném příkladu si také všimněme různého charakteru popsaných vlastností. Zatímco u některých můžeme určit pouze to, zda budou ve výsledné variantě produktu zahrnuty či ne, u jiných máme na výběr z více možností. Mezi vlastnostmi můžou být navíc definovány různé vztahy a nelze je tedy většinou zachytit jako prostý výčet. Pro jejich popis se používá právě zmíněný model vlastností.

2.3.2 Diagram vlastností

Model vlastností (z angl. *feature model*) slouží k popisu shodností a různorodostí mezi jednotlivými systémy z dané rodiny produktů. Formálně se model vlastností skládá z

diagramu vlastností (z angl. *feature diagram*), jež graficky znázorňuje jednotlivé vlastnosti a jejich vazby, a dodatečných informací, popisujících skutečnosti, které nebyly v diagramu zahrnuty.

Jako ukázku diagramu vlastností uveďme přímo příklad z [Kang90] (obr. 2.3). Základním rysem všech modelů vlastností je, že jednotlivé vlastnosti jsou organizovány hierarchicky a výsledný diagram má tedy stromovou strukturu. Kořen tohoto stromu reprezentuje popisovaný systém[4], ostatní uzly pak jeho jednotlivé vlastnosti[5]. Každá vlastnosti by měla mít jméno, které odpovídá jejímu charakteru a které ji jednoznačně odlišuje od ostatních vlastností. Pro každou vlastnost dále platí, že může být ve výsledném systému[6] zahrnuta pouze tehdy, pokud je zde přítomná i jí přímo nadřazená vlastnost (rodič).



Obrázek 2.3: Příklad diagramu vlastností (FODA notace).

V diagramu se dále používají některá specifické grafické prvky. Vlastnost, která je označena prázdným kolečkem, nazýváme jako *volitelnou vlastnost* (z angl. *optional feature*). U takové vlastnosti si můžeme zvolit, zda bude ve výsledném systému zahrnuta či ne (za předpokladu že je zde přítomen i její rodič). Analogicky k této se zbývající vlastnosti označují jako *povinné* (z angl.*mandatory features*). Speciálním případem jsou pak tzv. *alternativní vlastnosti* (z angl. *alterative features*), z nichž musí být ve výsledném systému zahrnuta právě jedna. Počáteční spojení uzlů těchto vlastností jsou propojena prázdnou výsečí.

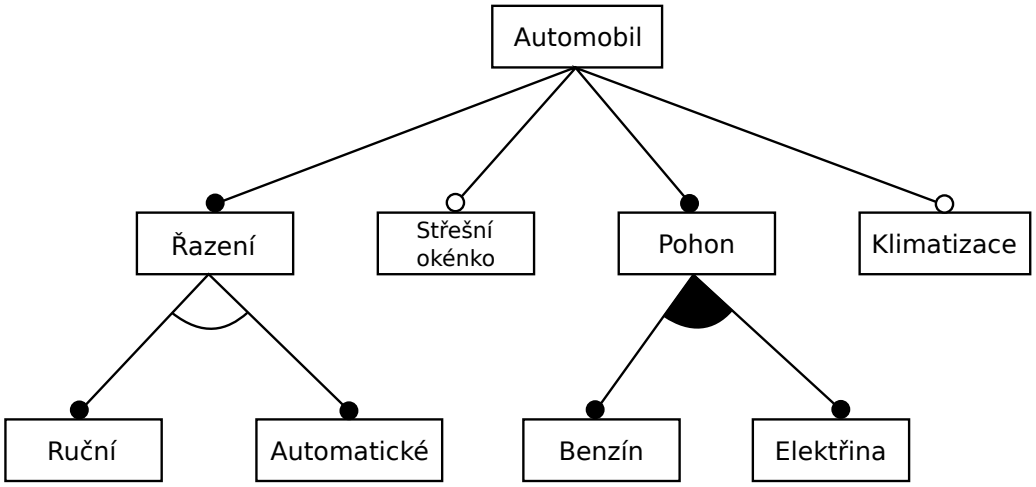
Notace použitá v diagramu na obrázku 2.3 se dle zdrojové publikace [Kang90] označuje

[4]Dle [Cza00] by měl kořenový prvek označován jako *konceptuální uzel* (z angl. *conceptual node*), neboť představuje zamýšlený koncept. Tento pojem je převzatý z oblasti *konceptuálního modelování* (z angl. *conceptual modeling*).

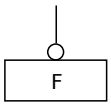
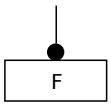
[5]Tyto uzly označujeme tedy jako *uzly vlastností* (z angl. *feature nodes*)

[6]Dle [Cza00] bychom místo měli spíše použít termín *popis instance daného konceptu* (z angl. *description of an instance of a concept*).

jako FODA[7] notace. Tato notace je již poměrně zastaralá a byla později nahrazena notací popsanou v [Cza98] a [Cza00]. Mírně modifikovaný diagram používající novější notaci můžeme vidět na obrázku 2.4. Hlavním rozdílem je zde orámování jednotlivých uzlů a označení povinných vlastností vyplněným kolečkem. Navíc přibyl speciální typ vlastností, tzv. *slučitelné vlastnosti* (z angl. *or-features*), označené vyplněnou výsečí. Pro danou množinu slučitelných vlastností platí, že ve výsledném systému musí být přítomna alespoň jedna z nich. Z důvodu přehlednost zanesme prvky této notace a jejich význam do tabulky 2.2.



Obrázek 2.4: Příklad diagramu vlastností (rozšířená notace autorů Czarnecki, Eisenecker).

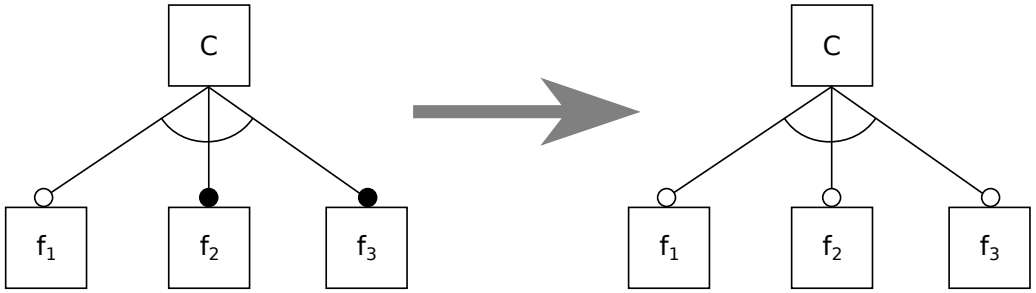
Symbol	**Název**	**Význam**
	Volitelná vlastnost.	Může být vybrána, byl-li vybrán její rodič.
	Povinná vlastnost.	Musí být vybrána, byl-li vybrán její rodič.
	Alternativní vlastnosti.	Právě 1 z N vlastností musí být vybrána, byl-li vybrán jejich rodič.
	Slučitelné vlastnosti.	Alespoň 1 z N vlastností musí být vybrána, byl-li vybrán jejich rodič.

Tabulka 2.2: Přehled prvků notace autorů Czarnecki, Eisenecker.

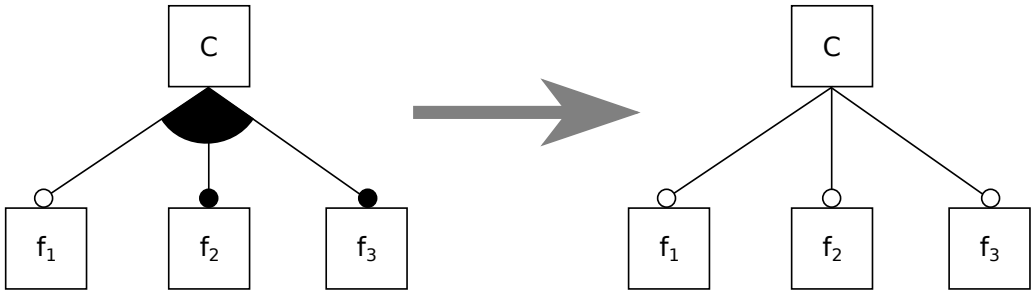
Pro úplnost je nezbytné poznamenat, že volitelnost se dá specifikovat i pro alternativní a slučitelné vlastnosti a vznikají tak další dva typy: *volitelné alternativní vlastnosti* a *volitelné slučitelné vlastnosti*. První uvedený typ označuje množinu vlastností, z nichž může

[7] Z názvu díla Fature Oriented Domain Analysis.

být ve výsledném systému přítomna nejvýše 1. Pokud máme množinu alternativních vlastností, z nichž některé jsou volitelné a jiné povinné, je tato množina ekvivalentní množině alternativních vlastností, kde jsou volitelné všechny (viz obr. 2.5). Druhý uvedený typ je v podstatě redundantní, neboť je ekvivalentní použití (samostatných) volitelných vlastností, jak ukazuje obrázek 2.6. Oba zmíněné obrázky byly převzaty z [Cza00]. Proces transformace naznačený na těchto obrázcích nazýváme *normalizací diagramu vlastností*. Pro přehlednost uveďme kombinace všech typů vlastností a jejich sémantiku do tabulky 2.3



Obrázek 2.5: Normalizace skupiny alternativních vlastností z nichž jen některé jsou volitelné



Obrázek 2.6: Normalizace skupiny slučitelných vlastností z nichž některé jsou volitelné

	Volitelná vlastnost	**Povinná vlastnost**
Alternativní vlastnosti	Nejvýše 1 vlastnost z dané množiny musí být vybrána.	Právě 1 vlastnost z dané množiny musí být vybrána.
Slučitelné vlastnosti	Není požadavek na to která vlastnost musí být vybrána.	Alespoň 1 vlastnost z dané množiny musí být vybrána.

Tabulka 2.3: Přehled jednotlivých kombinací typů vlastností a jejich sémantika.

2.3.3 Doplňující informace

Na začátku předchozí části jsme zmínili, že součástí modelu vlastností není jen samotný diagram, ale i různé doplňující údaje. Následuje výčet takových možných údajů, tak jak je uveden v [Kang90] a [Cza00]:

- *Popis sémantiky*: Každá vlastnost by měla mít alespoň krátký popis toho, co představuje. Samotný popis by nemusel obsahovat jen text, ale i jiné prvky (např. diagramy, pseudokód atd.). Popis by měl zajistit provázání vlastnosti s ostatními modely používanými při vývoji.
- *Odůvodnění*: Zdůvodnění, proč je daná vlastnost v modelu zahrnuta. Může obsahovat doporučení, kdy a za jakých podmínek vhodné je danou (volitelnou) vlastnost vybrat.
- *Zainteresované osoby a aplikace*: Seznam zainteresovaných osob, pro které je tato vlastnost důležitá (např. zákazníci, uživatelé, vývojáři atd.), případně aplikace požadující danou vlastnost (daná vlastnost představuje funkcionalitu nějaké komponenty).
- *Příklady systémů*: Příklady systému, které danou vlastnost implementují.
- *Priorita*: Ohodnocení, udávající důležitost vlastnosti v rámci daného projektu.
- *Otevřenost*: Označení, zda bude daná vlastnost v budoucnu rozšiřována (např. o další pod-vlastnosti).
- *Omezení*: Popis další vazeb mezi jednotlivými vlastnostmi, které nešly vyjádřit grafickou notací v diagramu. Příkladem (pro obr. 2.4) by mohl být například požadavek *klimatizace vyžaduje benzínový pohon* (závislost), nebo *nelze mít zároveň střešní okénko a klimatizaci* (vyloučení). Tato pravidla a možnosti jejich zápisu budou detailněji diskutována v části 2.8.
- *Místo dostupnosti*: Definuje kdy, kde a komu je daná vlastnost v doméně dostupná.
- *Místo vazby*: Definuje kdy, kde a kdo může danou vlastnost provázat se systémem. Rozlišujeme tři různé okamžiky, kdy může k provázání dojít:
  - *Při překladu*: Překladač zahrne danou vlastnost při sestavení aplikace.
  - *Při spuštění*: Vybrané vlastnosti jsou do aplikace zahrnuty při jejím spuštění.
  - *Za běhu*: Vlastnost může být za běhu aplikace interaktivně aktivována/deaktivována.
- *Způsob vazby*: Určuje, jakým způsobem je provázání dané vlastnosti se systémem docíleno.
  - *Statická vazba*: Provázání je určeno předem.
  - *Dynamická vazba*: Provázání je určeno v okamžiku použití.[8]
  - *Proměnlivá vazba*: Provázání zůstává mezi jednotlivými použitími stálé, ale může se změnit.

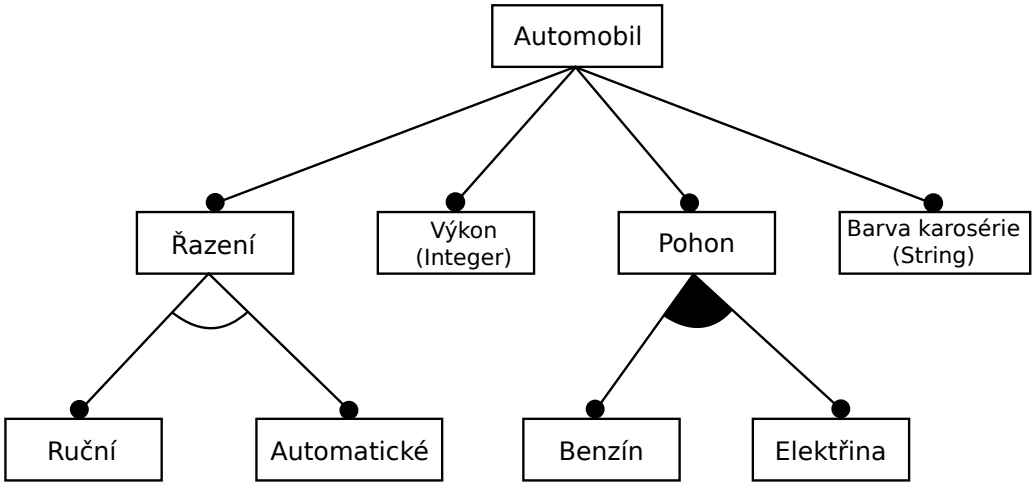
[8] Příkladem takové vazby může být volání virtuálních metod v programovacích jazycích C++ a Java.

2.4 Modely vlastností s atributy

Od vzniku prvních definic modelu vlastností [Kang90], [Cza98] a [Cza00] se postupně objevovaly skutečnosti, které takto definovaný model nedokázal zachytit. Příkladem budiž potřeba specifikovat v modelu číselné hodnoty (např. výkon automobilu). Ačkoliv publikace [Kang90] asociaci číselné hodnoty s vlastností přímo nevylučovala, [Cza00] takovou možnost nijak neuvažuje (u vlastnosti můžeme pouze specifikovat to, zda je v systému zahrnuta či ne). Nicméně, v pozdějších letech ti samí autoři, na základě praktických zkušeností, zavádějí již pojem *atribut*.

Atribut je prvek, jež slouží k výběru jedné z většího množství hodnot z dané domény. Publikace [Cza05b] uvádí možnost asociovat s každou vlastností jeden atribut zvoleného typu, jehož hodnota je pak určena při konfiguraci daného systému. Typ atributu může být buď elementární (např. číslo, řetězec), nebo může představovat odkaz (referenci) na jinou vlastnost, která je v daném systému přítomna. V případě potřeby mít pro danou vlastnost specifikováno více atributů se tyto atributy zavedou v podobě několika pod-vlastností (potomci daného uzlu v diagramu vlastností). Pro ilustraci použití atributů uveďme obrázek 2.7 používající pro atributy notaci[9] zavedenou v [Cza05b]. Příklad vlastnosti s atributem typu reference bude uveden později v části 2.8 po představení některých dalších pojmů.

Jiní autoři, jako například [Pas05], naopak umožňují s jednou vlastností asociovat libovolný počet atributů, zde nazývaných jako *properties*. V nástroji popsaném v této publikaci lze navíc jednotlivé atributy seskupovat do logických celků, označovaných jako *property set*. Autoři tento přístup odůvodňují praktickým použitím, kdy reálné modely vlastností mohou mít i velký počet atributů na jednu vlastnost. Diagram takového modelu by byl při použití předchozího přístupu (maximálně jeden atribut pro jednu vlastnost) velmi nepřehledný.



Obrázek 2.7: Diagram vlastností obsahující atributy.

[9] Čtenář se může pozastavit nad tím, že v této notaci není pro daný atribut specifikováno žádné jméno, ale jen datový typ. Vzhledem k tomu, že vlastnost může mít pouze 1 atribut, je uvedení jeho jména zbytečné (s vlastností je defakto asociován pouze typ atributu).

2.5 Modely vlastností s kardinalitou

Další z řady rozšíření původního konceptu modelu vlastností bylo zavedení tzv. *kardinality* pro jeho jednotlivé prvky. Abychom lepé pochopili důvod pro takové rozšíření, podívejme nejprve se na obrázek 2.8a, obsahující diagram vlastností jednoduchého řídícího systému. Vidíme, že systém obsahuje snímač, který může mít buď část pro měření polohy, pohybu nebo teploty a volitelně může být schopen i autodiagnostiky. Problém nastane, pokud bychom potřebovali popsat řídící systém, jež by měl takovýchto snímačů několik a kde by každý snímač mohl být nakonfigurován individuálním způsobem. S použitím původní notace bychom museli mít v diagramu několik identických podstromů a ani tím bychom však nedokázali zachytit požadavek, kdy by měl být počet takových snímačů volitelný.

Na modifikovaném obrázku 2.8b vidíme označení této vlastnosti kardinalitou [1..4], která nám říká, že daná vlastnost (snímač) musí být ve výsledném systému zahrnuta v jedné až čtyřech kopiích, kde každá taková kopie může být nakonfigurována individuálním způsobem. Obecně vyjadřujeme kardinalitu vlastnosti jako interval ve tvaru $[m..n]$, kde m je dolní a n horní hranice počtu výskytů dané vlastnosti v systému ($0 \leq m \leq n \ \wedge \ 0 < n$). Navíc je možné místo n dosadit zástupný symbol $*$, označující neomezenou horní hranici[10].

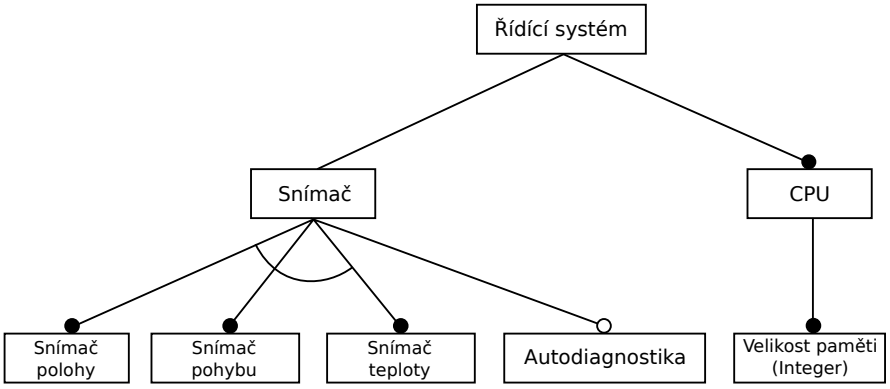
Dále si povšimněme, že volitelné a povinné vlastnosti jsou v této interpretaci vlastně speciálními případy vlastností s kardinalitou [0..1] a [1..1]. Pro tyto vlastnosti se však z důvodu přehlednosti (a jejich častého použití) využívá v diagramu původní notace (prázdné/vyplněné kolečko). Vlastnosti, které mají horní mez kardinality vyšší než 1, označujeme jako *duplikovatelné* (z angl. *clonnable*). Dle dolní hranice je pak dále přesněji rozdělujeme na *volitelné duplikovatelné* (*optional clonnable*) a *povinné duplikovatelné* (*mandatory clonnable*).

Výše uvedený popis a notace je převzata z [Cza05b]. Samotná myšlenka použití kardinality při popisu vlastností se ale prvně objevila již v [Rieb02][11], jejíž autoři se inspirovali UML notací. Tato publikace však ještě nezavádí kardinalitu přímo, ale používá ji jako prostředek pro zobecnění pojmů alternativní a slučitelné vlastnosti. Důvod pro toto zobecnění můžeme ukázat na již zmíněném obrázku 2.8b. Původní notace nám umožňuje zachytit pouze 4 (respektive 3) omezující podmínky, jak bylo ukázáno v tabulce 2.3. Pokud bychom tedy v rámci našeho příkladu potřebovali zachytit fakt, že snímač může mít nejen 1, ale i 2 měřící části, museli bychom to vyjádřit pomocí dodatečných omezení mimo diagram (viz podkapitola 2.8).

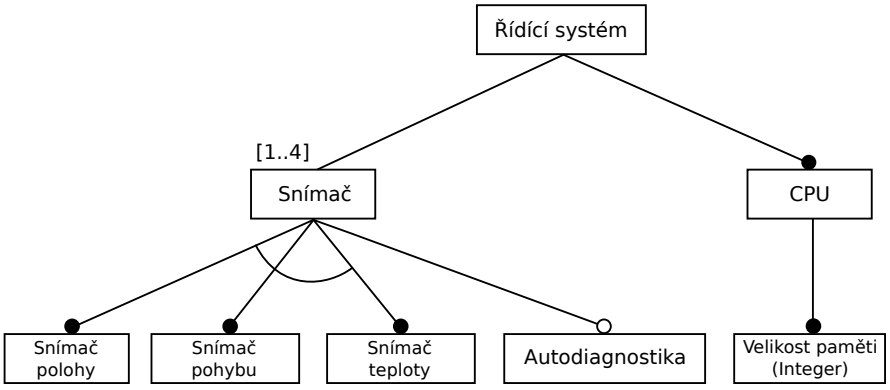
Autoři [Rieb02] proto nahrazují alternativní a slučitelné vlastnosti pojmem *skupina vlastností* (z angl. *feature group*), jež představuje množinu vlastností (mající společného rodiče), která má specifikovanou kardinalitu. Kardinalitu skupiny vlastností zapisuje dle [Cza05b] jako interval ve tvaru $\langle i - j \rangle$, s podobnými pravidly, která platila pro kardinalitu vlastností. Pro danou skupinu vlastností s kardinalitou $\langle i - j \rangle$ poté platí, že ve výsledném systému musí být z této skupiny přítomno nejméně i a nejvíce pak j vlastností. Zde můžeme vidět, že alternativní a slučitelné vlastnosti lze vlastně chápat jako speciální případ skupin s kardinalitou $\langle 1 - 1 \rangle$ a $\langle 1 - k \rangle$ (kde k je velikost skupiny[12]). Pro tyto speciální

[10] Čtenář seznámený s notací UML si jistě povšiml, že je tato notace v podstatě identická.
[11] Místo *cardinality* zde bylo ještě používáno označení *multiplicity*.
[12] Místo k by zde šlo samozřejmě také dosadit symbol $*$.

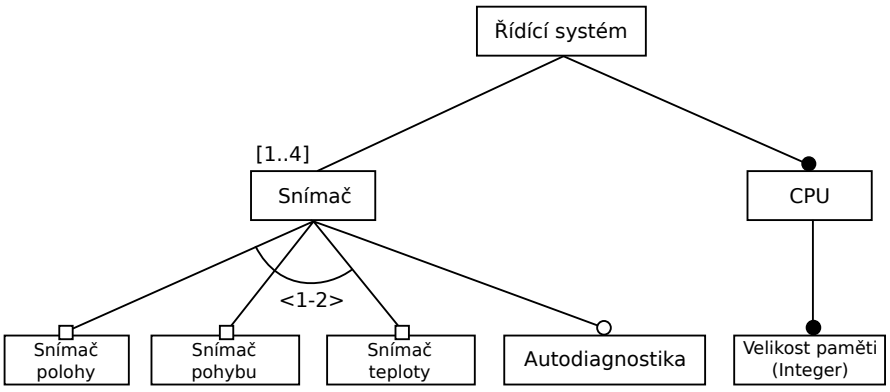
případy, označované jako *XOR skupina* a *OR skupina*, se z důvodu přehlednosti (a jejich častého použití) používá v diagramu původní notace (prázdná/vyplněná výseč). Notaci pro obecnou skupinu lze vidět na obr. 2.8c.



(a) Původní notace.

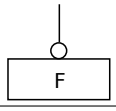
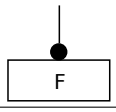
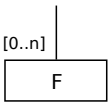
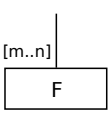
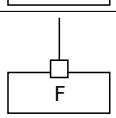
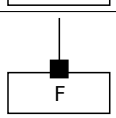
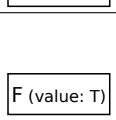
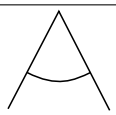


(b) Nová notace umožňující vyjádřit násobnost vlastnosti.



(c) Nová notace zavádějící skupiny.

Obrázek 2.8: Zavedení kardinality u diagramu vlastností.

Symbol	Popis
	Samostatná vlastnost s kardinalitou [0..1], tedy *volitelná vlastnost*.
	Samostatná vlastnost s kardinalitou [1..1], tedy *povinná vlastnost*.
	Samostatná vlastnost s kardinalitou $[0..n]$, $1 < n$, tedy *volitelná duplikovatelná vlastnost*.
	Samostatná vlastnost s kardinalitou $[m..n]$, $1 < n \ \wedge \ 0 < m \leq n$, tedy *povinná duplikovatelná vlastnost*.
	Seskupená vlastnost s kardinalitou [0..1].
	Seskupená vlastnost s kardinalitou [1..1].
	Vlastnost s *atributem* typu T a hodnotě *value*[13].
	Skupina vlastností s kardinalitou $\langle 1-1 \rangle$, tedy *XOR skupina*.
	Skupina vlastností s kardinalitou $\langle 1-k \rangle$ (k je velikost skupiny), tedy *OR skupina*.
	Skupina vlastností s kardinalitou $\langle i-j \rangle$

Tabulka 2.4: Notace diagramu vlastností s kardinalitou.

Modely vlastností využívající kardinalitu vlastností a skupin označujeme dle [Cza04] souhrnně jako *modely vlastností s kardinalitou* (z angl. *cardinality-based feature models*). Vlastnosti tohoto modelu pak rozdělujeme na tzv. *seskupené* (z angl. *grouped*) a *samostatné* (z angl. *solitary*) dle toho, zda jsou, či nejsou součástí nějaké skupiny.

Zde je velmi důležité upozornit na rozdílné chápání seskupených vlastností oproti původním přístupu. Podívejme se na obrázek 2.8a ukazující starou notaci a 2.8c používající novou notaci popsanou v [Cza05b]. Zanedbejme nyní rozdílný význam alternativních vlastností (obr. 2.8a) a použité skupiny $\langle 1-2 \rangle$ (obr. 2.8c) a zaměřme se pouze na seskupené vlastnosti. Vidíme, že alternativní vlastnosti, které byly v 2.8a povinné, jsou v 2.8c označeny jako volitelné. To je dáno tím, jak chápeme význam seskupených vlastností. Konkrétní

[13]Hodnota atributu se uvádí až při procesu konfigurace či specializace, viz podkapitola 2.6.

případ na obr. 2.8c bychom interpretovali jako: skupina z níž musí být vybrány 1-2 vlastnosti (kardinalita skupiny), kde každá vlastností musí být v rámci této skupiny vybrána 0-1 krát (kardinalita vlastností). Pokud bychom nějaké ze seskupených vlastností přiřadili kardinalitu [1..1], musela by být tato vlastnost v rámci skupiny vybrána vždy. Čtenář necht' si porovná tuto sémantiku s původním přístupem (tab. 2.3). Z důvodu těchto rozdílů proto [Cza05b] zavádí pro seskupené vlastnosti odlišnou notaci, a to označení čtverečky místo koleček (viz obr. 2.8c).

Abychom shrnuli doposud uvedená fakta, zobrazme prvky nové notace spolu s jejich popisem přehledně do tabulky 2.4 a jejich význam do tabulky 2.5. K této notaci je důležité zmínit několik omezení. Za prvé, vlastnost může být součástí pouze jediné skupiny. Druhým omezení je, že seskupené vlastnosti mohou mít kardinalitu pouze [0..1][14] a nelze je tedy duplikovat. Pokud bychom potřebovali mít seskupenou vlastnost duplikovatelnou, museli bychom tento požadavek realizovat v podobě její pod-vlastnosti, pro kterou již toto omezení neplatí. Je zajímavé podotknout, že například jiní autoři [Cech04] toto omezení na seskupené vlastností nekladou.

Prvek	**Význam**
Vlastnost s kardinalitou $[m..n]$.	Ve výsledném systému musí být přítomno nejméně m a nejvíce pak n kopií dané vlastnosti. Toto omezení se vždy vyhodnocuje v rámci společného rodičovského uzlu jednotlivých vlastností.
Skupina vlastností s kardinalitou $\langle i-j \rangle$.	Z dané skupiny vlastností musí být ve výsledném systému přítomno nejméně i a nejvíce pak j vlastností. Toto omezení se vždy vyhodnocuje v rámci společného rodičovského uzlu jednotlivých vlastností.

Tabulka 2.5: Sémantika notace diagramu vlastností s kardinalitou.

Na závěr této podkapitoly ještě dodejme možnost dalšího zobecnění konceptu kardinality v modelu vlastností, tak jak ho popisují autoři [Cza04]. Kardinalita vlasností by mohla být chápána ne jako jeden interval, ale jako sekvence několika intervalů. Například [1..1][5..10] by udávalo, že se daná vlastnost musí ve výsledném systému objevit jednou či 5-10 krát. Pro použití tohoto principu i pro skupiny však sami autoři nevidí přílišné praktické uplatnění.

2.6 Konfigurace modelu vlastností

V části 2.2 jsme popsali využití modelu vlastností pro fázi analýzy v doménovém inženýrství. Na základě takto definovaného modelu se pak při tvorbě výsledného systému vyberou

[14]Respektive mohou mít i kardinalitu [0..0] a [1..1], jak je vidět z tab. 2.4, ale tyto hodnoty se dle [Cza05b] používají až při procesu specializace, který bude popsán samostatně v podkapitole 2.7. Zajímavé je podotknout, že stejní autoři ve svých předchozích publikacích [Cza04] a [Cza05a] nejprve kardinalitu seskupených vlastností zavrhují jako zbytečnou.

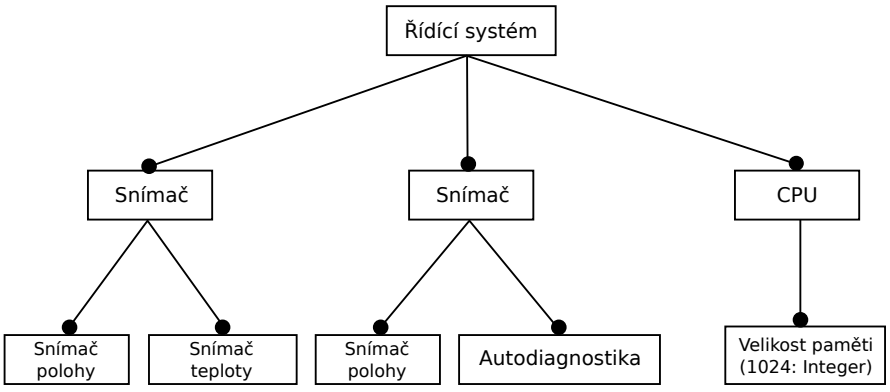
vlastnosti, které v něm budou zahrnuty (například použitím zvolených komponent). To, které kombinace vybraných vlastností můžeme považovat za validní, nám definuje právě model vlastností.

Soubor vlastností vybraných v souladu s pravidly definovanými příslušným modelem vlastností nazýváme dle [Cza04] jako *konfigurace modelu vlastností* (z angl. *feature model configuration*) nebo pouze zkráceně *konfigurace*. Proces, při kterém dojde k vybrání takové validní kombinace vlastností, pak nazýváme *procesem konfigurace*.

Na tomto místě vhodné je uvést, jakým způsobem reprezentovat daný soubor vybraných vlastností. Pokud bychom uvažovali původní model vlastností, tak jak je popsaný v [Kang90] a [Cza00], můžeme konfiguraci definovat jako pouhý výčet (množinu) vlastností. Pokud bychom uvažovali model vlastností s kardinalitou a atributy, už si s takto jednoduchou reprezentací nevystačíme, a to ze dvou důvodů:

1. Konfigurace musí pro každý atribut obsahovat i jeho hodnotu dle specifikovaného typu (číslo, řetězec, reference apod.).
2. U vlastností, jejichž rodičem je duplikovatelná vlastnost, je nutné uvést, ke které z rodičovských kopií jsou přiřazeny. Jinými slovy, musíme definovat *kontext*, v jakém jsou vybrány.

Z těchto důvodů se konfigurace modelu vlastností tedy reprezentuje také jako stromová struktura. Diagram znázorňující konfiguraci pak používá stejnou notaci jako diagram modelu vlastností. Jako příklad si na obrázku 2.9 ukažme jednu z validních konfigurací modelu z obrázku 2.8c.



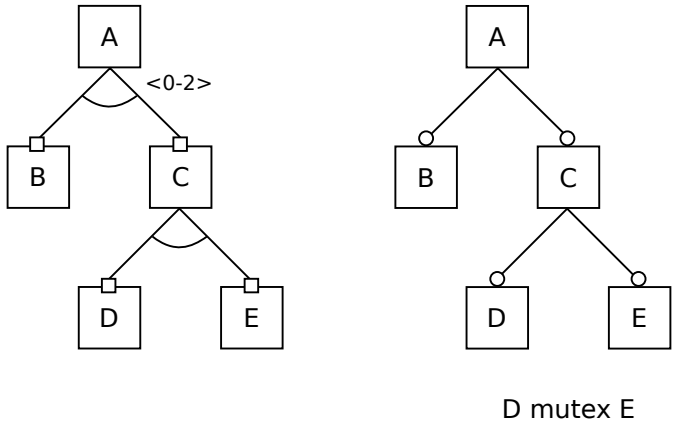
Obrázek 2.9: Ukázka konfigurace modelu vlastností z obr. 2.8c.

Každý model vlastností definuje jistou množinu validních konfigurací. Tuto množinu můžeme dle [Cza04] nazývat jako *prostor konfigurací* (z angl. *configuration space*). Dva modely vlastností pak můžeme považovat za ekvivalentní, pokud je jejich konfigurační prostor identický, jak tvrdí [Cza00].

Příklad dvou modelů vlastností ekvivalentních na základě této definice můžeme vidět na obrázku 2.10. V tomto případě má použití skupiny $\langle 0 - 2 \rangle$ na obr. 2.10a stejný efekt

jako použití samostatných volitelných vlastností na obr. 2.10b. V druhém uvedeném diagramu pak došlo k vyjádření výlučnosti D a E pomocí dodatečného (textově zadaného) omezení namísto použití XOR skupiny jako v diagramu prvním. Tato omezení budou později diskutována v části 2.8. Pro ukázku také uveďme výčet všech validních konfigurací popsaných oběma modely: $\{A, AB, AC, ABC, ACD, ACE, ABCD, ABCE\}$[15].

Jak vidíme, stejná situace se dala v příkladu vyjádřit dvěma různými způsoby, tedy máme zde redundanci. Autoři [Cza04] naznačují, že by bylo možné provádět mezi těmito reprezentacemi konverzi a případně i normalizaci do nějaké preferované formy, avšak toto nechávají na implementátorech případného modelovacího nástroje.



(a) První model. (b) Druhý model.

Obrázek 2.10: Ekvivalence dvou modelů vlastností z hlediska jejich prostoru konfigurací.

2.7 Specializace modelu vlastností

Proces konfigurace, popsaný v předchozí části, umožňoval na základě modelu vlastností sestrojit jednu z jeho validních konfigurací. Publikace [Cza04] uvádí obecnější přístup, který nazývá *procesem specializace*.

Říkáme, že diagram vlastností Y je *specializací* diagramu vlastností X, pokud je prostor všech konfigurací Y podmnožinou prostoru konfigurací X. Jinými slovy, Y vznikl transformací z X zpřísněním některých jeho omezení a platí tedy, že libovolná platná konfigurace Y je i platnou konfigurací pro X, ale ne naopak[16].

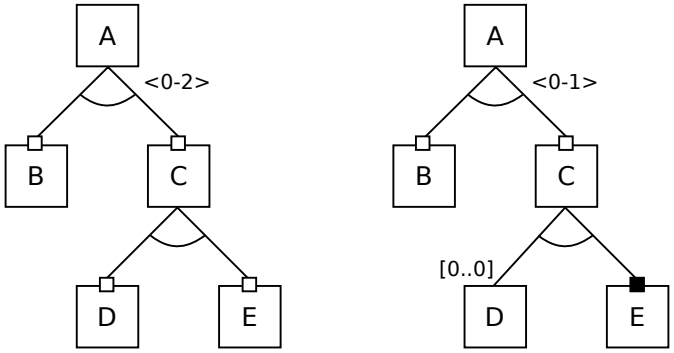
Jako příklad specializace si uveďme diagramy na obrázku 2.11. Diagram 2.11a představuje model vlastností s možnými konfiguracemi $\{A, AB, AC, ABC, ACD, ACE, ABCD, ABCE\}$. U jeho specializace na obr. 2.11b jsme provedli změnu kardinality skupiny ze $\langle 0-2 \rangle$ na $\langle 0-1 \rangle$ a navíc jsme v rámci C explicitně vybrali vlastnost E a eliminovali vlastnost D. Takto modifikovaný diagram vlastností má jako možné konfigurace pouze

[15] Vzhledem k nepoužití duplikovatelných vlastností jsme si ho mohli dovolit uvést v tomto tvaru.

[16] Ačkoliv specializujeme pouze diagram, můžeme tento proces souhrnně označovat jako specializace modelu vlastností.

$\{A, AB, ACE\}$. Čtenář nechť si sám ověří, že jde o podmnožinu konfigurací původního diagramu.

Pro úplnost dodejme, že na proces konfigurace se můžeme dívat jako zvláštní případ procesu specializace, jehož výstupem je diagram vlastností mající pouze jedinou platnou konfiguraci.



(a) Diagram vlastností. (b) Specializace předchozího diagramu.

Obrázek 2.11: Příklad specializace diagramu vlastností.

2.7.1 Konfigurace ve fázích

V případě rozsáhlých modelů vlastností může být tvorba konfigurace zdlouhavá a komplikovaná. Navíc můžou o výběru různých vlastností v konfiguraci rozhodovat odlišné osoby (zákazníci, vývojáři, vedoucí atd.). Autoři [Cza04] proto navrhují postup, který nazývají *konfigurace ve fázích* (z angl. *staged configuration*).

Základní myšlenkou tohoto přístupu je rozdělení procesu konfigurace na několik fází, kdy v každé fázi dojde k eliminaci některých nechtěných konfigurací. Výsledkem poslední fáze je pak jediná, cílová konfigurace. Jinými slovy, v jednotlivých fázích postupně transformujeme model vlastností tak, abychom zmenšovali výsledný konfigurační prostor.

Jeden ze dvou způsobů jakými docílit konfigurace ve fázích je, dle [Cza04], právě již zmíněný proces specializace. Druhým možným způsobem je pak tzv. *víceúrovňová konfigurace* (z angl. *multi-level configuration*), kdy je pro každou fázi (úroveň) používán odlišný model vlastností. Přechod mezi jednotlivými fázemi probíhá tak, že je na základě výstupní konfigurace z n-té fáze vygenerován vstupní model vlastností fáze $n + 1$. Hlavní výhodou tohoto přístupu oproti specializaci je, že osoba provádějící konfiguraci v dané fázi vidí pouze malou část ze všech možných voleb. Zatímco proces specializace začíná s úplným modelem popisujícím všechny možné konfigurace, víceúrovňová konfigurace může mít v první fázi zjednodušený model, popisující jen několik relevantních možností.

Detailnější popis této problematiky přesahuje svým rozsahem rámec této práce a pro další informace proto odkážeme čtenáře na zmíněnou literaturu [Cza04].

2.8 Specifikace omezení

Diagram vlastností je poměrně efektivní nástroj pro znázornění jednotlivých vlastností a jejich vzájemných vztahů, avšak některé skutečnosti zachytit nedokáže. Podívejme se zpět na příklad z obrázku 2.8c, znázorňující jednoduchý řídící systém. V tomto diagramu jsme dokázali vyjádřit některá omezení pomocí kardinalit vlastností (řídící systém má 1 až 4 snímače) a skupin (snímač má 1 až 2 měřící části). Problém by však nastal, pokud bychom potřebovali definovat omezení typu: *snímač může mít 2 měřící části, jen pokud je velikost interní paměti alespoň 1024B.*

Omezení prvního typu (tedy kardinality vlastností a skupin) označují někteří autoři [Pas05] jako tzv. *lokální omezení* (z angl. *local constraints*). Tato omezení se vždy vztahují pouze ke společnému rodiči daných vlastností. Omezení druhého typu, které popisovala obecný vztah mezi vlastnostmi z různých částí stromu, pak [Pas05] nazývá jako *globální omezení* (z angl. *global constraints*). Pokud mluvíme v rámci modelu vlastností obecně o *omezeních* (*constraints*), máme na mysli většinou právě omezení globální typu a takto budeme chápat i význam slova omezení ve zbytku tohoto textu.

Již původní publikace [Kang90] popisovala nutnost specifikace omezení, zde ještě nazývaných jako kompoziční pravidla (z angl. composition rules). Tato omezení zde byla rozdělena do dvou kategorií:

1. Omezení typu závislost. Například *vlastnost A vyžaduje vlastnost B*, jež bychom formálně zapsali jako `A requires B`.
2. Omezení typu vyloučení. Například *nelze mít zároveň A i B*, formálně zapsané jako `A mutex-with B`.

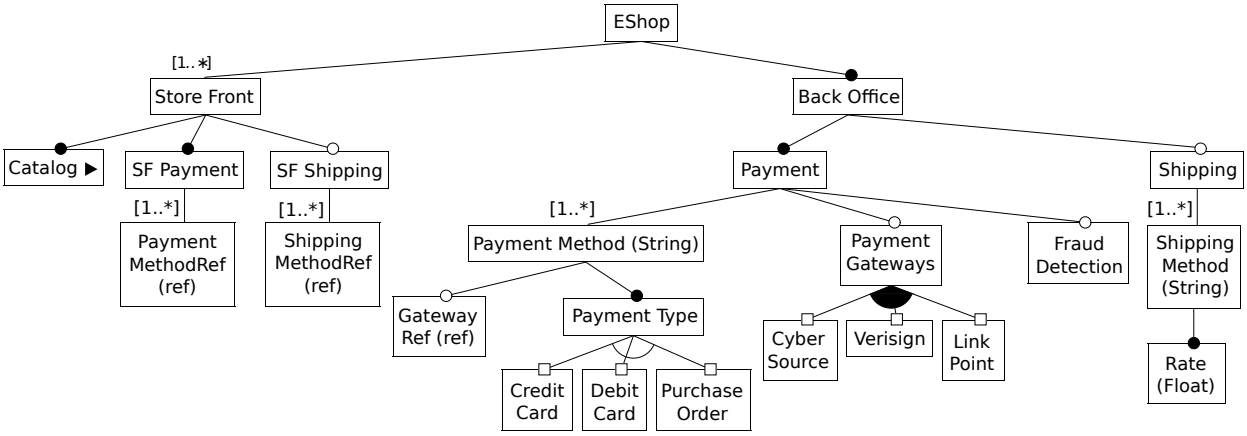
Obdobným způsobem jsou omezení uvažována i v [Cza00]. Obecně můžeme omezení chápat jako logický výraz zapsaný v textové formě, který je pro danou konfiguraci buď vyhodnocen jako pravdivý (konfigurace je validní) nebo nepravdivý (konfigurace je nevalidní). Aby byl daný logický výraz snadno strojově zpracovatelný a vyhodnotitelný, musí vyhovovat pravidlům gramatiky daného jazyka. Tento jazyk pak označujeme jako *jazyk omezení* (z angl. *constraint language*). Jako příklad takového jazyka si můžeme uvést jednoduchou gramatiku výše zmíněných omezení převzatou z [Kang90]:

```
<feature1> (‘requires’ | ‘mutex-with’) <feature2>
```

Obecně však není nikde specifikováno, jaký jazyk by měl být k vyjádření omezení použit. Autoři [Cza05b] uvádějí možnost použití jazyka OCL[17] pro modely vlastností s kardinalitou. Jako výhodu zmiňují jeho dobře definovanou sémantiku a to, že s ním budou někteří vývojáři pravděpodobně již obeznámeni. Jiní autoři, například [Cech04], navrhují pro popis omezení jazyk XPath[18], původně určený pro zpracování XML dokumentů. Dalším příkladem je nástroj *pure::variants*, používající pro vyjádření omezení vlastní dialekt jazyka Prolog [Pure13].

[17] Object Constraint Language (http://www.omg.org/spec/OCL/)
[18] XML Path Language (http://www.w3.org/TR/xpath20/)

Pro ukázku použití omezení uveďme příklad komplexnějšího modelu vlastností, převzatého z [Cza05b], jež je znázorněný na obrázku 2.12. Vidíme, že jde o diagram vlastností imaginárního internetového obchodu, který má administrativní část (back office) a jednu nebo více prodejních míst (front store). Pro administrativní část můžeme vytvořit několik různě nakonfigurovaných platebních metod a případně i několik doručovacích metod. Jednotlivým prodejním místům pak můžeme tyto platební a doručovací metody přiřazovat. Tento příklad je ukázka vhodného použití atributů typu reference, kdy se s jejich pomocí můžeme odkazovat na již existující prvky konfigurace (doručovací a platební metody). Nicméně, na tomto příkladu si ukážeme použití OCL, jako jazyka pro popis omezení. Níže uvedené příklady omezení (výpis 2.1) jsou opět převzaty z [Cza05b][19].



Obrázek 2.12: Příklad modelu vlastností (E-Shop).

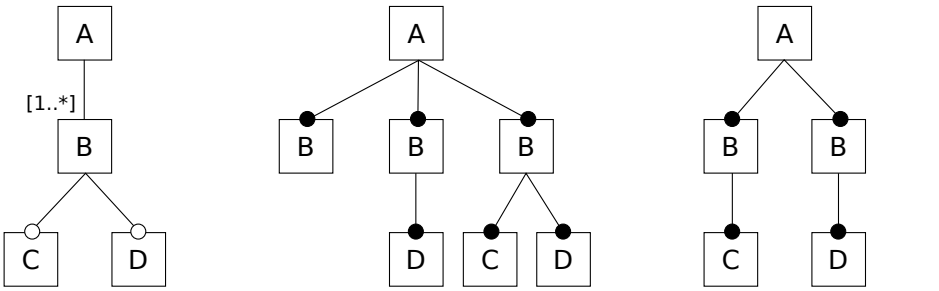
```
context Payment inv:
    FraudDetection.isSelected() implies PaymentGateways.isSelected()

context GatewayRef inv:
    EShop.BackOffice.Payment.PaymentGateways.sub()->includes(attribute)

context StoreFront inv:
    SFPayment.PaymentMethodRef->size() <= 3

context Rate inv:
    att >= 0
```

Výpis 2.1: Příklady omezení v jazyce OCL.

První omezení říká, že pokud byla vybrána vlastnost odhalování podvodů, musí být vybrána i nějaká platební brána (omezení typu závislost). Druhé omezení zaručuje, že reference na platební bránu bude vždy na nějakou existující platební bránu odkazovat. Třetí omezení udává, že prodejní místo nemůže podporovat více jak tři platební metody. Poslední omezení jednoduše zakazuje zápornou či nulovou přepravní rychlost.

[19]Použití OCL v tomto příkladě předpokládá vyhodnocení jeho výrazů nad modelem, který strukturálně odpovídá uvedenému diagramu a používá stejné pojmenování prvků.

Co bychom si měli na uvedené příkladu dále všimnout, je vztažení jednotlivých omezení vždy k nějaké vlastnosti, tzv. kontextu. Specifikace kontextu je nutná zejména při používání duplikovatelných vlastností. Podívejme se na obrázek 2.13 znázorňující model vlastností a jeho dvě odlišné konfigurace. Pro tento model definujeme dvě různá omezení[20] a podíváme se jak budou vyhodnocena:



(a) Model vlastností. (b) Příklad konfigurace 1. (c) Příklad konfigurace 2.

Obrázek 2.13: Příklad modelu vlastností se dvěma různými konfiguracemi.

1. `C implies D`
2. `context B: C implies D`

První omezení implikuje nutnost vybrání vlastnosti D, pokud byla vybrána vlastnost C. V tomto případě jsou obě dvě konfigurace zcela validní. Druhé omezení říká to samé, avšak jeho vyhodnocení probíhá vždy v rámci společného rodiče B. Zde již konfigurace 2.13c nevyhovuje. Ekvivalentně bychom mohli tato omezení zapsat i jako výrazy predikátové logiky 2.1 a 2.2, kde X představuje vlastnost stejného jména a predikát $P(r, p)$ vyjadřuje vztah rodič-potomek.

$$\exists C \rightarrow \exists D \tag{2.1}$$

$$\forall B\ (\exists C \wedge P(B, C) \rightarrow \exists D \wedge P(B, D)) \tag{2.2}$$

Zajímavým pozorováním je, že pomocí výrokové logiky můžeme vyjádřit i lokální omezení, definovaná grafickou notací v diagramu vlastností, jak ukazuje tabulka 2.6[21]. Obdobným způsobem bychom mohli vyjádřit i celý model vlastností. V takovém případě by už pak rozdíl mezi globálními a lokálními omezeními nebyl patrný.

Na závěr této podkapitoly ještě dodejme, že ačkoliv můžeme textově zapsaná omezení zřejmě považovat za nejflexibilnější možnost vyjádření vztahů v modelu vlastností, nejde o jediný možný způsob. Někteří autoři zanášejí (globální) omezení do diagramu v grafické podobě, což má zajímavé důsledky pro jeho strukturu. Tyto odlišné reprezentace budou diskutovány v části 2.10.

[20] Pro jednoduchost jsme použili námi smyšlený jazyk podobný OCL.

[21] Hodnotu dané logické proměnné chápeme jako pravda, pokud je odpovídající vlastnost v konfiguraci přítomna.

Prvek diagramu vlastností	**Význam**
r je kořen diagramu	r
f_1 je volitelná pod-vlastnost f	$f_1 \rightarrow f$
f_1 je povinná pod-vlastnost f	$f_1 \leftrightarrow f$
$f_1, f_2, \ldots, f_n$ tvoří OR skupinu pod-vlastností f	$\bigvee_{i=1}^{n} f_i \leftrightarrow f$
$f_1, f_2, \ldots, f_n$ tvoří XOR skupinu pod-vlastností f	$(\bigvee_{i=1}^{n} f_i \leftrightarrow f) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$

Tabulka 2.6: Vyjádření některých vztahů z diagramu vlastností pomocí výrokové logiky.

2.9 Další rozšíření modelu vlastností

Až doposud jsme zmínili jen některá rozšíření (atributy, skupiny, kardinality) původního konceptu modelu vlastností, tak jak byl definován v [Kang90]. Tato podkapitola ve stručnosti shrnuje i některá další rozšíření popsaná v [Cza04].

2.9.1 Modularizace

Diagram vlastností popisující variabilitu komplexního systému může být velmi rozsáhlý. V takovém případě má význam ho rozdělit na několik menších celků (podstromů), které jsou spravovány nezávisle. V původním diagramu se pak vyskytuje speciální typ uzlu, který na tyto pod-diagramy odkazuje[22]. Principiálně se můžeme na ten samý podstrom odkazovat i ve více místech jednoho či několika diagramů.

Příklad modelu vlastností používající takto popsanou referenci jsme již mohli vidět na obrázku 2.12. Pozorný čtenář si zde jistě všiml odlišné notace s vyplněnou šipkou pro uzel *Catalog*, který reprezentuje právě zmíněný odkaz. V našem případě se zde odkazujeme na model vlastností popisující variabilitu katalogu zboží.

Použití referencí na jednotlivé prvky diagramu může samozřejmě přinést problém rekurze, a to jak přímé (diagram vlastností odkazuje na svojí část), tak nepřímé (vzájemné odkazy mezi dvěma a více diagramy).

2.9.2 Specifikace vztahu

V diagramu vlastností, tak jak jsme ho zde popsali, jsme pomocí spojení určovali pouze vztah mezi rodičem a potomkem, avšak nijak přesně jsme nedefinovaly význam tohoto vztahu. Někteří autoři, jako například [Lee02], rozlišují až čtyři různé typy *vztahů* (z angl. *relationships*):

1. *skládá-se-z* (*composed-of*) - Vyjadřuje vztah celku a jeho části, viz příklad řídícího systému a snímačů na obr. 2.8c. V diagramu je znázorněn plnou čarou.
2. *zobecnění/upřesnění* (*generalization/specialization*) - Vyjadřuje vztah mezi obecnou a konkrétní vlastností. Příkladem může být vztah mezi vlastnostmi *řadící algoritmus*

[22]Zde se nejedná o atribut typu reference, nebot' ten odkazuje na prvek konfigurace, ne prvek diagramu vlastností.

a *algoritmus quick sort*. V diagramu je znázorněn tečkovanou/čárkovanou čarou.

3. *je-implementován* (*implemented-by*) - Udává, že jedna vlastnost je nezbytná pro implementaci druhé. V diagramu je znázorněny tučnou čarou.

Zajímavé je, že poslední zmíněný vztah může zapříčinit to, že struktura výsledného diagramu nebude strom, ale acyklický graf, v případě že daná vlastnost potřebuje ke své implementaci více odlišných vlastností (má v diagramu více rodičovských uzlů).

2.9.3 Kategorie vlastností a anotace

Původní publikace [Kang90] rozdělovala jednotlivé vlastnosti do kategorií podle jejich významu. Vlastnosti pak byly podle toho označovány jako *funkcionální* (služby aplikace), *operační* (interakce uživatele s aplikací) a *prezentační* (prezentace informace uživateli). Někteří autoři [Griss98] toto schéma rozšiřují o další kategorie vlastností, jako například *architektonické* (souvisí se strukturou systému) a *implementační*.

Jiné rozšíření spočívá v přidání dalších doplňujících informací do modelu vlastností, dle autorů [Cza04], označovaných jako anotace. Původní publikace [Kang90] zmiňovala pouze popis vlastností, definici omezení, zdůvodnění použití a čas vazby. Příklady dalších takových anotací jsme si již uvedli v podkapitole 2.3.3.

2.10 Odlišné reprezentace modelu vlastností

Až doteď jsme pro vyjádření modelu vlastností používali diagram s notací autorů Czarnecki, Eisenecker (a kol.) [Cza00], [Cza05b] a krátce před tím jsme představili i původní FODA notaci [Kang90]. Nicméně, v oblasti modelování vlastností vznikli i další odlišné reprezentace od různých autorů, většinou využívající diagram založený právě na původní FODA notaci. Cílem této podkapitoly je seznámit čtenáře s jejich průřezem.

Poměrně rozsáhlý souhrn a srovnání různých notací diagramů vlastností a jejich sémantiky nalezneme v [Scho06]. Autoři těchto notací popisují celkem sedm, včetně FODA notace (zde označené jako OFT) a notace autorů Czarnecki, Eisenecker (označené jako GPFT). Dalšími popsanými notacemi jsou pak OFD, RFD, VBFD, EFD a PFT[23].

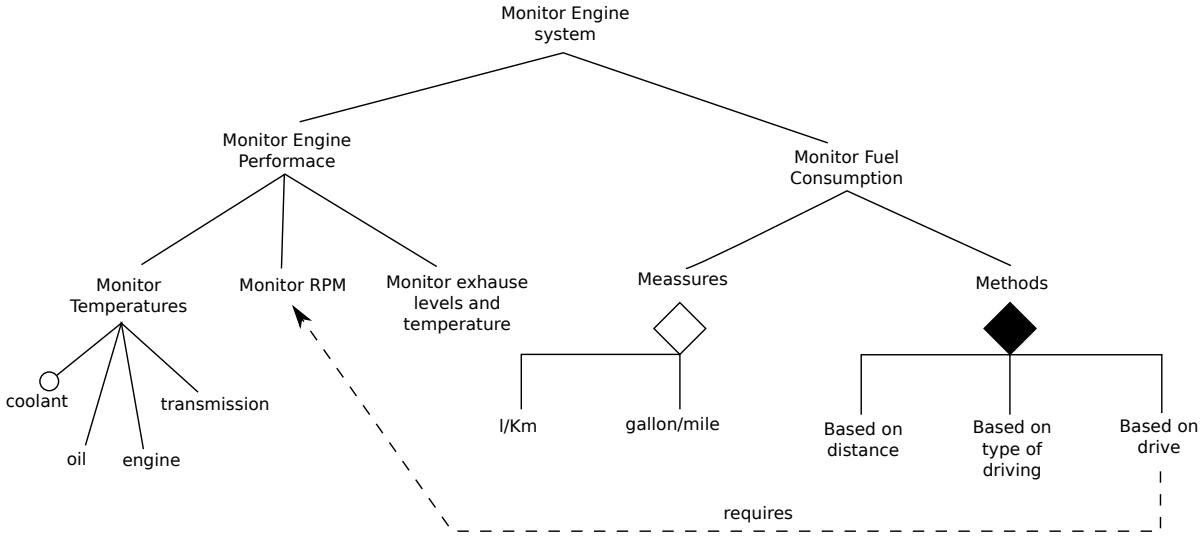
Všechny zmíněné mají společné to, že používají jako základ stromový diagram, ale většinou s různou grafickou úpravou a odlišnou sémantikou prvků. Vyjmenujme zde proto jen nejzásadnější rozdíly mezi těmito notacemi:

1. *Struktura diagramu* - Některé notace používají klasickou stromovou strukturu (OFT, GPFT, PFT), zatímco jiné používají acyklický graf (OFD, RFD, VBFD, EFD). V druhé uvedené skupině tedy může mít například jedna vlastnost dva rodičovské prvky.

[23]Tyto zkratky byly v [Scho06] zavedeny pro snadné rozlišení jednotlivých notací. Pro jejich přesný význam odkážeme čtenáře na zmíněnou publikaci.

2. *Typ uzlů a vazeb* - Všechny notace mají prvek s významem povinné a volitelné vlastnosti, i když pro něj někdy používají odlišnou grafickou reprezentaci. Dále všechny notace umožňují vyjádřit omezení nad množinou vlastností se sémantikou logické funkce *and*, *or*, *xor* nebo jiné. Grafické znázornění je někdy opět odlišné. Některé (GPFT, EFD) pro toto používají kardinalitu.

3. *Jazyk omezení* - OFT, OFD, GPFT používají k vyjádření omezení nějaký textový jazyk, zatímco PFT používá pro (globální) omezení grafickou notaci, kdy jsou jednotlivé vlastnosti napříč stromem propojeny hranou s vyznačenou sémantikou. Zbylé RFD, VBFD, EFD umožňují omezení definovat jak textově, tak graficky. Všechny notace umožňují nějakým způsobem vyjádřit alespoň dva standardní typy omezení (závislost a vyloučení).



Obrázek 2.14: Ukázka diagramu vlastností používající notaci RFD.

Některé notace zavádějí poměrně specifické prvky. Například OFD rozděluje diagram do vrstev dle typu vlastností (funkční, operační, technologické, implementační) a používá různě znázorněné typy vazeb (skládá se z, zobecnění/upřesnění, je implementován). Jiné, jako VBFD, umožňují označit vazby mezi vlastnostmi časem spojení (binding time). Aby čtenář získal alespoň přibližnou představu, uveďme zde ukázku diagramu používající notaci RFD (obr. 2.14), na kterém můžeme vidět i graficky vyznačené (globální) omezení. Tento diagram, znázorňující variabilitu monitorovacího systému, byl převzat ze zmíněné publikace [Scho06].

Výše uvedené notace používali k popisu modelu vlastností grafický diagram v podobě stromu či acyklického grafu, avšak to nemusí být vždy jediný možný způsob. Publikace [Bouch10] uvádí několik nevýhod použití diagramů, jako například jejich špatnou škálovatelnost (orientace v rozsáhlých diagramech) a to že grafická notace neumožňuje vyjádřit komplikovanější omezení. Její autoři proto přicházejí s návrhem TVL[24], čistě textově založeným jazykem pro popis modelů vlastností. Pomocí syntaxe tohoto jazyka lze

[24]Text-Based Variability Language.

popsat modely vlastností obsahující většinu popsaných rozšíření, jako například kardinalitu a atributy. Jazyk dále obsahuje některé zajímavé prvky, jako je výčtový typ pro atributy, možnost definice konstant, používání aritmetických a logických výrazů či dekompozici stromu na znovupoužitelné části. Popsaný model může mít strukturu acyklického grafu. Omezení jsou vyjádřena jako logický výraz přiřazený k vlastnostem. Syntaxe TVL je silně inspirovaná jazykem C, jak můžeme vidět na níže uvedené ukázce ve výpisu 2.2 (jde o vyjádření diagramu vlastností z obr. 2.8c).

```
RidiciSystem {
    group allOf {
        Snimac [1..4] {
            group [1..2] {
                SnimacPolohy;
                SnimacPohybu;
                SnimacTeploty;
            }
            opt Autodiagnostika;
        }
        CPU {
            int VelikostPameti;
            VelikostPameti >= 0;
        }
    }
}
```

Výpis 2.2: Příklad modelu vlastností v jazyce TVL.

3 Dostupné modelovací nástroje

Cílem této kapitoly je seznámit čtenáře s některými nástroji umožňujícími modelování vlastností. Účelem zde není popsat všechny dostupné nástroje, ale spíše jejich průřez, zachycující různé přístupy těchto nástrojů k samotnému modelování. Nástroje se zde liší zejména svým rozsahem, funkcionalitou, zaměřením, použitými technologiemi a způsobem ovládání.

Valnou část popsaných nástrojů tvoří aplikace, jež mají svůj původ vzniku na akademické půdě. Některé z nich slouží víceméně jako výzkumný prototyp pro podložení závěrů publikovaných jejich autory (*CaptainFeature*, *FMP*), zatímco jiné představují poměrně komplexní a propracované nástroje s reálnou možností praktického využití (*FeatureIDE*). Komerční aplikace (*pure::variants*) pak tvoří jen malou část ze všech dostupných nástrojů.

Na závěr této kapitoly bude provedeno srovnání těchto nástrojů z hlediska jejich funkcionality, ovládání, uživatelské přívětivosti a použitých technologií. Toto srovnání nám později poslouží jako výchozí bod pro definování požadavků na námi vytvářený modelovací nástroj.

3.1 CaptainFeature

CaptainFeature[1] je jeden z prvních modelovacích nástrojů, který umožňoval tvorbu modelů vlastností s kardinalitou i atributy. Model vlastností je zde uvažován stejným způsobem jako v [Cza05a], včetně chápání kardinalit jako posloupnosti intervalů. Samotný nástroj je napsán v jazyce Java s použitím knihovny SWING.

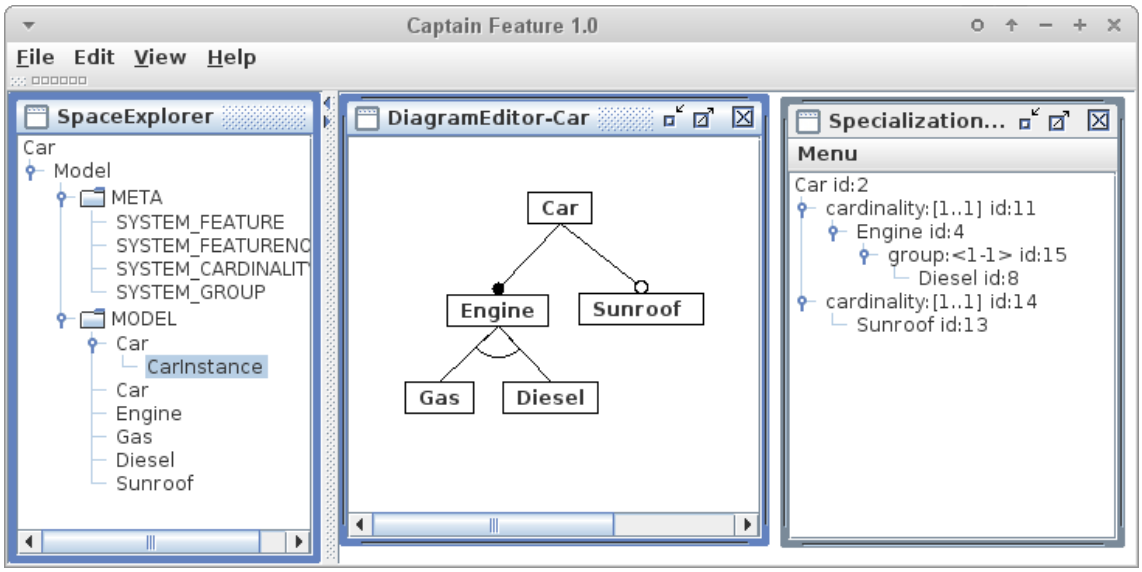
Model vlastností je vytvářen graficky, v podobě jednoho či více diagramů. Nástroj podporuje specializaci diagramů, která zároveň slouží zároveň i k vytváření jejich konfigurace. Pro definování omezení je k dispozici vlastní jazyk CFCL. Zajímavostí je možnost prohlédnout si použitý meta-model[2], který je sám znázorněn jako diagram vlastností.

Nástroj představuje výzkumný prototyp a jeho grafické rozhraní je tak velmi minimalistické, avšak umožňuje všechny základní operace nad modelem. Tyto operace jsou však přístupné většinou pouze přes kontextové menu, což není příliš uživatelsky přívětivé. Bez studia přiloženého manuálu je navíc některé funkce obtížné vůbec dohledat. Specializace modelu se provádí ve stromovém editoru a je poměrně nepřehledná. Nástroj se v této době již nevyvíjí.

3.2 Feature Modeling Plug-in

Feature Modeling Plug-in[3] (FMP) je zásuvný modul pro vývojové prostředí Eclipse[4], umožňující vytváření, editaci, konfiguraci i specializaci modelů vlastností s kardinalitou a

[1] `http://sourceforge.net/projects/captainfeature/`
[2] Meta-model definuje jak vypadá struktura daného modelu, v našem případě modelu vlastností.
[3] `http://gp.uwaterloo.ca/fmp/`
[4] `http://www.eclipse.org/`



Obrázek 3.1: Ukázka práce s nástrojem CaptainFeature.

atributy, tak jak byly uvedeny v [Cza05b]. Tento nástroj, popsaný v [Ant04], představuje výzkumný prototyp a jeho vývoj byl již ukončen.

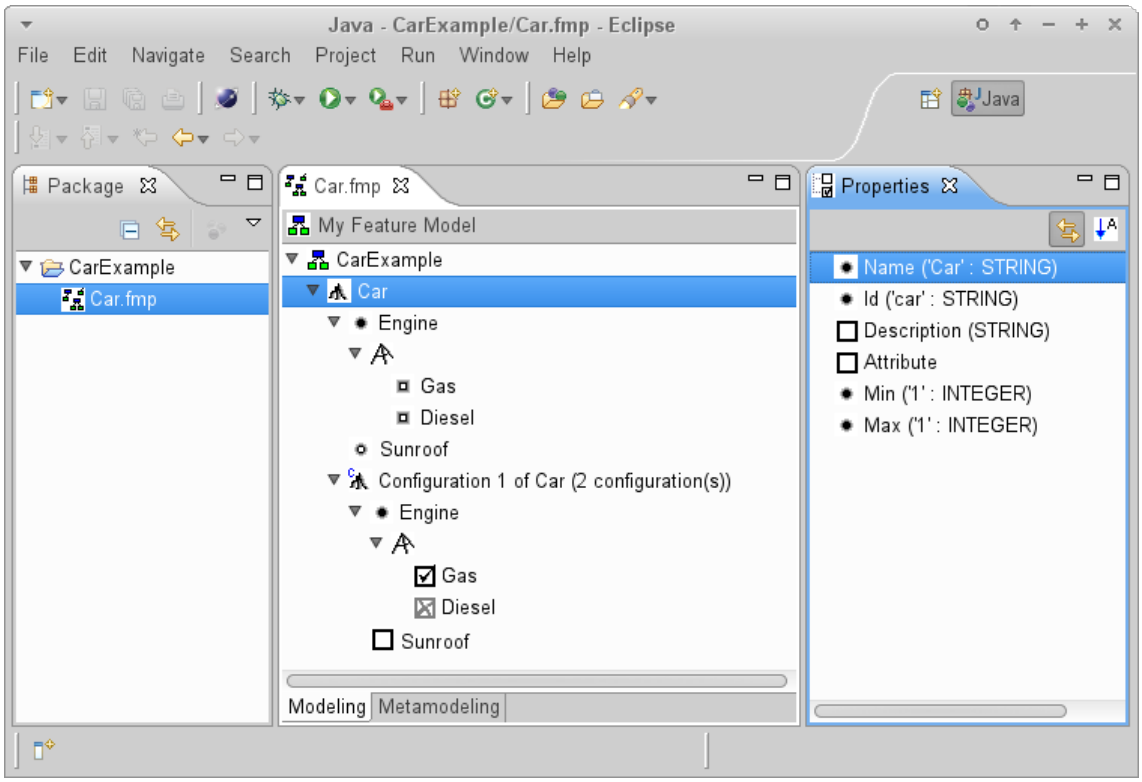
Pro editaci je použit klasický stromový editor, takže je i editování rozsáhlých modelů (do šířky) poměrně přehledné. Model se dá navíc dekomponovat do samostatných celků, na které se dá odkazovat pomocí referencí. Jako jazyk omezení byl vybrán XPath[5], což umožňuje definovat poměrně komplexní omezení, včetně aritmetických a logických operací nad atributy. Při vytváření konfigurace pak uživatel pouze zatrhává chtěné vlastnosti ve stromě. Editor mu u toho dopomáhá blokováním neplatných voleb a zobrazením počtu zbylých validních konfigurací. Stejně jako u Captain Feature, i zde je možné si prohlédnou použitý meta-model, avšak oproti Captain Feature je zde možno tento meta-model dále rozšiřovat a přidávat tak do modelu vlastností nové prvky.

Zásadním problémem FMP však je, že samotná konfiguraci i specializace probíhá v tom samém stromě jako editace modelu (duplikováním a upravováním jeho části), což celý tento proces znepřehledňuje. Většina akcí je, podobně jako u Captain Feature, prováděna přes kontextové menu, které je tak bohužel poměrně rozsáhlé. Zápis omezení v jazyce XPath je sice efektivní, ale některé rozsáhlé výrazy mohou být pak ne příliš čitelné. Uživatelům by navíc mohla chybět možnost zobrazit si výsledný digram v grafické podobě.

3.3 XFeature

Autoři nástroje XFeature[6] zvolili poměrně inovativní přístup k reprezentaci modelu vlastností, popsaný v [Cech04]. Jak model vlastností, tak i jeho konfiguraci zde chápeme jako XML dokumenty se strukturou vyhovující jistému XML schématu (jejich meta-modelu). Pro konfiguraci vlastností je toto schéma generováno XSLT transformací z odpovídajícího

[5] `http://www.w3.org/TR/xpath20/`
[6] `http://www.pnp-software.com/XFeature/`



Obrázek 3.2: Ukázka práce s nástrojem Feature Modeling Plug-in.

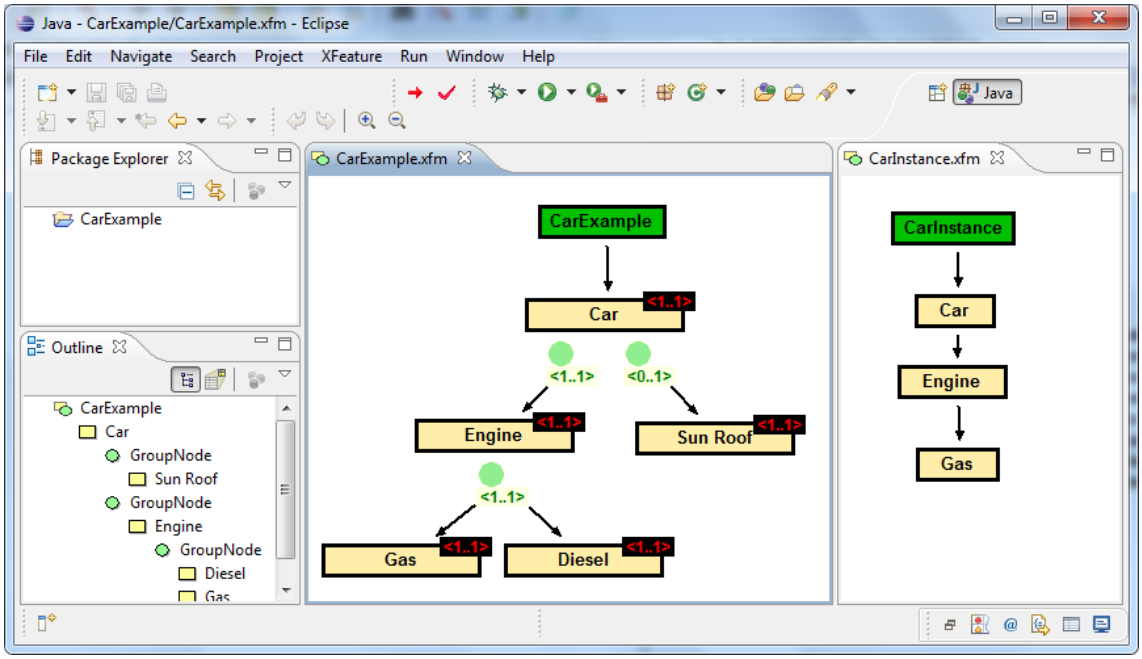
modelu vlastností. Schéma pro samotný model vlastností však pevně dáno není. Uživatelé si mohou toto XML schéma vytvářet sami (musí vyhovovat pevně danému XML schématu vyšší úrovně – meta-meta-modelu), nebo mohou použít již nějaký existující dostupný meta-model. Definice omezení pak probíhá pomocí již zmíněného jazyka XPath.

Největší výhodou celého popsaného přístupu je právě možnost tvorby vlastního meta-modelu, dle potřeb uživatele. Další výhodou je také velice jednoduše prováděná validace (konfigurace i modelu), jež spočívá ve validování vůči danému XML schématu a vyhodnocení výrazů v jazyce XPath (omezení). Samotní autoři svůj návrh podporovali existencí velkého množství XML nástrojů, které uživatelům usnadní vytváření modelu.

Technický popis konceptu nástroje XFeature lze nalézt v [Pas05]. Jádro tohoto nástroje tvoří pouze soubor XML schémat a XSLT transformací, které se dají použít samostatně, nicméně z důvodu uživatelské přívětivosti byla nad tímto jádrem vytvořena grafická nadstavba v podobě zásuvného modulu pro vývojové prostředí Eclipse. Uživatel zde nejprve vytvoří soubor s popisem nastavení projektu (výběr meta-modelu apod.) a k němu poté vytváří modely vlastností (zde označované jako *family model*) a jejich konfigurace (zde označované jako *application model*). Samotná tvorba modelu probíhá editací stromového diagramu v grafickém editoru, prakticky výhradně s použitím kontextového menu, a dodatečně i v příslušném editačním panelu. Při konfiguraci uživatel opět vytváří stromový diagram, avšak jsou mu nabízeny pouze volby validní dle daného modelu vlastností (podobně jako u FMP).

Celé řešení má však několik nedostatků, jako například editace modelu pomocí kontex-

tového menu, jež je poměrně nepřehledné a zpomaluje při práci. Dané grafické ztvárnění diagramu pak neodpovídá žádné z používaných notací. Editor umožňuje přiřazovat vlastnostem atributy pouze přes různé předdefinované sestavy (*attribute set*), což může být flexibilní, avšak pro uživatele také zbytečně komplikované a zdržující. Při validování modelu jsou uživateli zobrazeny výpisy z přiloženého XML validátoru, které nejsou vždy úplně srozumitelné. Zmiňovaná možnost vlastního meta-modelu nemusí být v každém případě výhodou, neboť jeho tvorba není nijak triviální.



Obrázek 3.3: Ukázka práce s nástrojem XFeature.

3.4 Software Product Lines Online Tools

Software Product Lines Online Tools[7] (SPLOT) je nástroj se záměrem přenést výzkum v oblasti modelování vlastností blíže k praktickému využití. Tento nástroj je, poměrně netradičně, implementován jako webová aplikace, což mu přináší řadu zajímavých výhod, jako například online editaci modelu vlastnosti s možností následného uložení do databáze. Jednotlivé modely mohou navíc uživatelé mezi sebou sdílet.

Samotný (stromový) editor má lehce omezené schopnosti (pouze povinné/volitelné vlastnosti a OR/XOR skupiny), avšak jeho ovládání je velice intuitivní a přehledné. Zajímavě je řešeno definování omezení jako Booleovských formulí (ve formě disjunktivních klauzulí), kdy uživatel nejprve dosadí vybrané vlastností do formule jako literály a následně pak může některé označit negací. Tento jazyk není sice tak komplexní jako např. XPath, avšak dokáže ho poměrně rychle zvládnout i nezkušený uživatel.

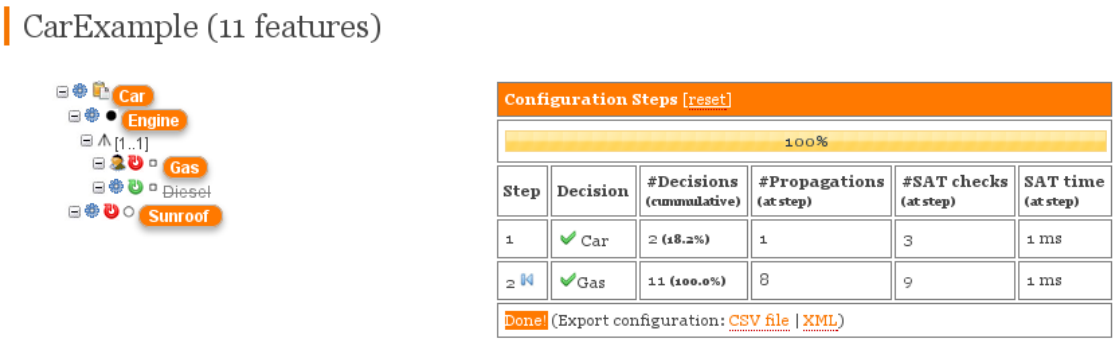
Konfigurace probíhá, podobně jako u FMP, zatrháváním vlastností ve stromě s automa-

[7] `http://www.splot-research.org/`

tickým blokováním nevalidních voleb. Editor zde navíc uživateli dokáže ukázat vlastnosti jejichž výběr je v rozporu s jinými a automaticky pak konfiguraci opravovat či doplňovat.

Na straně serveru je pak tato aplikace implementována pomocí technologie Java EE v podobě 2 celků: SPLOT (jádro webové aplikace) a SPLAR (knihovna algoritmů pro vyhodnocování modelů vlastností).

Obrázek 3.4: Ukázka práce s nástrojem SPLOT (editace modelu).



Obrázek 3.5: Ukázka práce s nástrojem SPLOT (konfigurace).

3.5 FeatureIDE

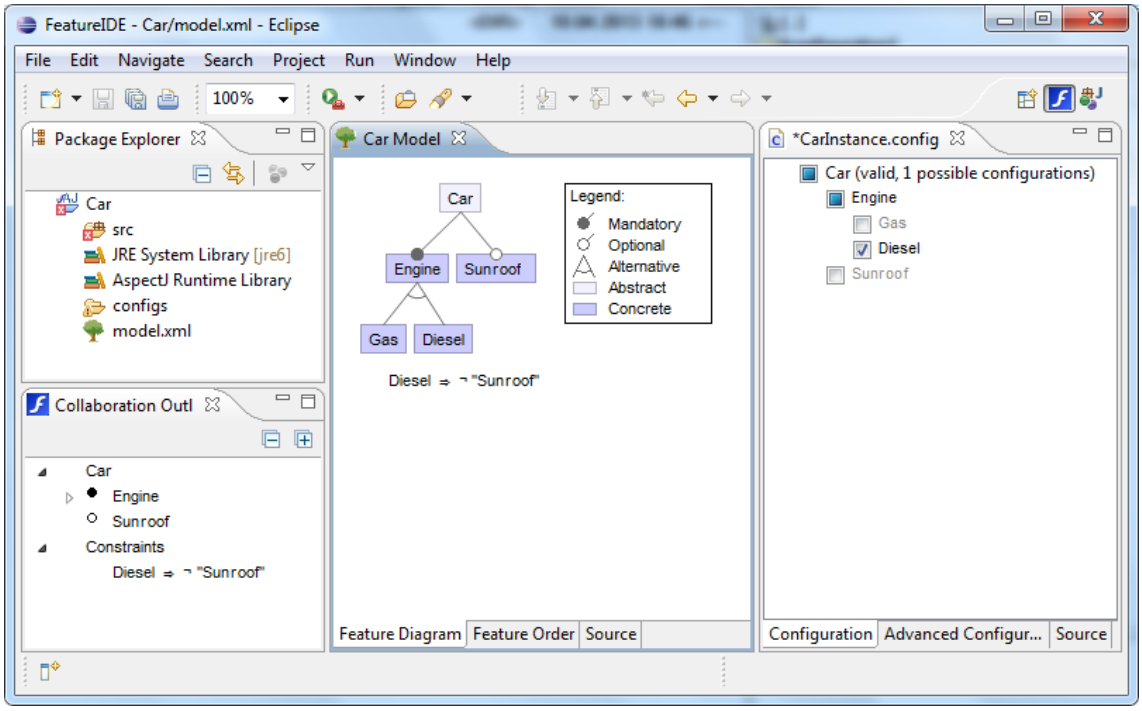
FeatureIDE[8] můžeme dle jeho autorů [Thum12] popsat jako open-source framework pro pro vývoj software postavený na modelování vlastností[9]. Tento nástroj je implementován jako sada zásuvných modulů vývojového prostředí Eclipse, jež jsou pak s tímto prostředím distribuovány jako jeden celek.

Hlavním rozdílem oproti předchozím příkladům je však zaměření samotného nástroje, který se neorientuje jen pouze na fázi doménové analýzy, ale snaží se pokrýt celý proces doménového inženýrství (viz obr. 2.2). Jinými slovy, mimo samotné vytváření modelů vlastností a jejich konfigurace je také podporováno:

[8] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
[9] Poněkud nepřesný překlad pojmu *Feature-oriented software development.*

1. Mapování vlastností na jednotlivé zdrojové části aplikace.
2. Sestavení výsledné aplikace dle vybraných vlastností (konfigurace).

Aby bylo možné tyto dva kroky zajistit, bylo FeatureIDE integrováno s celou řadou již dostupných nástrojů[10], jako je AHEAD, FeatureHouse, FeatureC++, DeltaJ, AspectJ, Munge a Antenna. Podporu pro další nástroje je pak možné přidat skrze rozšíření. Nástroj nejen že podporuje editaci zdrojových kódů jazyků výše zmíněných nástrojů, ale integruje tyto nástroje s modely vlastností v jeden celek, takže je pro uživatele například možné určit, která část zdrojového kódu odpovídá implementaci které vlastnosti a naopak.



Obrázek 3.6: Ukázka práce s nástrojem FeatureIDE.

Zaměřme se ale zpět na oblast modelování vlastností. FeatureIDE umožňuje vytváře modely vlastností pomocí grafické editace diagramů. U modelu vlastností jsou podporovány pouze povinné/volitelné vlastnosti[11] a OR/XOR skupiny, kde každá vlastnost může mít navíc pod sebou maximálně jednu skupinu. Samotná práce s editorem je velmi rychlá a pohodlná, ačkoliv probíhá výhradně přes kontextové menu, s případným ručním přetažením uzlu pod jiného rodiče. Uživatel zde má navíc možnost použít pro strom diagramu jedno z automatických rozvržení místo manuálního pozicování uzlů. Zajímavá je podpora přímé editace zdrojového kódu modelu (XML dokumentu), kdy jsou jeho změny automaticky promítány do diagramu a naopak.

[10] AHEAD, FeatureHouse a FeatureC++ jsou nástroje či jazyky zaměřené na tzv. *Feature-oriented programming*. DeltaJ je rozšíření jazyka Java pro tzv. *Delta-oriented programming*. AspectJ je rozšíření jazyka Java pro aspektově orientované programování. Munge a Antenna jsou preprocesory zdrojového kódu jazyka Java. Pro detailnější informace odkážeme čtenáře na [Thum12].

[11] Některé vlastnosti je možno také označit jako skryté (nezobrazují se), či abstraktní (nemají v kódu žádnou implementaci).

Pro editaci omezení nabízí FeatureIDE vlastní jazyk postavený na Booleovské algebře. Editor těchto omezení podporuje napovídání a automatické doplňování vhodných možností. Daná omezení jsou pak zobrazena jako součást diagramu. Proces konfigurace je velmi podobný předchozím nástrojům (FMP, SPLOT), kdy jsou uživateli automaticky blokovány nevyhovující možnosti a je mu ukazován zbývající počet možných konfigurací. Uživatel má navíc možnost zapnout si pokročilý mód konfigurace, kdy může některé vlastnosti z konfigurace explicitně vyřadit. Důležité je také zmínit možnost importu či exportu modelů z jiných nástrojů (FMP, SPLOT, SPLConqueror, GUIDSL). Pro popis ostatních vlastností FeatureIDE odkážeme čtenáře na [Thum12].

3.6 pure::variants

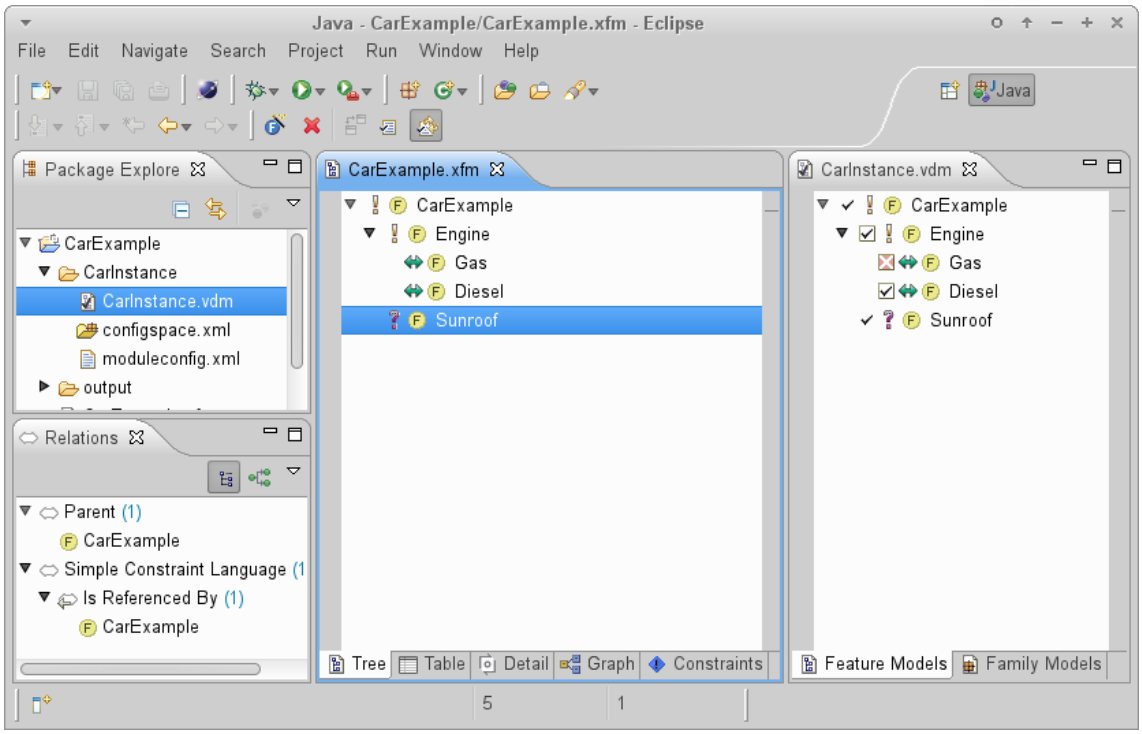
Pure::variants představuje jedno z mála komerčních řešení pro modelování variability software. Tento nástroj se, podobně jako FeatureIDE, nezaměřuje pouze na samotné modelování vlastností, ale snaží se pokrýt všechny fáze vývojového procesu software. Nástroj umožňuje pracovat celkem se čtyřmi různými modely:

1. *Feature model* - Model vlastností.
2. *Family model* - Definuje architekturu rodiny produktů (její logické i fyzické elementy) a mapování jednotlivých vlastností na tuto architekturu.
3. *Variant description model* - Konfigurace vlastností.
4. *Variant result model* - Výstupní model popisující konkrétní aplikaci z rodiny produktů a obsahující informace pro její sestavení.

Pro každý z těchto uvedených modelů (s výjimkou posledního, jež lze pouze zobrazit) je k dispozici samostatný editor. Editace modelu vlastností probíhá v klasickém stromovém editoru s použitím kontextového menu, jež obsahuje všechny základní funkce a je poměrně přehledné. Uživatel může vytvářet celkem 4 typy různých vlastností (povinné, volitelné, alternativní a slučitelné), kde u poslední uvedené varianty lze v rámci skupiny volitelně nastavit i její kardinalitu. Hlavním rozdílem oproti ostatním nástrojům je zde možnost přiřadit k vlastnostem poměrně velké množství dodatečných informací, jako je například stav, verze, priorita, popis, zdůvodnění, datum vytvoření, autor apod. Jedné vlastnosti lze také přiřadit neomezené množství atributů, jejichž typ není omezen pouze na základní datové typy, ale může jít například o datum, prioritu, URL, odkaz na jinou část modelu atd. Editor navíc podporuje i alternativní zobrazení modelu vlastností jako grafové struktury, avšak pouze s omezenými možnostmi editace.

Zajímavý je přístup nástroje ke globálním omezením, která jsou rozdělena do dvou skupin: omezení (*constraints*) a tzv. restrikce (*restrictions*). První uvedená skupina slouží definování globálních podmínek integrity celé konfigurace, druhá pak slouží k definování podmínek za jakých lze danou vlastnost v konfiguraci vybrat (restrikce musí být tedy přiřazena vždy k určité vlastnosti). Pro zápis obou typů omezení lze použít buď dialekt jazyka prolog *pvProlog*, nebo jazyk vycházející z Booleovské algebry nazvaný *pvSCL*, kde druhý zmiňovaný pak podporuje nejen logické, ale relační operace pro porovnání atributů.

Oba dva jazyky lze navíc použít k výpočtu počátečních hodnot atributů. Mimo popsaná omezení lze mezi jednotlivými vlastnostmi definovat také přímý vztah (*relation*), jako například *vyžaduje* či *je v konfliktu s*.



Obrázek 3.7: Ukázka práce s nástrojem pure::variants.

Tvorba konfigurace probíhá, jako u některých předchozích nástrojů, zatrháváním či vyřazováním vlastností ve stromu. Nástroj pure::variants nabízí ze všech zkoumaných nástrojů pravděpodobně nejlepší asistenci uživateli při tvorbě konfigurace, včetně blokování neplatných voleb, zvýrazňování chyb a možnostmi automatické opravy či automatického doplňování. Navíc lze jednotlivé konfigurace od sebe navzájem odvozovat (dědí pak mezi sebou vybrané vlastnosti). Výsledný popis sestavené aplikace (*variant result model*) je pak možné z konfigurace vytvořit transformačním procesem, který je buď standardní (odvozen z informací z modelů) nebo uživatelsky definovaný (vyjádřený jako XSLT transformace, nebo pomocí jazyka JavaScript).

Mimo tyto základní editory je také k dispozici, již zmíněný, *family model editor* (hierarchický popis architektury s mapováním vlastností), maticový editor (zobrazení několika konfigurací ve sloupcích oproti vybraným vlastnostem v řádcích) či *compare editor* (vizuální porovnání dvou modelů vlastností). Pure::variants dále nabízí několik pomocných pohledů (*views*), například pro zobrazení atributů a vztahů mezi vlastnostmi, či pro zobrazení výsledné konfigurace.

Pro integraci s ostatními nástroji je k dispozici import a export modelů v různých datových formátech (CSV, XML, HTML, DOT[12]). Mimo to však pure::variants obsahuje

[12]Jazyk nástroje GraphViz (http://www.graphviz.org/), který umožňuje vizualizovat grafové struktury. Zde slouží k zápisu stromové struktury modelu vlastností.

i přímou integraci do některých jiných komerčních nástrojů, jmenovitě například IBM Rational DOORS/ClearQuest/Rhapsody, či Enterprise Architect. Popis dalších vlastností nástroje pure::variants lze nalézt v [Pure13].

3.7 BigLever Software Gears

Gears od společnosti BigLever Software je dalším příkladem komerčního řešení pro vývoj produktových řad software. Obdobně jako pure::variants, i Gears pokrývá celý vývojový proces od fáze analýzy až po sestavení konkrétní varianty aplikace, avšak na rozdíl od pure::variants není Gears jeden samostatný nástroj, ale ucelený framework pro tvorbu produktových řad[13].

Základem Gears je vývojové prostředí nazvané *Software Product Line Development Environment*, jež slouží k návrhu produktové řady. Součástí takového návrhu je zejména vytváření modelů vlastností a jejich mapování na jednotlivé body variace (*variation points*), označující místa ve kterých se mohou jednotlivé varianty výsledných aplikací lišit. Příkladem takových míst jsou například složky, soubory, části textových souborů, nebo vybrané elementy UML modelů (označené pomocí daného stereotypu).

Vyhodnocení jednotlivých bodů variace je pak provedeno pomocí tzv. konfigurátoru, jež z různých vstupních fragmentů (požadavky, návrhy modelů, zdrojový kód, uživatelská dokumentace, testovací případy) vytváří obdobné fragmenty pro konkrétní variantu aplikace. To, které části fragmentů a způsob jakým budou do výsledné varianty zahrnuty, je definováno pomocí tzv. profilů (*feature profiles*), obsahujících danou konfiguraci vlastností.

Místo toho, aby Gears implementoval vlastní nástroje pro jednotlivé fázi vývoje, poskytuje integraci s již existujícími produkty, jako je například řada IBM Rational (DOORS, Rhapsody, Synergy, Quality Manager, Team Concert), či vývojová prostředí Eclipse a Microsoft Visual Studio[14]. Propojení s těmito nástroji je realizováno pomocí rozšíření frameworku Gears, tzv. mostů (*bridge*), jež mohou být buď jednostranné (implementované na straně Gears), či oboustranné (implementované i jako rozšíření na straně nástroje[15]). Gears navíc nabízí SDK, které umožňuje i ostatním vývojářů vytvořit most pro integraci Gears s jejich vlastními aplikacemi. Pro další informace odkážeme čtenáře na [Kru13].

3.8 KConfig

Všechny doteď představené nástroje byly poměrně univerzální, neboť umožňovaly modelovat variabilitu prakticky libovolného objektu. KConfig je ukázkou opačného přístupu, kdy je modelování variability zaměřeno pouze na jeden konkrétní předmět, v tomto případě jádro operačního systému Linux[16].

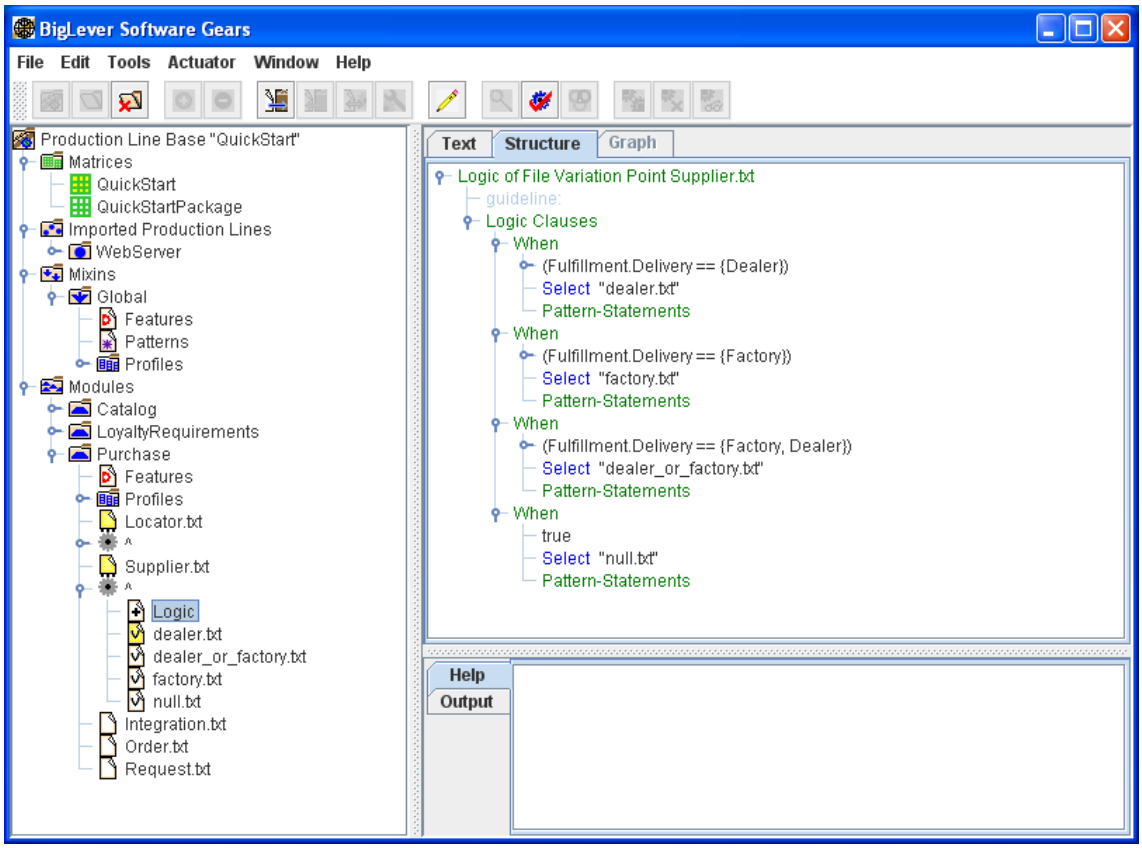
Na tomto místě je důležité čtenáře upozornit, že KConfig není ve své podstatě nástroj,

[13] Product Line Engineering Lifecycle Framework™, jak toto řešení nazývá přímo společnost BigLever.

[14] Na Gears se tak vlastně můžeme dívat jako na integrační framework pro vývoj SPL.

[15] To zejména v případě, že zpracovávané fragmenty jsou v nástroji interně uloženy v nějakém proprietárním formátu.

[16] `https://www.kernel.org/`



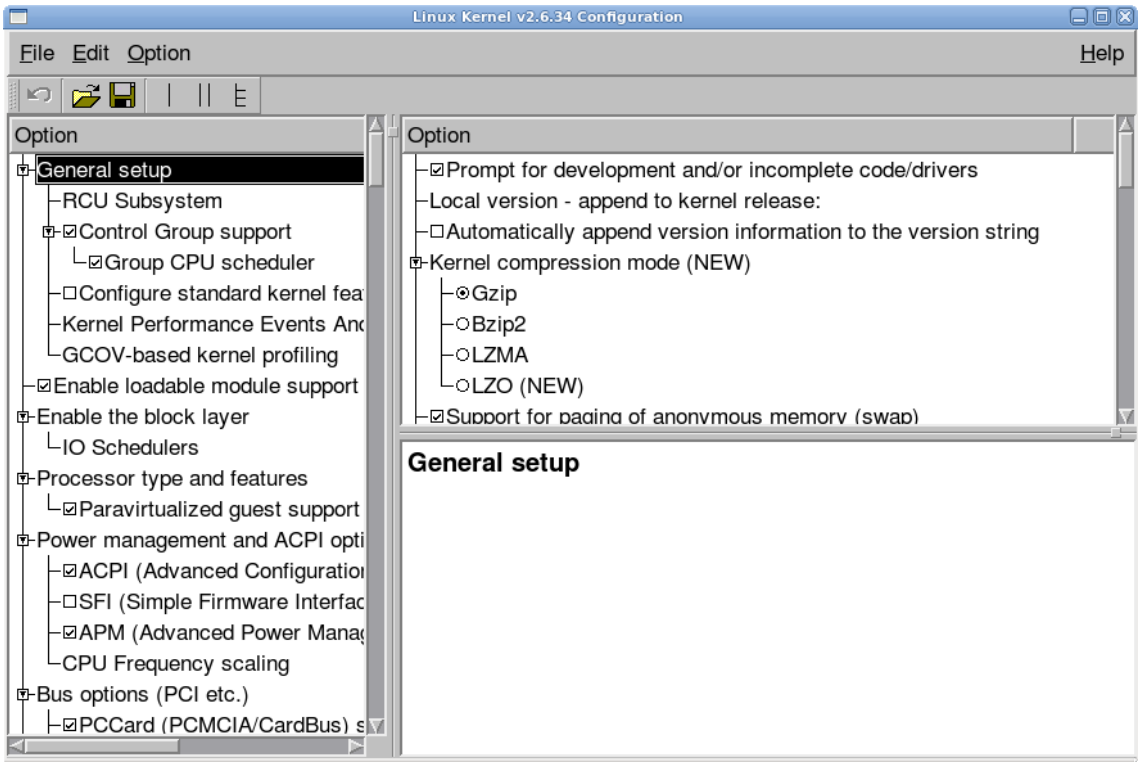
Obrázek 3.8: Ukázka práce s nástrojem Software Product Line Development Environment (součást frameworku Gears).

ale pouze textový jazyk[17], který slouží k popisu různých konfiguračních voleb určujících jak bude Linuxové jádro sestaveno. Nad konfigurační databází zapsanou v tomto jazyce pak uživatel může spustit jeden z dostupných konfiguračních nástrojů (např. *menuconfig* či *xconfig*, viz obr. 3.9) a vybrat v něm například jaké moduly budou do jádra zahrnuty.

Tento příklad zde uvádíme zejména proto, že jde o reálnou ukázku použití modelování variability v praxi. Konfigurační databázi jádra Linux navíc můžeme dle [She10] chápat jako na jeden z nejrozsáhlejších publikovaných modelů vlastností. Popis možností jazyka KConfig a jeho vazby na koncept modelování vlastností lze nalézt ve zmíněné publikaci [She10].

Samotný jazyk KConfig není využíván pouze v Linuxovém jádře, ale i u jiných projektů, jako BusyBox[18] či uClibc[19]. Kromě KConfig existují i jiné obdobně koncipované jazyky, například CDL, používaný pro vestavěný operační systém eCos[20].

[17] `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`
[18] `http://www.busybox.net/`
[19] `http://www.uclibc.org/`
[20] `http://ecos.sourceware.org/`



Obrázek 3.9: Ukázka práce s nástrojem xconfig.

3.9 Některé další nástroje

Mimo výše zmíněný výčet existují samozřejmě i další nástroje zaměřené na modelování vlastností, avšak většinou se svým rozsahem a zaměřením od těch zde zmíněných příliš neliší. Přesto si několik dalších pro přehled uvedeme.

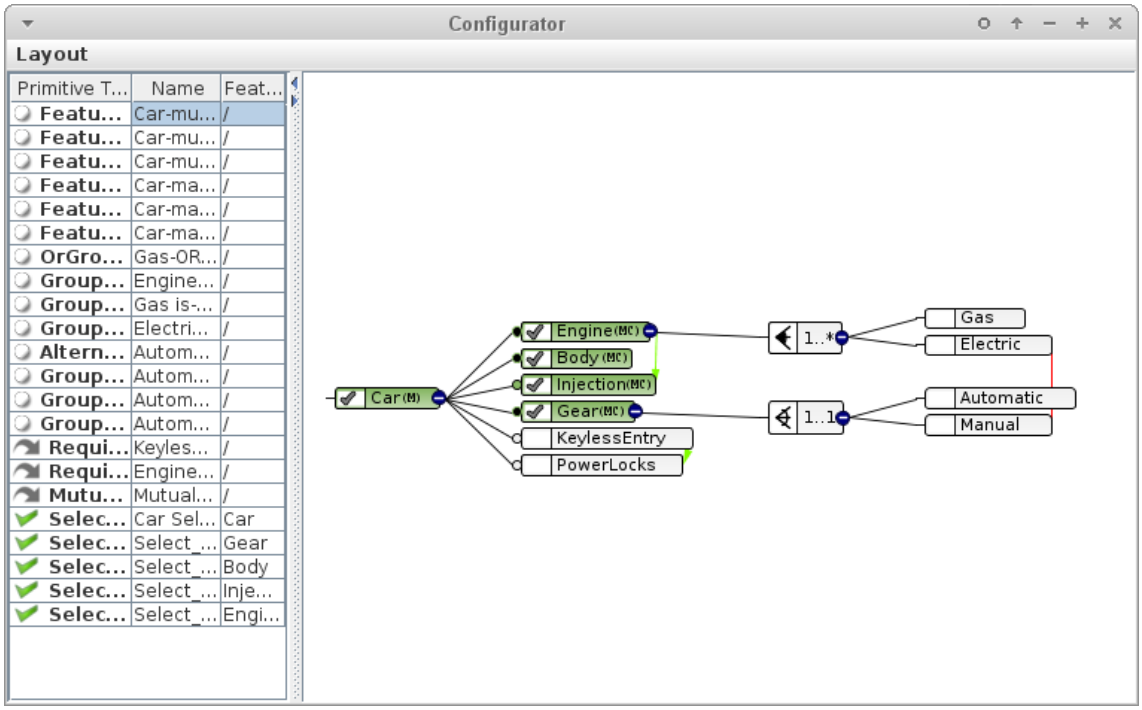
Mezi rozsáhlejší projekty zaměřené na vývoj produktových řad sofrware bychom mohli zařadit CIDE[21], vývojové prostředí, které umožňuje programátorovi anotovat (doslova v editoru vizuálně obarvit) kód implementující různé vlastnosti. Obdobným projektem je pak i Feature Commander[22], odlišující barevně zdrojový kód jednotlivých vlastností (identifikovaných makry preprocesoru jazyků C/C++).

Velká část nástrojů je, podobně jako FMP, implementována jako zásuvný modul do vývojového prostředí Eclipse. Zde můžeme kupříkladu jmenovat Feature Diagram Editor[23] nebo Feature Diagrams[24], jež se řadí k jednodušším nástrojům (prakticky umožňují pouze tvorbu diagramů vlastností).

Poměrně pokročilejší zásuvné moduly Eclipse jsou pak Compositional Variability Ma-

[21] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/`
[22] `http://wwwiti.cs.uni-magdeburg.de/~feigensp/xenomai/`
[23] `http://sdqweb.ipd.kit.edu/wiki/Feature_Diagram_Editor/`
[24] `http://sourceforge.net/projects/featurediagrams/`

nager[25] (framework pro modelování variability), Feature Mapper[26] (umožňující mapování vlastností na různé části EMF modelů a jejich vizualizaci), Hydra[27] (obsahující grafický editor diagramů spolu s textovým editorem omezení, validátorem a grafickým konfigurátorem), S2T2 Configurator[28] (obsahující interaktivní grafický konfigurátor, viz obr. 3.10) či MOSKitt Feature Modeler[29].



Obrázek 3.10: Interaktivní konfigurace s nástrojem S2T2 (starší verze 0.1.1).

Na tomto místě je důležité zmínit, že v rámci vývoje prostředí Eclipse existuje samostatný projekt EMF Feature Model[30], jehož cílem je vytvořit framework a editor pro modelování vlastností. Tento projekt je zatím ve stádiu počátečního vývoje.

Posledními nástroji, které zde ještě uvedeme, jsou pak Feature Modeling Tool[31] (rozšíření pro Microsoft Visual Studio) a Requiline[32] (umožňuje zároveň modelování vlastností i požadavků).

Na závěr ještě dodejme, že někteří autoři se nevydali cestou grafického modelování, ale používají k vyjádření modelu vlastností textový jazyk. Příkladem je například Clafer[33] (jazyk pro modelování problémové domény, umožňující unifikovaně popsat vlastnosti, třídy a

[25] `http://www.cvm-framework.org/`
[26] `http://featuremapper.org/files/`
[27] `http://caosd.lcc.uma.es/spl/hydra/`
[28] `http://download.lero.ie/spl/s2t2/`
[29] `http://www.moskitt.org/eng/proyectomfmmoskittfeaturemod/`
[30] `http://eclipse.org/proposals/feature-model/`
[31] `http://giro.infor.uva.es/FeatureTool.html`
[32] `http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline/`
[33] `http://www.clafer.org/`

meta-modely, spolu s jejich omezeními), nebo FAMILIAR[34] (umožňuje definici, manipulaci a slučování modelů vlastností).

3.10 Porovnání nástrojů

V této kapitole jsme si představili poměrně velké množství odlišných nástrojů. Pokusme se zde tedy uvést seznam jejich nejzásadnějších rozdílů, jež bychom měli brát v potaz při návrhu a implementaci našeho vlastního nástroje.

1. *Zaměření* - Většina nástrojů byla vytvořena jako univerzální (dokázaly modelovat vlastnosti libovolného objektu), avšak některé byly zaměřené na specifikou cílovou oblast (KConfig).

2. *Implementační technologie* - Velká část nástrojů byla implementována jako zásuvný modul do vývojového prostředí Eclipse (FMP, pure::variants, FeatureIDE, CIDE atd.), některé výjimečně i jako rozšíření pro MS Visual Studio (Feature Modeling Tool). Příkladem dalších implementačních technologií je například použití frameworku Qt s jazykem C++ (FeatureCommander) či použití frameworku .NET (Requiline).

3. *Meta-model*

   - Většina nástrojů podporuje modelování alespoň povinných/volitelných vlastností a OR/XOR skupin[35]. Některé nástroje (pure::variants) povolovaly specifikovat kardinalitu v rámci skupiny, jiné (FMP) pak i u vlastností. Zvláštním případem je pak CaptainFeature, který umožňoval uvést kardinalitu nejobecněji, jako posloupnost intervalů.
   - Většina nástrojů umožňovala definovat atributy, a to buď maximálně jeden pro každou vlastnost (FMP) nebo neomezené množství (pure::variants). Některé nástroje atributy vůbec neuvažovaly (FeatureIDE). Speciálním případem je pak XFeature, který umožňoval v rámci meta-modelu atributy rozdělovat do skupin.
   - Zatímco některé nástroje podporovaly pouze několik základní typů atributů, jako číslo, řetězec či reference (FMP), jiné nabízely i poměrně netradiční typy, jako kupříkladu datum nebo URL (pure::variants).
   - Různá byla také podpora dodatečných informací uchovávaných v rámci modelu. Dobrým příkladem je zde pure::variants, který umožňoval pro vlastnost specifikovat například prioritu, verzi, datum vytvoření a spoustu dalších.

4. *Rozšiřitelnost meta-modelu* - Většina nástrojů používala pevně daný meta-model, jež některé dokázaly uživateli i zobrazit (CaptainFeature). FMP jako jeden z mála umožňoval do meta-modelu přidávat dodatečné prvky a speciálním případem je pak XFeature s možností definice vlastního meta-modelu.

[34] `http://familiar-project.github.io/`

[35] Některé nástroje (FeatureIDE, pure::variants) používají označení slučitelné/alternativní vlastnosti.

5. *Implementační technologie meta-modelu* - Velká část nástrojů realizovaných jako zásuvný modul Eclipse (FMP, Feature Diagrams, S2T2 atd.) zvolila pro implementaci meta-modelu Eclipse Modeling Framework (EMF), jež je součástí zmíněné platformy. Některé nástroje (pure::variants, FeatureIDE), ač také postavené na platformě Eclipse, implementují meta-model vlastním způsobem. Speciálním případem je zde opět XFeature, jehož meta-model je postaven čistě na XML technologiích.

6. *Datový formát* - Prakticky všechny nástroje ukládaly model ve formátu XML či XMI. Objevily se ale i výjimky (FeatureIDE), kdy byla konfigurace uložena jen jako seznam vybraných vlastností po řádcích.

7. *Editor modelu vlastností* - Nástroje v zásadě používaly dva přístupy editace modelu, a to buď pomocí stromového editoru (FMP, SPLOT, Compositional Variability Manager) nebo grafického editoru diagramů (CaptainFeature, XFeature, FeatureIDE). Nástroje používající grafický editor pak umožňovaly buď pouze manuální pozicování prvků (XFeature) nebo i jejich automatické rozmístění dle jednoho či více algoritmů (FeatureIDE). Zvláštním případem je FeatureIDE, které podporovalo současně s grafickou editací i přímou editaci XML reprezentace modelu.

8. *Způsob editace* - Nástroje typicky používaly pro editaci hlavně kontextové menu, které však bylo v některých případech poněkud přehledné (XFeature). Využití nástrojové lišty s tlačítky a klávesových zkratek bylo mezi editory různé. Některé grafické editory (S2T2) navíc používaly i nástrojovou paletu. Velká část nástrojů také nějakým způsobem podporovala přetažení prvků (*drag and drop*). Detaily jednotlivých objektů se pak obvykle nastavovaly v příslušném panelu (typicky šlo o *properties view* u Eclipse).

9. *Editor konfigurace vlastností* - Zde většina nástrojů upřednostnila klasický stromový editor před nějakým grafickým způsobem reprezentace konfigurace. Ve stromovém editoru šlo obvykle vlastnost buď vybrat zatržením, nebo ji rovnou z konfigurace vyřadit. Zajímavý byl přístup FeatureIDE, které umožňovalo konfiguraci v základním a pokročilém módu (bylo povoleno více stavů vlastností). Alternativní grafickou reprezentaci konfigurace pak zvolilo jen několik nástrojů (XFeature, S2T2).

10. *Asistence při konfiguraci* - V zásadě většina nástrojů podporovala alespoň detekci a zobrazení konfiguračních chyb a skrývání neplatných konfiguračních voleb. Některé nástroje (SPLOT, pure::variants) dokázaly i identifikovat konflikty a nabídnout uživateli jejich automatické odstranění. Dalším zajímavým prvkem bylo automatické doplňování voleb (pure::variants, FeatureIDE) či zobrazení zbývajícího počtu validních konfigurací (FMP, pure::variants, FeatureIDE). Nástroje s podporou kardinality (FMP, XFeature) pak umožňovaly i duplikaci vlastností.

11. *Specializace modelu vlastností* - Specializaci modelu vlastností umožňovaly pouze dva nástroje (CaptainFeature a FMP)[36]. Specializace by také byl pravděpodobně schopen i XFeature, při správně zvoleném meta-modelu.

[36]Některé nástroje (pure::variants, FeatureIDE) sice umožňovaly při konfiguraci explicitně vyřadit některé volby, avšak zde o specializaci modelu vlastností nejedná.

12. *Definice omezení* - Většina nástrojů umožňovala definovat omezení v textové formě, buď pomocí nějakého vlastního jazyka vycházejícího z Booleovské algebry (FeatureIDE, pure::variants), nebo pomocí již nějakého existujícího jazyka (XPath u FMP a XFeature, pvProlog u pure::variants). Speciálním případem byl pure::variants, který u textových omezení rozlišoval 2 typy (omezení a restrikce). Jen malá část nástrojů umožňovala definovat mezi vlastnostmi i jiné (netextové) vazby (pure::variants, S2T2).

13. *Editace omezení* - Pro textová omezení nabízely nástroje většinou nějaké automatické doplňování vhodných klíčových slov či názvů vlastností a jejich barevné zvýraznění. Zajímavý byl přístup nástroje SPLOT umožňující uživateli vytvářet omezení pouze s pomocí myši. Nástroje podporující definici přímých vazeb mezi vlastnostmi je vytvářely buď pomocným dialogem (pure::variants) či graficky (S2T2).

14. *Vizualizace* - Některé nástroje poskytovaly uživateli různé pomocné pohledy na modelované objekty. Dobrým příkladem je zde pure::variants, poskytující například pohled zobrazující vztahy mezi jednotlivými vlastnostmi. Zmíněný nástroj také umožňoval vizualizaci modelu vlastností jako grafu s omezenými možnostmi editace.

15. *Integrace s ostatními nástroji* - Nástroje obvykle umožňovaly alespoň import či export datových formátů jiných známých nástrojů (obvykle šlo o pure::variants), případně i export do jiného datového formátu než XML (např. CSV u FeatureIDE). Pro některé (pure::variants, Gears) byla pak důležitá i přímá integrace s ostatními komerčními nástroji.

4 Požadavky na vytvářený nástroj

V předchozí kapitole jsme si představili řadu již existujících nástrojů a provedli jsme jejich srovnání z hlediska technické úrovně, funkcionality a přístupu k uživateli. Na základě zjištěných faktů se nyní pokusíme sestavit seznam požadavků na námi vytvářený nástroj.

4.1 Obecné požadavky

- *Zaměření* - Nástroj by měl být univerzální a měl by uživatele co nejméně omezovat v tom co chce modelovat.
- *Implementační technologie* - Zvolená implementační technologie by měla hlavě usnadnit vývoj samotného nástroje. Dalším důležitým požadavkem je pak multiplatformnost výsledného řešení.
- *Součásti* - Nástroj by měl umožňovat tvorbu modelů vlastností a z nich odvozených konfigurací. Nástroj by případně mohl i umožňovat specializaci modelů vlastností, avšak tato funkcionalita se nezdá být příliš zásadní, neboť ji implementovalo jen velmi málo nástrojů (CaptainFeature, FMP). Umožnění popisu architektury rodiny produktů a jejího propojení s modelem vlastností (tak jak to umí například pure::variants) je nad rámec této práce, a proto nebude v nástroji řešeno.

4.2 Požadavky na meta-model

- *Struktura* - Model vlastností by měl mít stromovou strukturu, a ne strukturu acyklického grafu. Takto uvažovala model vlastností většina nástrojů.
- *Vlastnosti* - Mělo by být možné vytvářet základní typy vlastností, tedy povinné a volitelné. Ačkoliv některé z nástrojů nepodporovaly duplikovatelné vlastnosti, autorovi se zdá být tento typ vlastností poměrně zásadní a proto by ho měl výsledný nástroj také podporovat. Kardinalita vlastnosti by měla být vyjádřena pouze jako interval, ne jako posloupnost intervalů u CaptainFeature.
- *Skupiny* - Většina nástrojů seskupování vlastností nějakým způsobem podporuje a měl by ho tedy podporovat i vytvářený nástroj, včetně možnosti specifikace kardinality skupin jako intervalu.
- *Atributy* - Jako většina ostatních nástrojů, i námi vytvářený nástroj by měl umožňovat přiřazovat vlastnostem atributy. Zde se jeví jako nejpraktičtější přístup nástrojů typu pure::variants, které umožňovaly neomezený počet atributů na jednu vlastnost. Pro atributy by mělo být možné zvolit jeden ze základních datových typů (alespoň číslo a řetězec) či případně i typ reference. Podpora speciálních datových typů (URL, datum a čas apod.) jako u pure::variants se nezdá být příliš zásadní, neboť se dají vždy zastoupit datovým typem řetězec.

- *Dodatečné informace* - Podobně jako u pure::variants, i výsledný editor by měl umožňovat přiřadit vlastnostem nejen jméno, ale i některé další vhodné informace (například popis, verzi apod.).
- *Dekompozice* - Vytvářený model by mělo být možné dekomponovat do několika menších celků, uložených například v samostatných souborech. Tím se zvýší celková přehlednost editace rozsáhlých modelů vlastností.
- *Datový formát* - Model by měl být uložen v datovém formátu, který je lidsky i strojově čitelný (např. XML) a je ho možné tedy zpracovat i externími nástroji. Případně by nástroj mohl podporovat i export do jiných datových formátů.
- *Rozšiřitelnost* - Tuto možnost podporovalo jen několik málo nástrojů (FMP), tedy nezdá se být jako příliš zásadní. Navíc by pravděpodobně zkomplikovala výslednou implementaci[1].

4.3 Požadavky na editaci modelu vlastností

- *Struktura* - Zásadním rozhodnutím zde je, zda pro editaci modelu požadovat stromový editor či grafický editor diagramů. Ačkoliv může být stromový editor při rozsáhlém modelu vlastností přehlednější, bylo by vhodné použít spíše editor grafický, a to ze dvou důvodů:
  1. Úprava modelu přes stromový editor probíhala ve většině nástrojů hlavně přes kontextové menu, které nebylo vždy zcela úplně pohodlné používat. Provádění některých operací (typicky těch co mění strukturu stromu) je v grafickém diagramu jednoduší a přehlednější.
  2. Uživatel bude pravděpodobně již seznámen s nějako grafickou notací diagramu vlastností a bude pro něj tedy srozumitelnější i jednodušší přímo s takovým diagram pracovat.
- *Grafická notace* - Editor diagramu by měl používat nějakou známou notaci, například tu kterou jsme si ukázali v tabulce 2.4.
- *Ovládání* - Editor by měl v rozumné míře nabízet všechny dostupné prvky pro editaci, jako je paleta nástrojů, nástrojová lišta, kontextové menu, klávesové zkratky, *drag and drop* apod. Cílem je zde hlavně usnadnit práci uživateli.
- *Funkcionalita* - Editor by měl podporovat kompletní editaci modelu, tedy tvorbu-/úpravu/mazání vlastností, skupin a atributů. Pro všechny operace by měla být k dispozici možnost je vrátit či znovu vyvolat (*undo*/*redo*).
- *Pozicování prvků* - Editor by měl uživateli umožnit jak manuální, tak automatické pozicování prvků dle nějakého rozvržení.
- *Validace* - Editor by měl validovat veškeré vstupy a upozorňovat uživatele na chyby. Případné chyby by měly být v editoru viditelně vyznačeny, aby si jich uživatel všiml.

[1] Meta-model by měl být navržen tak, aby jeho případné rozšiřování nebylo nezbytně nutné.

- *Export diagramu* - Diagram by mělo být možné exportovat jako bitmapový, případně i jako vektorový obrázek.

4.4 Požadavky na editaci omezení

- *Způsob definice omezení* - Omezení by měla být zadávána v textové podobě pomocí nějakého jazyka omezení, tak jako tomu je ve většině existujících nástrojů. Grafické vyjádření omezení v diagramu není nutné.
- *Jazyk omezení* - Nástroj by měl podporovat již nějaký existující jazyk omezení (např. XPath, OCL, či pvSCL), nebo by měl mít jazyk vlastní, ovšem dostatečně expresivní (například na úrovni zmíněného pvSCL).
- *Editor* - Editor omezení by měl být vhodně integrován do hlavního editoru diagramu vlastností. Dále by měl editor zvýrazňovat syntaxi zvoleného jazyka a případně i napovídat uživateli při editaci či automaticky doplňovat vhodné možnosti (*intelli-sense*).

4.5 Požadavky na editaci konfigurace vlastností

- *Struktura* - Ačkoliv většina nástrojů používala ke konfiguraci obvykle stromový editor se zatrháváním voleb, bylo by vhodné použít přístup nástroje XFeature. Uživatel při něm vytvářel podobný grafický diagram jako u modelu vlastností, avšak byl omezen pouze na některé validní možnosti (dle příslušného modelu vlastností). Tento přístup má několik výhod:
  1. Uživatel vidí obdobný digram pomocí kterého před tím popsal model vlastností.
  2. Uživatel na začátku nevidí všechny možnosti jako u stromového editoru, ale zobrazený diagram se postupně rozvíjí (roste), jak uživatel vybírá některé možnosti.
- *Funkcionalita* - Editor by měl umožnit přidávání a ubírání vlastností v konfiguraci a jejich případnou duplikaci (pokud to horní mez kardinality dovolí). Dále by mělo být možné nastavovat hodnotu atributů vybraných vlastností.
- *Ovládání* - viz stejný požadavek v části 4.3.
- *Validace* - Editor by měl automaticky validovat aktuální stav konfigurace a viditelně upozorňovat uživatele na chybná místa.
- *Asistence uživateli* - Editor by měl automaticky skrývat neplatné volby, a naopak aktivovat volby, které jsou povinné. Dále by editor mohl podporovat automatické řešení konfliktů konfigurace či zobrazení počtu zbývajících validních konfigurací.
- *Synchronizace s modelem vlastností* - Hlavní potíží při tvorbě konfigurace je její propojení s modelem vlastností od kterého byla odvozena. Protože předpokládáme, že model i konfigurace vlastností budou uloženy v samostatných souborech, klademe na nástroj následující požadavky:

1. Konfigurace vlastností musí být zobrazitelná/editovatelná i v případě nepřítomnosti původního modelu vlastností. Uživatel by měl být v takovém případě na jeho nepřítomnost upozorněn a měl by mít možnost ho znovu zvolit.
2. Editor by měl detekovat změnu původního modelu vlastností a měl by uživateli nabídnout možnost tyto změny do již vytvořené konfigurace zahrnout[2]. Případné změny v modelu vlastností by měly být uživateli nějak vizuálně prezentovány.

4.6 Ostatní požadavky

- *Integrace s ostatními nástroji* - Editor by měl umožňovat import či export datových formátů modelů vlastností jiných nástrojů, zejména pak těch nejznámějších (pure::variants). Zde se jako zásadní jeví hlavně možnost importu, umožňující přechod uživatelů ostatních nástrojů.
- *Vizuální nápověda* - Častým problémem pro uživatelů bývá špatná orientace v rozsáhlých diagramech vlastností. Nástroj by měl proto uživateli poskytnou alternativní pohledy na editovaný model, které by mu umožnily lépe se zorientovat:
  1. Uživatel by měl mít k dispozici zmenšený náhled na celý diagram pro rychlou orientaci.
  2. Uživatel by měl mít možnost si diagram zobrazit jako klasický strom s možností zabalování a rozbalování jeho podčástí.
  3. Uživatel by měl mít možnost nechat si přehledně zobrazit, které vlastnosti jsou ovlivněny kterými omezeními a naopak. Toto často bývá pro uživatele zásadní problém, neboť jedno omezení může ovlivňovat i několik vlastností, jež jsou umístěny na opačných koncích diagramu.

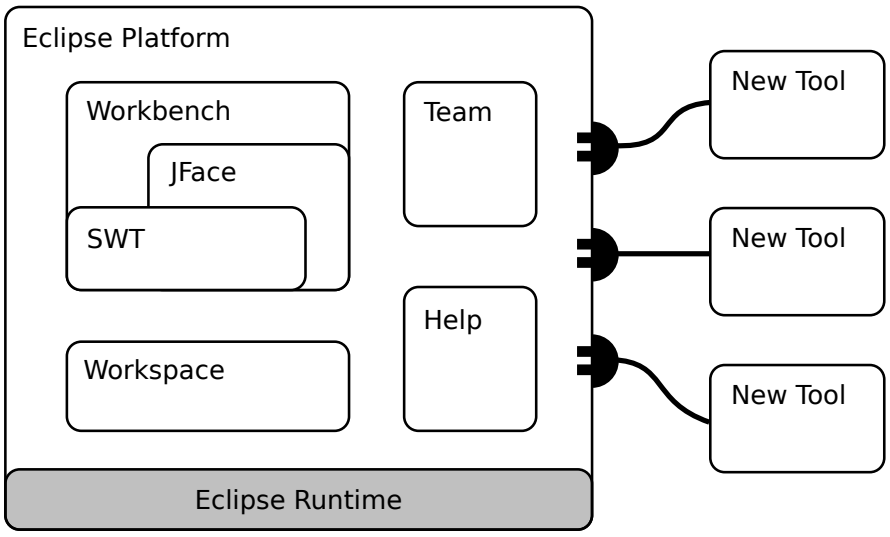
[2]Případně by mohl editor umožňovat i nějaký systém verzování modelů vlastností, kde by si uživatel mohl vybrat jakou verzi modelu vlastností použije. Toto je důležité zejména proto, že model vlastností se postupně vyvíjí v čase, tak jak se vyvíjí i daná rodina produktů.

5 Použité technologie

V této kapitole budou popsány základní technologie, jež byly použity pro implementaci výsledného nástroje. Konec této kapitoly pak obsahuje krátké zdůvodnění výběru těchto technologií.

5.1 Eclipse

Eclipse[1] bychom mohli jednoduše popsat jako multiplatformní integrované vývojové prostředí (IDE), napsané (z větší části) v programovacím jazyce Java. Pokud bychom chtěli být přesnější, popsali bychom Eclipse spíše jako otevřenou vývojovou platformu skládající se ze základního jádra a souboru zásuvných modulů, implementujících většinu její funkcionality. Zmíněné zásuvné moduly představují zřejmě největší výhodu celé platformy, neboť lze pomocí nich Eclipse libovolně rozšiřovat a stavět na něm zcela nové nástroje. Dostupnost mnoha existující modulů s potřebnou funkcionalitou pak navíc velmi urychluje vývoj. Poměrně trefně Eclipse popisují jeho samotní vývojáři jako *IDE for everything and nothing in particular*.



Obrázek 5.1: Architektura platformy Eclipse (verze 3).

Podívejme se nyní blíže na samotnou architekturu této platformy, jež je znázorněna na obrázku 5.1[2]. Základem celé platformy je běhové prostředí (*platform runtime*), které umožňuje načítat a vykonávat kód jednotlivých zásuvných modulů (*plug-ins*). Toto běhové prostředí je postavené na specifikaci frameworku OSGi[3] (verze 4), konkrétně její implementaci Equinox[4]. Samotné běhové prostředí Eclipse je napsané v jazyce Java a vyžaduje

[1] `http://www.eclipse.org/`
[2] Obrázek i části textu byly převzaty z [Bea06].
[3] Modulární systém pro jazyk Java, který implementuje dynamický komponentový model.
[4] `http://www.eclipse.org/equinox/`

tedy pro svojí funkčnost i běhové prostředí tohoto jazyka.

Každý zásuvný modul[5] je realizován jako JAR obsahující přeložené třídy, zdroje (např. ikony aplikace) a textový soubor *MANIFEST.MF* s metadaty zásuvného modulu. Jako typická metadata bychom mohli například uvést označení modulu, jeho jméno a verzi, jaké ostatní moduly jsou pro jeho běh potřeba, nebo jaké baličky daný modul importuje či exportuje. Platforma Eclipse oproti OSGi přidává navíc i soubor *plugin.xml*, definující tzv.:

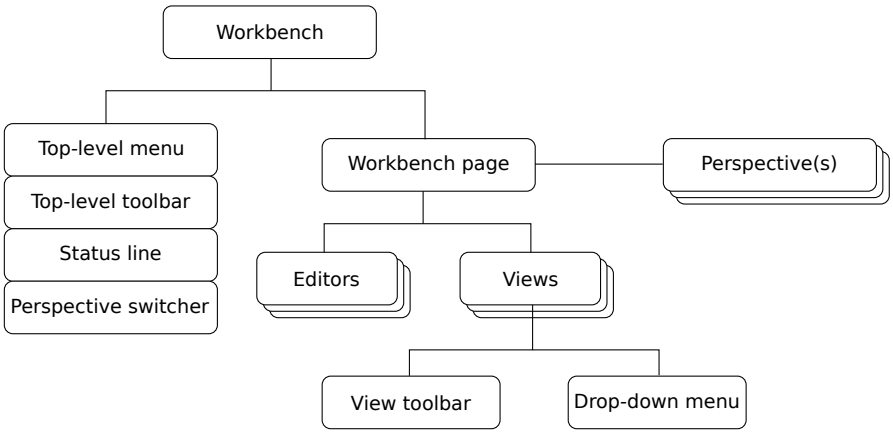
- *extension points* - Místa na kterých lze rozšířit funkcionalitu stávajícího zásuvného modulu.
- *extensions* - Implementace zmíněných rozšíření.

Typickým příkladem takového místa rozšíření je například hlavní menu aplikace, do kterého mohou ostatní moduly skrze své rozšíření přispívat (přidávat nové položky). Vrat'me se ale zpět k obrázku 5.1. Pro tvorbu uživatelského rozhraní jsou jednotlivým modulům k dispozici dvě knihovny:

- *SWT* - Nízkoúrovňová knihovna pro tvorbu grafického uživatelského rozhraní. Podobně, jako například SWING, nabízí i tato knihovna multiplatformní API pro tvorbu základních prvků uživatelského rozhraní (tlačítka, menu atd.), avšak na rozdíl od SWING využívá SWT pro tyto prvky nativní prostředky daného operačního systému[6].
- *JFace* - Pokročilejší nadstavba nad předchozí knihovnou, jejíž implementace již není (jako u SWT) systémově závislá. Na rozdíl od SWT umožňuje JFace práci s prvky uživatelského rozhraní s využitím návrhového vzoru MVC a přidává také například mechanismus pro tvorbu akcí nebo pokročilých dialogových oken (např. průvodců).

Na těchto dvou knihovnách je pak postaven tzv. *workbench*, představující výchozí bod pro tvorbu uživatelského rozhraní v prostředí Eclipse. Samotný uživatel vnímá *workbench* v podobě okna (obr. 5.3), skládajícího se z menu, nástrojové a stavové lišty a stránky (*workbench page*). Pro přehlednost si popišme jednotlivé části struktury *workbench*, znázorněné také na obr. 5.2 (převzatém z [Mca10]).

- *Workbench page* - Stránka sdružující editor a několik pohledů.
- *Editor* - Editor umožnující otevírání, modifikaci a ukládání objektů. Jde typicky například textový, formulářový či grafický editor.
- *View* - Pohled zobrazující informace o objektech dostupných v daném pracovním prostoru uživatele. Pohled může typicky sloužit například k zobrazení seznamu chyb, seznamu součástí projektu či k alternativnímu zobrazení editovaného objektu. Pohled může mít vlastní menu i nástrojovou lištu.
- *Perspective* - Rozvržení editoru a několika pohledů v rámci stránky. Jedna stránka může mít i několik perspektiv, mezi kterými si uživatel přepíná.



Obrázek 5.2: Struktura uživatelského rozhraní Eclipse workbench (logický pohled).

Pomocí rozšiřování *workbench* mohou ostatní zásuvné moduly tyto prvky do Eclipse přidávat. Tímto způsobem bylo například vytvořeno JDT[7], jež (velmi zjednodušeně řečeno) přidává do Eclipse vlastní editor pro jazyka Java a několik nových pohledů a perspektiv. Posledními komponentami celé architektury, které jsme ještě nepopsali, jsou pak:

- *Workspace* - Pracovní prostor uživatele, obsahující vytvořené projekty, jejich součásti a nastavení.
- *Team* - Rozhraní pro integraci platformy s různými verzovacími systémy.
- *Help* - Rozšiřitelný systém nápovědy celé platformy.

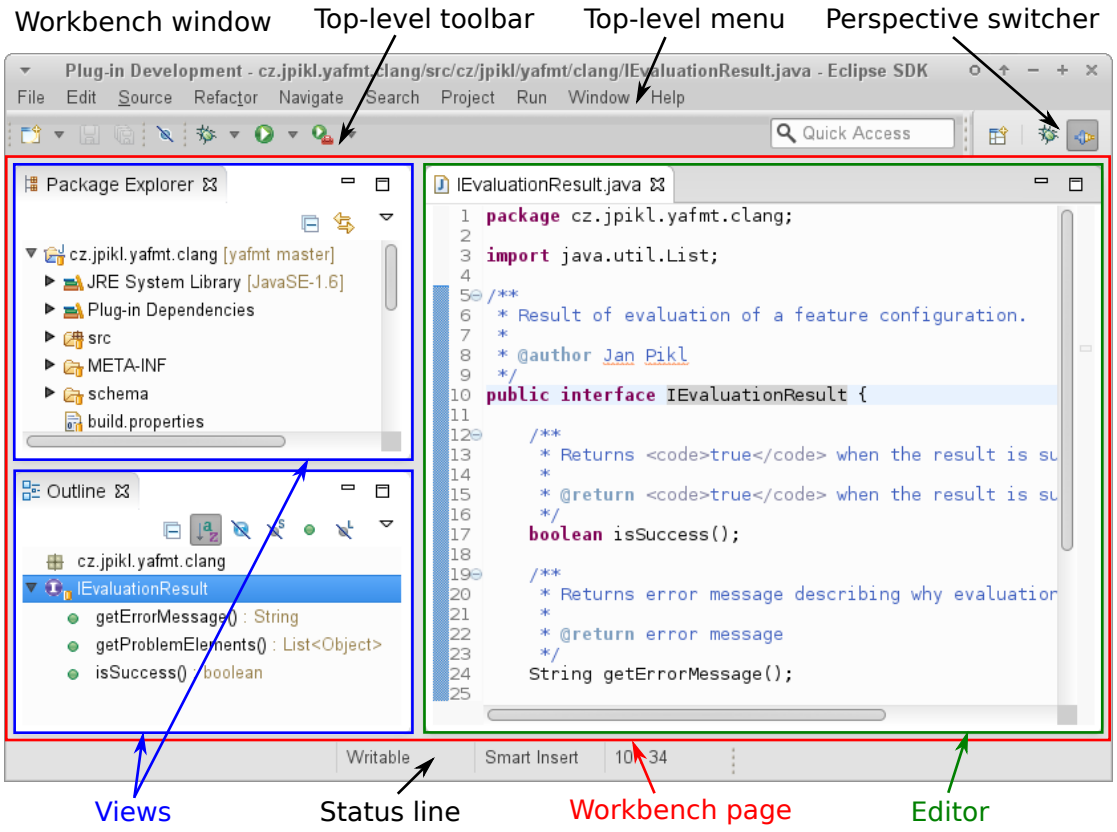
Samotná platforma Eclipse může být použita dvěma způsoby, a to buď jako integrované vývojové prostředí (IDE), rozšiřitelné pomocí zásuvných modulů, nebo jako tzv. *Rich Client Platform* (RPC), představující běhové prostředí spolu s minimální množinu modulů, jež lze použít k tvorbě samostatných aplikací.

Na závěr ještě dodejme, že výše popsaná architektura odpovídala platformě Eclipse verze 3. V současné době probíhá nástup její 4. verze (označované jako e4), která přináší několik zásadních změn, jako například deklarativní popis GUI pomocí modelu, servisně orientovaný programový model postavený na OSGi či integraci webových technologií (CSS, JavaScript) jako vývojových prostředků pro celou platformu. e4 navíc obsahuje speciální vrstvu pro zajištění zpětné kompatibility s API verze 3.

5.2 Eclipse Modeling Framework

Eclipse Modeling Framework[8] (zkráceně EMF) bychom mohli stručně popsat jako framework pro strukturované modelování se schopností generování zdrojového kódů. Pro

[5]V terminologii OSGi označovaný spíše jako *bundle*.
[6]Například WinAPI na systému Windows či knihovnu GTK+ na systému Linux.
[7]Balík nástrojů pro pro platformu Eclipse pro vývoj aplikací v jazyce Java.
[8]`http://www.eclipse.org/modeling/emf/`



Obrázek 5.3: Struktura uživatelského rozhraní Eclipse workbench (okno aplikace).

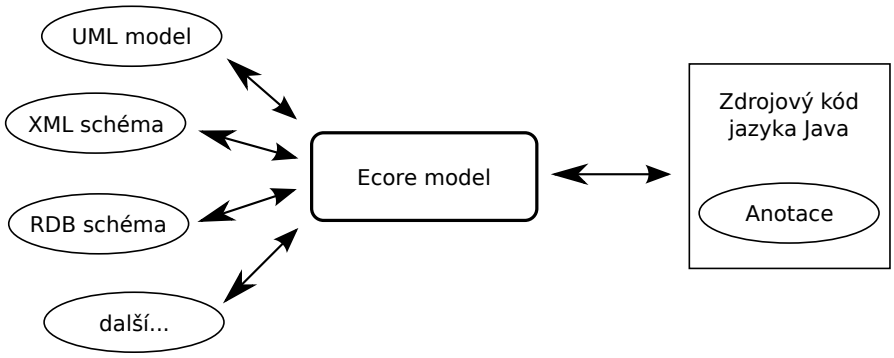
neznalého čtenáře bude pravděpodobně obtížné si pod touto definicí představit něco konkrétního, a proto vysvětlíme principy EMF na jednoduchém příkladu, jež je převzatý z knihy [Stein08].

Představme si, že jako vývojář aplikace potřebujeme vytvořit jednoduchý doménový model (kupříkladu pro systém objednávek zákazníků). Pravděpodobně bychom mohli začít tím, že si model nejprve navrhneme v podobě UML diagramu a na základě toho pak vytvoříme i zdrojový kód jednotlivých tříd a rozhraní. Vzhledem k tomu, že je objednávky nutné uchovávat po delší dobu, bude nutné napsat i kód, který daný model uloží a načte ze souboru, typicky ve formátu postaveném na XML. Aby bylo možné takto serializovaný model validovat, bude navíc potřeba vytvořit i příslušné XML schéma (XSD).

Jak můžeme vidět, byli jsme nuceni vytvořit hned několik artefaktů (UML diagram tříd, zdrojový kód tříd, XML schéma), ačkoliv každý z nich vyjadřoval tu samou informaci (popis struktury našeho modelu). Tuto informaci obecně označujeme jako meta-model.

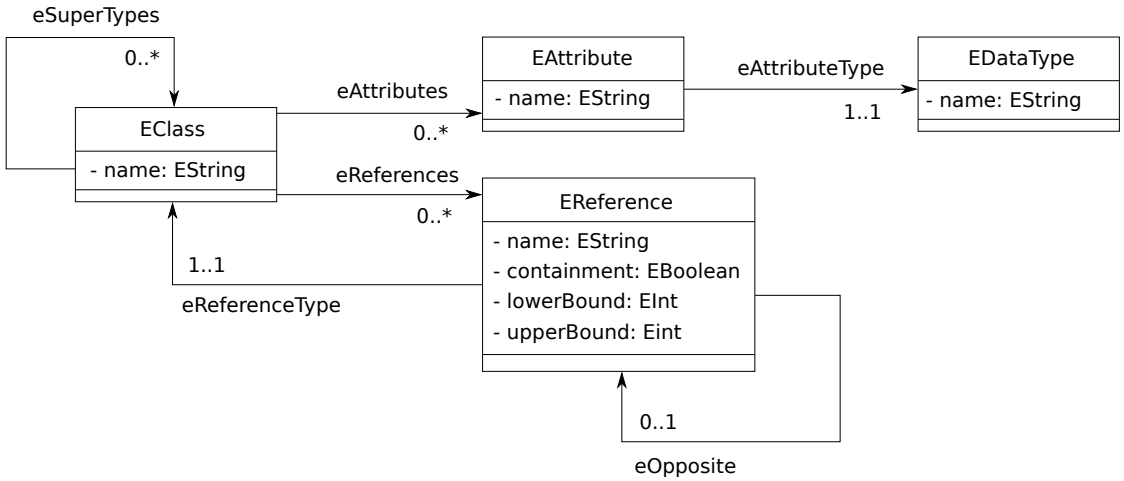
Pro ujasnění terminologie zde definujme pojem meta-model jako strukturální popis modelu. Model pak můžeme chápat jako instanci meta-modelu. Konkrétním příkladem meta-modelu a jeho instance je například dvojice třída, objekt nebo dvojice XML schéma, XML dokument.

Vrat'me se ale zpět k našemu příkladu, kde jsme ten samý meta-model (poněkud zby-



Obrázek 5.4: Možnosti konverze Ecore modelu v rámci EMF.

tečně) vyjádřili několikrát, pokaždé v jiné formě (UML diagram tříd, zdrojový kód tříd, XML schéma). Pokud bychom místo toho na začátku použili EMF, mohli jsme si většinu této práce ušetřit, neboť právě jedna z hlavních výhod EMF je, že nám umožňuje meta-model definovat v univerzální formě a zní pak tyto ostatní formy meta-modelu jednoduše vygenerovat (viz obr. 5.4). Tato univerzální forma je v rámci EMF nazývána jako Ecore model[9]. Ilustraci toho, jak vypadá struktura Ecore modelu, můžeme vidět na obrázku 5.5 (upozorňujeme čtenáře, že jde pouze o velmi zjednodušený pohled)[10]. Oba uvedené obrázky byly převzaty z [Stein08].



Obrázek 5.5: Zjednodušené znázornění struktury Ecore modelu.

Výhodné je na celém tomto řešení zejména to, že vývojář nemusí Ecore model vytvářet přímo, ale může ho vygenerovat z libovolného uvedeného artefaktu (zdrojový kód, UML model, XML schéma atd.), tedy mezi Ecore a ostatními vyjádřeními meta-modelu existuje vzájemně jednoznačná konverze[11]. Pokud chtěl vývojář vytvořit Ecore model přímo, může

[9]Většinou se spíše používá obecnější označení EMF model.

[10]Pro zajímavost uveďme, že struktura Ecore modelu je v EMF vyjádřena jeho meta-modelem, kterým není nic jiného než opět Ecore model. Tedy, Ecore je svým vlastním meta-modelem.

[11]Aby mohl EMF ze zdrojového kódu získat dostatečné informace o meta-modelu, musí být tato infor-

to udělat buď programově (pomocí API frameworku EMF) nebo v grafickém editoru, jež je součástí EMF.

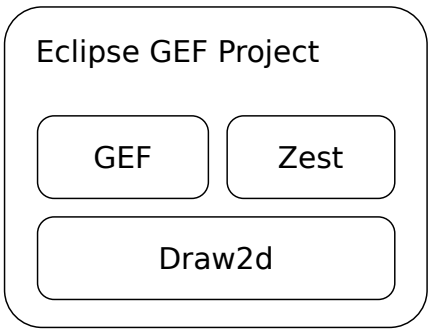
Samotný Ecore model je ve vnější paměti uchováván jako XMI dokument, který je při načtení do vnitřní paměti převeden do objektové podoby. K této podobě je pak následně přistupováno buď prostřednictvím vygenerovaných rozhraní nebo skrze reflexivní API. Velkou výhodou zmíněného API je, že dovoluje přistupovat identicky k libovolné instanci Ecore modelu a umožňuje tak tvorbu generických, znovupoužitelných komponent. Příkladem takové komponenty je například dostupná generická implementace načítání a ukládání libovolné instance Ecore modelu v podobě XMI dokumentu či generická implementace návrhového vzoru *observer*, využívaná v rámci celého EMF.

EMF díky výše popsaným vlastnostem představuje velmi mocný nástroj pro tzv. *model-driven architecture*. Tento pojem označuje způsob vývoje software, kdy jsou jednotlivé části architektury aplikace nejprve popsány pomocí platformě nezávislého modelu a z něj je pak (automaticky) vygenerován zdrojový kód aplikace. Tímto způsobem je například možné vygenerovat z Ecore kompletní zdrojový kód (stromového) editoru pro jím popsaný model.

Na závěr ještě dodejme několik faktů, jako například, že framework EMF se stal od verze e4 nedílnou součástí jádra vývojového prostředí Eclipse. Mimo samotné EMF ještě existuje jeho několik rozšiřujících projektů, umožňujících například porovnání, sledování změn, transakční zpracování, dotazování či validaci EMF modelů. Všechny tyto projekty jsou pak včetně EMF (označovaným jako EMF core) součástí tzv. Eclipse Modeling Project[12].

5.3 Graphical Editing Framework

Graphical Editing Framework (GEF)[13] je projekt, jehož cílem je zprostředkovat technologie pro tvorbu interaktivních grafických editorů a pohledů pro platformu Eclipse. Samotný projekt GEF se skládá ze tří částí (viz obr. 5.6), jež nyní samostatně popíšeme. Uvedený obrázek byl převzat z [Rub11].



Obrázek 5.6: Struktura projektu GEF.

mace v kódu vyjádřena pomocí anotací. Obdobně jsou některé tyto informace vyjádřeny v definici XML schéma pomocí speciálních atributů.

[12] `http://www.eclipse.org/modeling/`

[13] `http://www.eclipse.org/gef/`

5.3.1 Draw2d

Draw2d představuje elementární část projektu GEF. Jeho primárním úkolem je vykreslování a umísťování veškerých (dvourozměrných) grafických prvků v rámci kreslicí plochy a zajištění elementární interakce s uživatelem.

Základním principem Draw2d je, že místo toho aby programátor psal nízkoúrovňový kód pro vykreslení celého diagramu, používá již hotové grafické obrazce (tzv. *figures*), ze kterých postupně celý diagram skládá. K dispozici jsou základní obrazce (např. čtverec, elipsa, textový popisek), jejichž parametry (barva, okraj, rozměry atd.) lze konfigurovat. Programátor může nicméně vytvářet i obrazce vlastní, například kompozicí z těch již existujících. Veškeré obrazce jsou v rámci kreslící plochy organizovány do hierarchické struktury, což umožňuje optimalizovat většinu prováděných operací (např. při změně stavu obrazce je překreslen pouze on a jeho potomci). Jednotlivé obrazce lze navíc propojovat pomocí viditelných spojení (*connections*).

Pokročilejší schopností Draw2d je pak rozmísťování obrazců po kreslící ploše dle zvolených algoritmů rozvržení (*layouts*) či určení, jakým způsobem povede spojení mezi jednotlivými obrazci (*connection routers*). Draw2d navíc umožňuje zpracovávat i uživatelské vstupy (myš a klávesnice) provedené nad jednotlivými obrazci a spojeními.

5.3.2 GEF

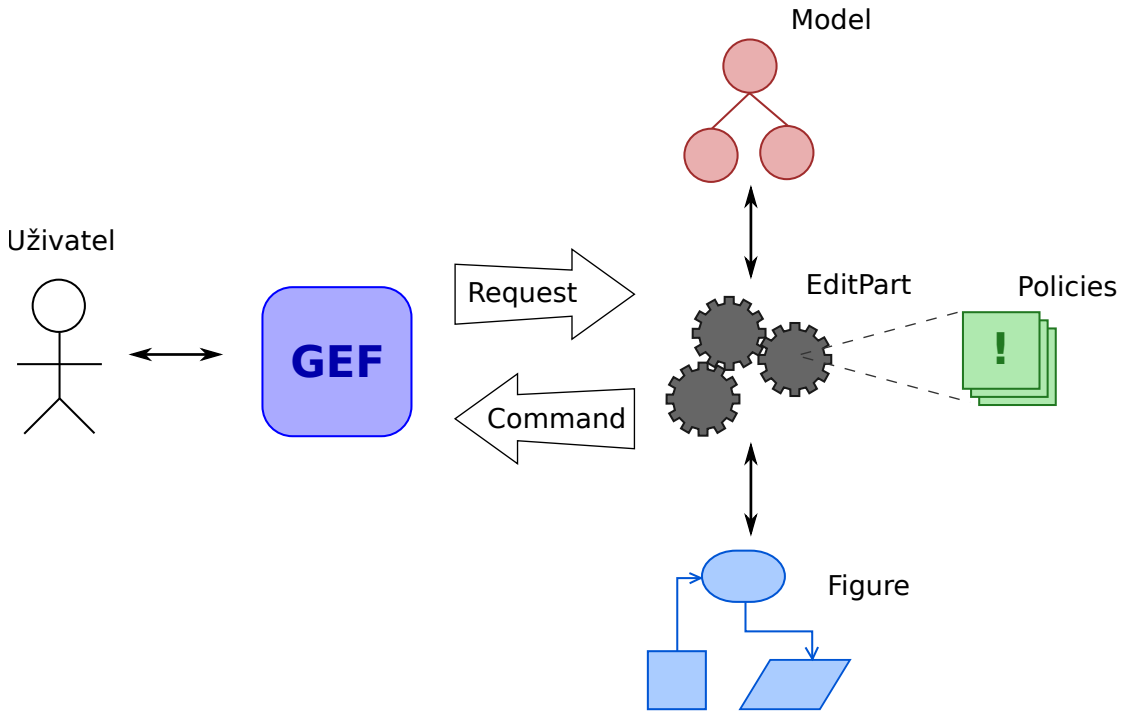
GEF je MVC framework[14] určený pro tvorbu interaktivních grafických editorů v prostředí Eclipse. Hlavní výhodou celého framworku je jeho velmi dobře navržená architektura a zejména to, že neklade žádné požadavky na editovaný model, či jak má být tento model prezentován uživateli.

Zmíněná architektura je z větší části založena na používání nejrůznějších návrhových vzorů, zejména pak vzoru *model-view-controller*, jež je použit pro interakci mezi uživatelem, editovaným modelem a zobrazeným diagramem. To, jak tato interakce probíhá, můžeme vidět obrázku 5.7. Základem fungování každého editoru postaveném na GEF jsou tři komponenty, odpovídající daným částem vzoru MVC:

- *Model* - Jeden či více objektů editovaných uživatelem. GEF neklade na strukturu modelu žádné požadavky, může jít klidně o prosté objekty jazyka Java či EMF model.
- *View* - Je realizován pomocí již zmíněného Draw2d v podobě jeho obrazců (*figures*). Každý obrazec zobrazuje odpovídající části modelu uživateli.
- *Controller* - Je realizován pomocí tzv. *editparts* z frameworku GEF. Úkolem těchto objektů je synchronizace mezi částmi modelu a jejich protějšky v podobě obrazců a úprava modelu dle vstupů uživatele. Pro každý prvek modelu existuje odpovídající *editpart*.

Popišme si nyní, jak vypadá takový obvyklý životní cyklus editoru postaveném na GEF. Po otevření editoru prozkoumá GEF prvky modelu a na základě specifikované tovární třídy

[14] Ačkoliv je jako GEF nazýván celý tento projekt, většinou se pod zkratkou GEF myslí právě tento framework.



Obrázek 5.7: Fungování editoru postaveném na GEF.

vytvoří instance odpovídajících *editparts*. Každý *editpart* poté sestrojí pro daný prvek modelu jeho vizuální reprezentaci v podobě obrazce (*figure*). Aby mohl *editpart* detekovat změny modelu a aktualizovat pak i příslušný obrazec, používá se obvykle návrhový vzor *observer*, kdy modifikovaný objekt svého pozorovatele (*editpart*) na tyto změny sám upozorňuje.

Žádný *editpart* nikdy nepracuje s uživatelským vstupem přímo, ale je od něj odstíněn pomocí framworku GEF. Ten zpracovává elementární uživatelské vstupy (myš, klávesnice) a konstruuje z nich vysokoúrovňové požadavky (*requests*), které jsou pak teprve přes *editparts* zpracovány. Příkladem jednoho takového požadavku může být například vytvoření objektu, který uživatel vybral v paletě nástrojů, nebo přidání spojení mezi dvěma objekty.

Zpracování požadavků probíhá na straně *editparts* pomocí tzv. politik (*policies*), jichž může mít mít *editpart* hned několik, pro každý typ požadavku jinou. Jednotlivé politiky mohou být navíc za běhu editoru dynamicky aktivovány či deaktivovány a jde tedy vlastně o aplikaci návrhového vzoru *strategy*. V případě, že je nalezena politika, jež dokáže daný požadavek zpracovat, zkonstruuje tato politika příkaz (*command*) zapouzdřující operaci provedenou na modelem. GEF následně tento příkaz přijme a vykoná. Každý takový příkaz navíc uchovává i původní stav měněného objektu, takže je možné jednotlivé změny vracet či znovu aplikovat (*undo*/*redo* operace). Opět zde tedy můžeme vidět použití návrhového vzoru, tentokráte (stejnojmenného) *command*.

5.3.3 Zest

Zest je knihovna umožňující vizualizaci stromových a grafových struktur. Podobně jako GEF, i Zest využívá pro vykreslování Draw2d, avšak na rozdíl od GEF neumožňuje zobrazovaný objekt přímo editovat.

Tato knihovna se dá použít dvěma způsoby. První z nich spočívá v ručním vytvoření všech objektů představujících daný graf (uzly a hrany) a jejich zobrazení pomocí dané komponenty. Tento přístup má nevýhodu v tom, že pokud máme nějaký existující model, který chceme zobrazit, musíme informaci mezi ním a grafem ručně synchronizovat, tedy není zde žádné oddělení mezi modelem a jeho prezentací.

Druhý způsob je poměrně flexibilnější a obaluje předchozí přístup použitím návrhového vzoru MVC, podobně jako tomu bylo mezi knihovnami JFace a SWT. Programátorovi zde v podstatě stačí implementovat rozhraní, jež zprostředkovává přístup k datům modelu a mapuje ho jednotlivé uzly a hrany. Zbylé komponenty knihovny už pak jen získané informace zobrazí v podobě grafu. Obdobným způsobem (implementací zprostředkujícího rozhraním) lze například ovlivňovat jakým způsobem budou jednotlivé uzly a hrany vykresleny. Při použití tohoto přístupu lze také navíc definovat filtry omezující viditelnou část grafu, což může přinést značné urychlení při vykreslování rozsáhlých struktur.

Důležitou částí této knihovny je pak sada různých algoritmů, které můžou být použity k rozmístění jednotlivých uzlů (například do stromu, radiálního stromu, mřížky atd.). Tyto algoritmy mohou být navíc mezi sebou kombinovány k dosažení odlišných výsledků. Poměrně zajímavá je také schopnost této knihovny animovat veškeré strukturální změny grafu, včetně změn při nichž dochází k přeskupení vrcholů.

5.4 Xtext

Xtext[15] je framework pro vývoj programovacích a doménově specifických jazyků. Hlavní výhodou tohoto frameworku je, že pokrývá všechny aspekty tvorby takového jazyka od jeho syntaktického analyzátoru až po jeho editor, včetně integrace s platformou Eclipse.

Xtext, podobně jako EMF, usnadňuje programátorovi práci tím, že dokáže většinu zdrojového kódu implementace automaticky vygenerovat. Programátorovi stačí pouze formálně popsat syntaxi daného jazyka pomocí gramatiky a Xtext z ní pak vytvoří následující komponenty:

- *Lexikální a syntaktický analyzátor jazyka.* Pro vytvoření těchto součástí využívá Xtext interně, již existující, generátor ANTLR[16].
- *EMF model* představující výstup syntaktického analyzátoru. Díky použití EMF může programátor těžit ze všech výhod tohoto frameworku popsaných v podkapitole 5.2. Programátor může navíc vynutit i použití zcela vlastního EMF modelu, jeho namapováním prvky gramatiky jazyka.

[15] `http://www.eclipse.org/Xtext/`
[16] `http://www.antlr.org/`

- *Textový editor jazyka* s podporou zvýrazňování syntaxe, nápovědou, automatickým doplňováním (*intelli-sense*), skládáním bloků kódu, možností refaktorizace apod.

Veškeré uvedené komponenty včetně editoru jsou vytvořeny v podobě zásuvných modulů, což umožňuje jejich bezproblémovou integraci do Eclipse. Celý proces generování je navíc zcela konfigurovatelný a výsledné komponenty lze přizpůsobovat pomocí vkládání závislostí (*dependency injection*). Programátor díky tomu může například nadefinovat vlastní barevné schéma pro zvýrazňování syntaxe v editoru, či změnit chybové zprávy syntaktického analyzátoru. Pro zmíněné vkládání závislostí je v rámci Xtext používán framework Google Guice[17].

5.5 Zdůvodnění výběru technologií a alternativy

Použití platformy Eclipse a frameworku EMF bylo specifikováno jako základní požadavek již v zadání této práce. Nicméně, i pokud by tento požadavek nebyl vznesen, byly by obě dvě technologie pravděpodobně stejně vybrány, vzhledem k jejich výhodám popsaným v podkapitolách 5.1 a 5.2. Jako jedinou nevýhodu vidí autor pouze poměrně dlouhou dobu nutnou pro porozumění a zvládnutí těchto technologií.

Pro tvorbu grafický editorů v prostředí Eclipse jsou mimo GEF dostupné ještě dva frameworky, a to GMF a Graphiti. GMF (Graphical Modeling Framework)[18] můžeme chápat jako framework zprostředkovávající integraci mezi EMF a GEF. Základní vlastností GMF je, že umožňuje na základě předloženého EMF modelu (a několika dalších) vygenerovat (téměř) kompletní zdrojový kód GEF editoru, jež si pak může programátor přizpůsobovat dle svých potřeb.

Stejně jako GMF, i Graphiti[19] je framework postavený na technologiích EMF a GEF, avšak na rozdíl od GMF neposkytuje Graphiti žádné možnosti generování kódu. Místo toho se tento framework zaměřuje spíše na poskytnutí standardní implementace většiny funkčnosti editoru a nabízí API, jež odstiňuje programátora od komplexnosti GEF a Draw2d. Editory postavené na tomto frameworku používají navíc standardizovaný vzhled digramů[20].

I přes zmíněné klady obou frameworků byl pro implementaci výsledného nástroje vybrán GEF, a to z několika důvodů:

- Vzhledem k delší době existence GEF oproti GMF a Graphiti je pro něj dostupné větší množství návodů a je také lépe dokumentován[21].
- GMF je poměrně komplexnější framework než GEF a jeho zvládnutí je tedy složitější a časově náročnější.
- Ačkoliv jsou GMF a Graphiti v jisté míře přizpůsobitelné, realizace některých specifických požadavků, se kterými není v těchto frameworcích počítáno, je poměrně

[17] `https://code.google.com/p/google-guice/`
[18] `http://www.eclipse.org/modeling/gmp/`
[19] `http://www.eclipse.org/graphiti/`
[20] Tento vzhled byl navržen specialisty ze společnosti SAP AG, odkud tento framework původně pochází.
[21] Vzhledem k autorově původní neznalosti platformy Eclipse byl toto zásadní důvod pro použití GEF.

problematická, na rozdíl od nízkoúrovňového GEF. Jedním takovým příkladem je vytvářený editor konfigurace, popsaný v kapitole 6.5, pracující rovnou nad dvěma souvisejícími modely (model vlastností a konfigurace vlastností), což je problém při použití GMF, neboť ten očekává pouze jediný model. Zmíněný editor konfigurace navíc používá i poměrně specifický způsob editace bez použití palety nástrojů.

Důvodem výběru posledního zmíněného frameworku Xtext jsou jeho popsané výhody a nedostupnost obdobné alternativy pro platformu Eclipse.

6 Implementace nástroje

V této kapitole popíšeme způsob implementace výsledného nástroje a jeho jednotlivých součástí. Samotný nástroj byl průběhu vývoje pracovně označován jako YAFMT[1] (Yet Another Feature Modeling Tool), což bylo nakonec zvoleno i jako jeho oficiální název.

6.1 Architektura nástroje

Nástroj YAFMT je implementován jako sada zásuvných modulů pro platformu Eclipse. Tyto moduly jsou navíc organizovány do čtyř skupin, tvořících samostatné komponenty[2]:

Feature Model Editor	- editor modelu vlastností
Feature Configuraction Editor	- editor konfigurace vlastností
Feature Model Visualizer	- vizualizace modelu vlastností
Boolean Constraint Language	- jazyk omezení BoolCL

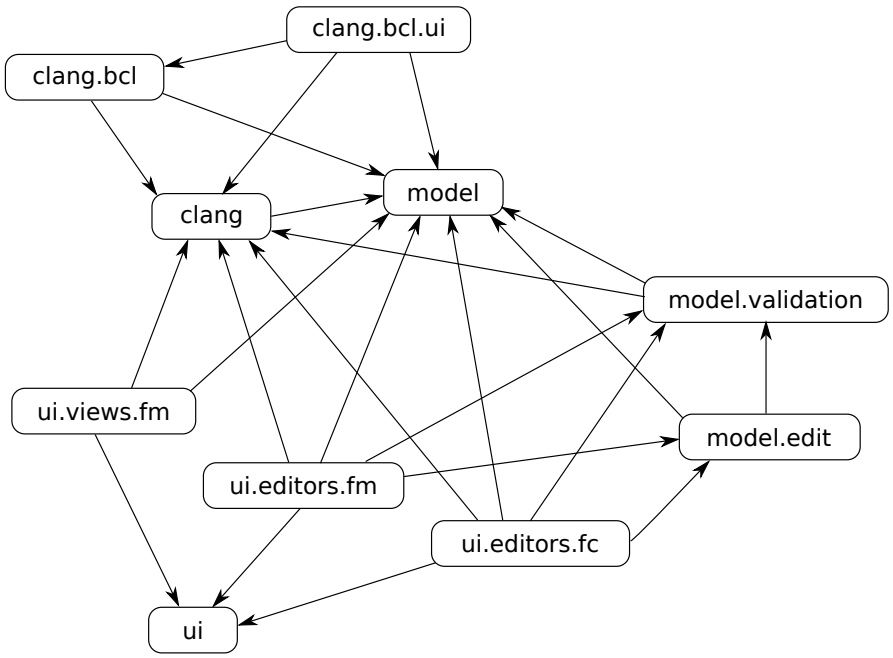
První tři uvedené komponenty lze nainstalovat a používat naprosto samostatně. Poslední komponenta je pak rozšiřující a umožňuje v rámci ostatních komponent používat jazyk omezení BoolCL. Uveďme nyní stručný popis jednotlivých zásuvných modulů, ze kterých jsou dané komponenty složeny:

cz.zcu.yafmt.model	- definice EMF modelu a jeho implementace
cz.zcu.yafmt.model.validation	- validátory pro jednotlivé části modelu
cz.zcu.yafmt.model.edit	- integrace modelu do uživatelského rozhraní
cz.zcu.yafmt.clang	- definice *extension point* pro přidávání jazyků omezení
cz.zcu.yafmt.clang.bcl	- implementace jazyka omezení BoolCL
cz.zcu.yafmt.clang.bcl.ui	- integrace jazyka BoolCL do uživatelského rozhraní
cz.zcu.yafmt.ui	- společný kód modulů tvořících uživatelského rozhraní
cz.zcu.yafmt.ui.editors.fm	- implementace editoru modelu vlastností
cz.zcu.yafmt.ui.editors.fc	- implementace editoru konfigurace vlastností
cz.zcu.yafmt.ui.views.fm	- implementace vizualizace modelu vlastností

Znázornění jednotlivých zásuvných modulů a jejich vzájemných závislostí je znázorněno na obrázku 6.1 (pro přehlednost byl odstraněn prefix *cz.zcu.yafmt*). Rozdělení modulů do jednotlivých komponent je pak znázorněno na obrázku 6.2. Moduly, jež na obrázku horizontálně protínají několik komponent, jsou mezi nimi sdíleny.

[1] Vyslovováno jako [waɪ-ɛf-ɛm-tiː]

[2] Abychom byli přesní, jde spíše o seskupení zásuvných modulů tvořící daný celek funkcionality. V terminologii Eclipse je takové seskupení označováno jako *feature* (toto označení nemá nic společného s modely



Obrázek 6.1: Zásuvné moduly nástroje YAFMT a jejich vzájemné závislosti.

6.2 Meta-model

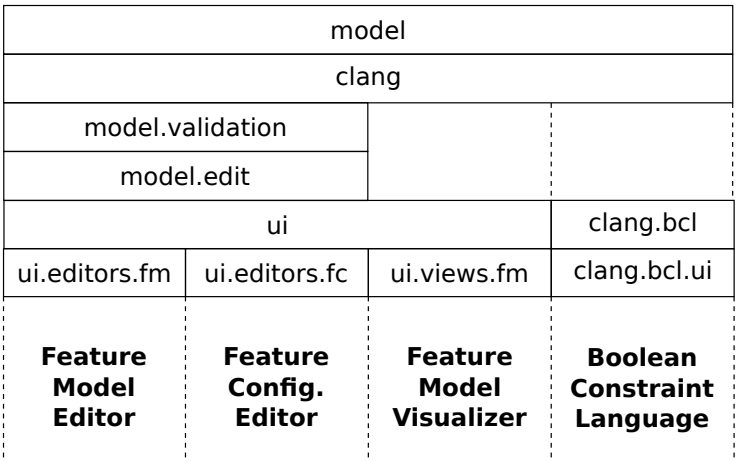
Model i konfigurace vlastností mají svůj meta-model definovaný ve formátu Ecore frameworku EMF. Z tohoto popisu meta-modelu byl pak automatickým procesem vygenerován odpovídající zdrojový kód a případně byly některé jeho části upraveny či doplněny. EMF standardně generuje pro každý prvek meta-modelu zvlášť jeho rozhraní i jeho implementaci, což bylo použito i v našem případě.

Zmíněné definice meta-modelu i vygenerovaný kód jsou umístěny v samostatném modulu *cz.zcu.yafmt.model*. Soubory popisu meta-modelů *FeatureModel.ecore* a *FeatureConfiguration.ecore* lze najít v adresáři *models*. Generování probíhalo zpracování souboru *FeatureModel.genmodel*, obsahujícím kromě odkazu na oba zmíněné soubory i parametry celého procesu generování. Složka *src* pak obsahuje vygenerovaný zdrojový kód rozdělený do dvou balíků *cz.zcu.yafmt.model.fm* a *cz.zcu.yafmt.model.fc*, dle příslušného meta-modelu.

6.2.1 Model vlastností

Meta-model modelu vlastností vychází z popisu modelů vlastností s kardinalitou a atributy uvedeném v [Cza05b], jež jsme popsali v kapitole 2.5. Tento meta-model byl z praktických důvodů mírně modifikován a rozšířen o některé prvky meta-modelu nástroje XFeature, popsaného v [Pas05]:

vlastností).



Obrázek 6.2: Použití zásuvných modulů v rámci jednotlivých komponent YAFMT.

1. Kardinalita seskupených vlastností není omezena pouze na hodnotu [0..1], ale může být libovolná, jako u samostatných vlastností. Tedy, seskupené vlastnosti mohou být duplikovatelné.

2. Vlastnost může mít libovolný počet pojmenovaných atributů.

Díky první modifikaci jsme získali o trochu obecnější meta-model, než je uveden v [Cza05b]. Pokud bylo v rámci původního meta-modelu potřeba vytvořit seskupenou duplikovatelnou vlastnost, musela být vytvořena jako pod-vlastnost jiné seskupené vlastnosti, jak ukazuje obrázek 6.3a. Ekvivalentní vyjádření odpovídající našemu meta-modelu je na obr. 6.3b. Obrázek 6.3c pak ukazuje případ, který se v rámci původního meta-modelu nedal zachytit[3].

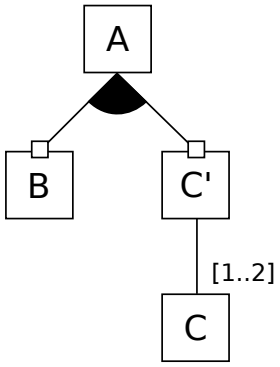
Druhá modifikace byla přidána z čistě praktického hlediska. V původním meta-modelu, popsaném v [Cza05b], muselo být více atributů na jednu vlastnost realizováno v podobě několika pod-vlastností, což je při jejich velkém počtu nepřehledné.

Struktura meta-modelu, vyjádřená v podobě diagramu tříd, je znázorněna na obr. 6.4. Model vlastností (*FeatureModel*) má právě jednu vlastnost představující kořen celého stromu. Případně může mít také odkazy na jiné vlastnosti umístěné mimo hlavní strom[4]. Vlastnost (*Feature*) může mít libovolný počet pod-vlastností a skupin (*Group*), jež pod sebou sdružují další vlastnosti. U vlastnosti i skupiny lze specifikovat dolní a horní mez kardinality (*lower*, *upper*). Každá vlastnost může mít dále neomezený počet atributů (*Attribute*) typu pravdivostní hodnota, celé číslo, desetinné číslo nebo řetězec (*AttributeType*). Jednotlivé vlastnosti jsou mezi sebou rozlišeny pomocí identifikátoru ID, který je pro ně v rámci celého modelu vlastností unikátní. Obdobně jsou identifikovány atributy, avšak jejich ID musí být unikátní pouze v rámci příslušné vlastnosti.

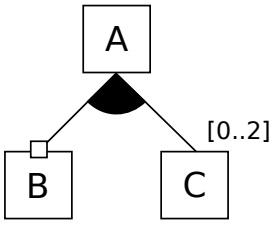
Důležité je zmínit způsob zachycení dodatečných informací v modelu vlastností. Na rozdíl od přístupu nástroje pure::variants, kdy bylo pro vlastnosti možné definovat velké

[3]V tomto případě je validní konfigurací jakákoliv dvou až pěti prvková kombinace vlastností X a Y.

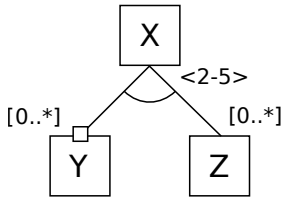
[4]Jde typicky o případ, kdy uživatel při editaci diagramu přidal vlastnost, ale ještě jí nepřiřadil žádného rodiče.



(a) Vyjádření s původním meta-modelem.



(b) Vyjádření s upraveným meta-modelem.



(c) Model vlastností nevyjádřitelný v původním meta-modelu.

Obrázek 6.3: Vyjádření některých modelů vlastností při použití meta-modelu nástroje YAFMT.

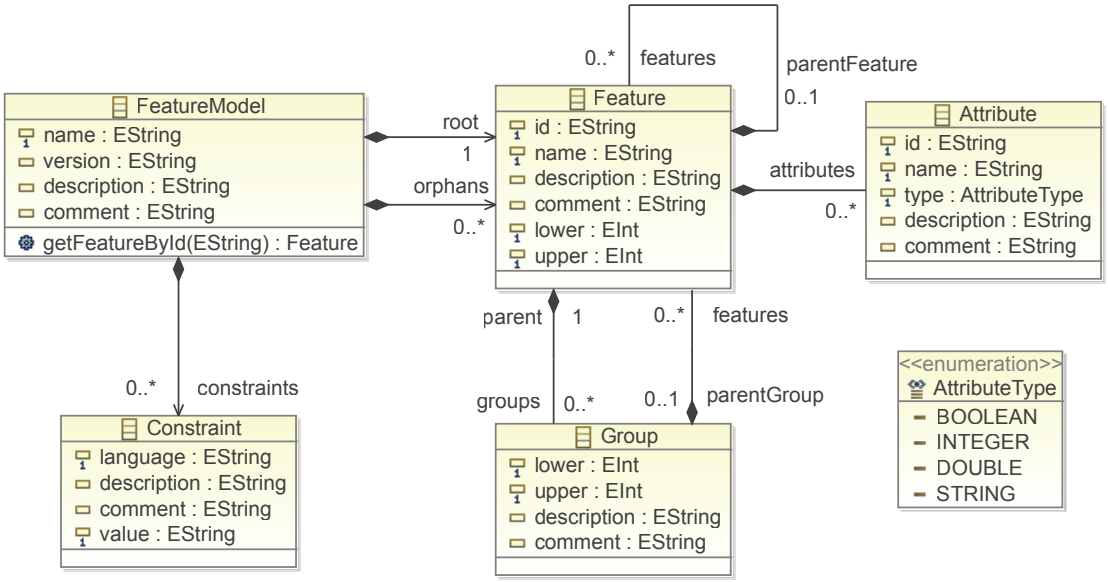
množství parametrů (verze, priorita, čas vytvoření, autor atd.), umožňuje náš meta-model pro každý jeho prvek zachytit pouze parametry dva. První z nich (*description*) slouží jako krátký (jednořádkový) popis daného elementu. Druhým z nich je komentář (*comment*), jež může obsahovat libovolně dlouhý textový obsah. Uživatel do něj tedy může zapsat libovolné parametry ve formě strukturovaného textu. Vzhledem k tomu, že nemůžeme předem předpokládat, jaké parametry bude chtít uživatel používat, je toto nejflexibilnější řešení. Pro model vlastností lze dodatečně specifikovat i textový parametr udávající jeho verzi.

Model vlastností může mít také definován libovolný počet textových omezení (*Constraint*). Jejich detailní popis bude uveden až v části 6.3.

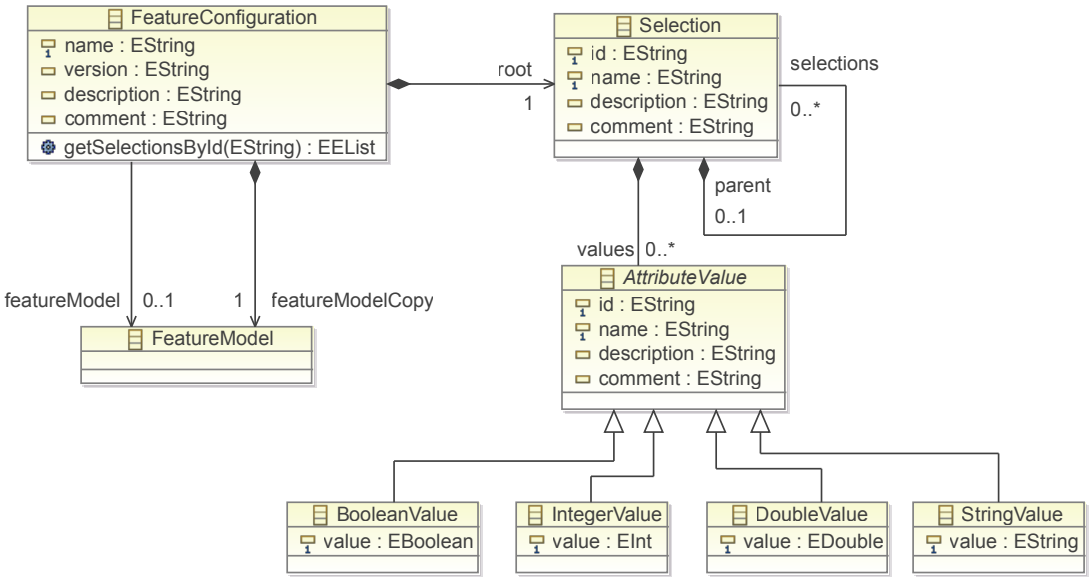
6.2.2 Konfigurace vlastností

Diagram tříd, jež popisuje strukturu meta-modelu konfigurace vlastností, je zachycen na obrázku 6.5. Vzhledem k tomu, že model vlastností může obsahovat i duplikovatelné vlastnosti, nelze konfiguraci definovat jako prostý výčet zvolených vlastností, ale jako stromovou strukturu, kopírující charakter původního modelu (viz diskuze tohoto problému v kapitole 2.6).

Jednotlivé vybrané vlastnosti jsou v konfiguraci reprezentovány třídou *Selection*. Každá taková vybraná vlastnost může mít libovolný počet potomků. Konfigurace vlastností (*FeatureConfiguration*) pak musí obsahovat jednu vybranou vlastnost, představující kořen



Obrázek 6.4: Diagram tříd modelu vlastností.



Obrázek 6.5: Diagram tříd konfigurace vlastností.

celého stromu. Propojení mezi konfigurací a modelem vlastností je provedeno pomocí identifikátorů ID, jež jsou pro vlastnost a její protějšek v konfiguraci shodné. Pokud byla do konfigurace vybrána vlastnost s atributy, jsou v konfiguraci přítomny jejich hodnoty představované abstraktní třídou *AttributeValue*. Pro každý typ atributu existuje jedna odvozená třída (*BooleanValue*, *IntegerValue*, *DoubleValue*, *StringValue*).

Uvedli jsme, že jednotlivé vybrané vlastnosti (*Selection*) jsou rozlišené pomocí ID, jehož hodnota odpovídá ID u jejich protějšku (*Feature*). Aby bylo tedy možné s konfigurací pracovat, musí mít konfigurace přístup i k datům z původního modelu vlastností, což představuje potenciální problém, neboť obojí je uchováváno ve formě samostatných souborů. Třída *FeatureConfiguration* proto obsahuje dvě položky:

1. *featureModel* - Odkaz na umístění souboru obsahující model vlastností. Umožňuje přístup k aktuální verzi modelu vlastností.
2. *featureModelCopy* - Kopie poslední známe verze modelu vlastností. Díky ní může být soubor s konfigurací používán i samostatně.

Třída *FeatureConfiguration* sice obsahuje i atribut pro specifikaci verze, avšak ten je zde přítomen z čistě uživatelského hlediska. Hodnota tohoto atributu není nijak používána k porovnání aktuálnosti verzí souboru.

6.2.3 Validace modelu

Standardně má vývojář pracující s EMF dvě možnosti jak implementovat validační mechanismus:

1. V rámci Ecore modelu lze definovat různé omezující podmínky. Při generování kódu je pak vytvořena validační třída, obsahující pro každou takovou podmínku předpřipravenou metodu. Do těchto metod pak vývojář zapíše daný validační mechanismus.
2. Použití nadstavby EMF Validation framework, fungující podobným způsobem. Zde lze omezující podmínky zapisovat přímo v rámci Ecore modelu, například v jazyce OCL. Lze také vytvářet omezení a jejich validátory, jež jsou znovupoužitelné mezi různými Ecore modely.

Druhý uvedený způsob je v našem případě zbytečně komplikovaný. Zásadním problémem prvního přístupu byla nemožnost nechat vygenerovat validační třídy v podobě samostatného modulu[5]. Dalším problémem pak bylo, že vygenerovaný validátor nedokáže validovat hodnotu zadanou jako vstup od uživatele, která ještě není součástí modelu.

Autor tedy zvolil přístup vytvoření vlastního validátoru. Validační třída pro model vlastností *FeatureModelValidator* i validační třída pro konfiguraci vlastností *FeatureConfigurationValidator* implementují standardní rozhraní *EValidator* z EMF, takže jsou použitelné stejným způsobem jako automaticky vygenerovaný validátor. Obě třídy navíc

[5]Validační třídy nemohly být součástí balíku *cz.zcu.yafmt.model*, neboť by mezi ním a *cz.zcu.yafmt.clang* vznikla cyklická závislost.

implementují i rozhraní *IStructuralFeatureValidator*, jež bylo použito při validování zadávaných uživatelských vstupů. Hlavní výhodou tohoto přístupu je, že validace modelu i uživatelských vstupů je implementována pouze na jediném místě. Ve výsledku se také nakonec ukázalo, že vygenerovaný kód by nepředstavoval až takovou výhodu, neboť byl jeho poměr k množství (ručně napsaného) validačního kódu poměrně zanedbatelný.

Pro model vlastností jsou validovány všechny základní parametry, jako například neprázdnost jmen, unikátnost ID nebo to, zda dolní mez u kardinality nepřekračuje mez horní. Pro kardinalitu skupin jsou navíc validovány i následující podmínky[6]

$$\sum_i upper(f_i) \geq lower(g) \tag{6.1}$$

$$\sum_i lower(f_i) \leq upper(g) \tag{6.2}$$

Příkladem nesplňujícím podmínku 6.1 je skupina s kardinalitou $\langle 3-4 \rangle$ obsahující dvě volitelné vlastnosti. Je zřejmé, že v rámci takové skupiny nelze nikdy vybrat 3 vlastnosti a nelze tedy ani sestrojit validní konfiguraci. Pro konfiguraci vlastností jsou validována omezení specifikovaná příslušným modelem vlastností. Tato omezení jsou dvou typů:

- *lokální* - Určené pomocí kardinality vlastností a skupin.
- *globální* - Zapsané pomocí příslušného jazyka omezení. Tato omezení budou podrobněji diskutována v podkapitole 6.3.

Zmíněné validační i pomocné třídy jsou umístěny v balíčku *cz.zcu.yafmt.model.validation*, jež je součástí stejnojmenného zásuvného modulu.

6.2.4 Integrace s uživatelským rozhraním

Základní vlastností EMF je, že ze vstupního Ecore modelu dokáže vygenerovat nejen zdrojový kód implementující daný meta-model, ale i kód některých dalších součástí. Jedna z těchto součástí, nazývána jako *EMF.Edit*, představuje mezivrstvu mezi EMF modelem a uživatelským rozhraním, usnadňující tak jejich propojení.

EMF.Edit není pevně vázáno na žádnou knihovnu pro tvorbu uživatelského rozhraní, nicméně předpokládá že bude při cílové implementaci použit návrhový vzor MVC nebo jeho obdoba. Při použití *EMF.Edit* je pro každou třídu meta-modelu vygenerována odpovídající třída jménem *NázevTřídyItemProvider*, jež slouží ke zprostředkování základních údajů o modelu vůči vrstvě uživatelského rozhraní. Příkladem takových údajů jsou například textové popisky, ikony pro jednotlivé objekty, popis parametrů, jež lze editovat apod. Programátor může kód těchto vygenerovaných tříd přizpůsobit dle svých potřeb. Napojení na uživatelské rozhraní pak probíhá adaptací zmíněných tříd. Pro zjednodušení práce nabízí EMF již hotové adaptéry pro knihovnu SWT, používanou v rámci Eclipse.

[6] g je skupina a f_i pak její jednotlivé vlastnosti. *lower* a *upper* označují dolní a horní mez kardinality.

EMF.Edit je standardně generován ve formě samostatného modulu, v našem případě *cz.zcu.yafmt.model.edit*. Mimo obvyklých úprav byl do tohoto modulu také přidán kód zajišťující validaci vstupů, využívající modul *cz.zcu.yafmt.model.validation*.

6.3 Jazyk omezení

Pro použití jazyka omezení jsme měli v podstatě dvě možnosti. Buď použít již nějaký existující jazyk (např. OCL, XPath apod.), či vytvořit jazyk zcela nový. Problémem první možnosti je případná obtížnost integrace takového jazyka do našeho nástroje. Problémem druhé možnosti je nutnost navrhnout daný jazyk tak, aby byl dostatečně expresivní a vyhovoval co největšímu počtu uživatelů.

Tento problém byl nakonec vyřešen zcela jiným způsobem, a to tak, že nástroj nepoužívá pevně zvolený jazyk omezení, ale umožňuje jejich přidání v podobě rozšiřujících zásuvných modulů. Uživatel si tak může pro každé omezení vybrat, v jakém jazyce bude zapsáno. S nástrojem jsou standardně dodávány dva jazyky, jež budou popsány dále v textu. V případě jejich nedostatečnosti může kdokoliv implementovat svůj vlastní jazyk omezení, což lze udělat skrze dvě místa rozšíření (*extension points*):

- *cz.zcu.yafmt.clang* - Umožňuje přidat nový jazyk omezení (jeho interpret).
- *cz.zcu.yafmt.clang.ui* - Umožňuje přidat specializovaný editor pro již existující jazyk omezení (například s podporou zvýrazňování syntaxe).

Uvedená místa rozšíření jsou definována v rámci zásuvného modulu *cz.zcu.yafmt.clang*. Ten také obsahuje různá podpůrná rozhraní a třídy nutné pro implementaci nového jazyka omezení. Vývojář, který by chtěl takový jazyk přidat, musí splnit dvě věci:

1. Musí v rámci svého zásuvného modulu definovat rozšíření *cz.zcu.yafmt.clang*. Ukázku jak vypadá taková definice v podobě XML můžeme vidět ve výpisu 6.1. Důležité je zde zejména ID, pomocí kterého jsou jednotlivé jazyky navzájem rozlišeny a které je pak uvedeno i v instanci třídy *Constraint*.
2. Musí implementovat rozhraní *IConstraintLanguage*. Název implementační třídy je součástí předchozí definice.

To, jak vypadají jednotlivá rozhraní tohoto rozšíření, lze vidět na obr. 6.6. Rozhraní *IConstraintLanguage* představuje tovární třídu, jež z omezení v daném jazyce vytváří jejich reprezentaci v podobě *IEvaluator*. To poté slouží k validaci daného omezení (např. zda jím odkazované vlastnosti opravdu existují) a k vyhodnocení konfigurace vlastností (zda-li splňuje dané omezení). Případně se lze pomocí *IEvaluator* dotázat na vlastnosti, jež jsou daným omezením ovlivněny[7]. *IValidationResult* a *IEvaluationResult* pak představují výsledek zmíněné validace a vyhodnocení. *IEvaluationResult* vrací, kromě výsledku a chybové zprávy, také prvky modelu, které zapříčinily porušení daného omezení. Ty jsou pak editoru

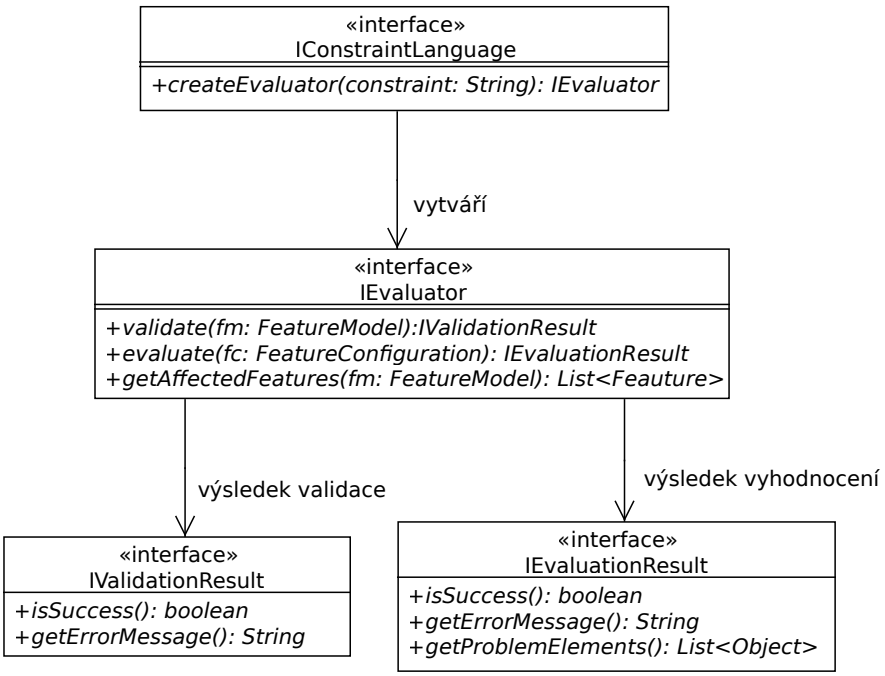
[7]Toho je využito zejména při vizualizaci modelu vlastností nebo při zvýrazňování souvisejících prvků v editoru.

pro uživatele viditelně označeny. Pro uvedená rozhraní existuje v balíčku *cz.zcu.yafmt.clang* také jejich základní implementace.

```
<extension point="cz.zcu.yafmt.clang">
  <language id="com.example.myclang"
            name="My Constraint Language"
            shortName="MyCL"
            class="com.example.myclang.MyConstraintLanguage"/>
</extension>
```

Výpis 6.1: Ukázka definice rozšíření *cz.zcu.yafmt.clang*.

Obdobným způsobem lze použít i místo rozšíření *cz.zcu.yafmt.clang.ui*, kde vývojář implementuje rozhraní *IEditingSupport* představující tovární třídu pro konstrukci vlastního editoru. Na začátku editace omezení je vždy vyhledáno, zda není implementace tohoto rozhraní pro zvolený jazyk omezení dostupná. V takovém případě je pak standardní editor nahrazen tím specializovaným.



Obrázek 6.6: Diagram tříd balíčku *cz.zcu.yafmt.clang*.

6.3.1 Sample Constraint Language

Sample Constraint Language (SamCL) je jazyk omezení, který slouží jako ukázka a návod pro vývojáře dalších jazyků. Tento jazyk je naprosto minimalistický a umožňuje vytvářet výrazy pouze dvou typů:

- `A requires B` - Vlastnost A vyžaduje vlastnost B.

- `A mutex-with B` - Nelze vybrat zároveň vlastnost *A* i *B*.

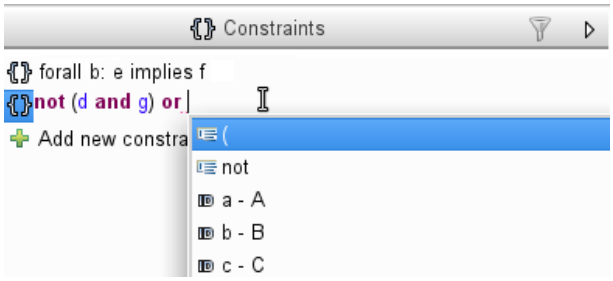
Na danou vlastnost se v rámci omezení odkazuje pomocí jejího unikátního ID. Kód tříd *SampleConstraintLanguage* a *SampleEvaluator*, implementujících SamCL, se nachází v balíčku *cz.zcu.yafmt.clang.sample*, jež standardní součástí modulu *cz.zcu.yafmt.clang*.

6.3.2 Boolean Constraint Language

Boolean Constraint Language (BoolCL) je jazyk omezení, postavený na pravidlech Booleovské algebry. Lze v něm tedy používat obvyklé logické operátory, jako je negace, logický součet a součin, implikace a ekvivalence, jež jsou zastoupeny klíčovými slovy `not`, `or`, `and`, `implies` a `equals`. Podobně jako u SamCL, i zde se lze na vlastnosti odkazovat přes jejich ID. Jednotlivé výrazy se pak dají uzavírat do závorek.

Vzhledem k tomu, že se může v konfiguraci vyskytnout i několik kopií stejné vlastnosti, zavádí jazyk BoolCL navíc klíčová slova `forall` a `exists`. Výsledkem výrazu `forall A: B` je pravda, pokud je podvýraz `B` pravdivý v kontextu každé kopie vlastnosti *A*. Obdobně je vyhodnocen i výraz `exists A: B`, kde však stačí, aby bylo `B` pravdivé alespoň jedenkrát. Kompletní gramatiku jazyka BoolCL lze nalézt v příloze B.

Pro vygenerování lexikálního a syntaktického analyzátoru tohoto jazyka byl použit framework Xtext, jež byl popsán v kapitole 5.4. Tento framework byl také použit pro vygenerování zdrojového kódu editoru jazyka BoolCL, který podporuje zvýrazňování syntaxe a automatické dokončování (*intelli-sense*), při kterém jsou uživateli nabízeny možnosti pro kompletaci klíčových slov jazyka a ID existujících vlastností (viz obr. 6.7). Z důvodu neexistující podpory frameworku Xtext pro integraci s některými prvky uživatelského rozhraní byla použita již existující komponenta XJI (Xtext JFace Integration) z projektu Yakindu[8].



Obrázek 6.7: Automatické dokončování editoru jazyka BoolCL.

Vyhodnocení omezení jazyka BoolCL probíhá průchodem stromové struktury, jež je výstupem jeho syntaktického analyzátoru. ID je ve výrazu vyhodnoceno jako pravda, pokud je v konfiguraci vlastností přítomna alespoň jedna kopie vlastnosti s tímto ID. Ostatní uzly stromu jsou pak vyhodnoceny jako jim odpovídající logické operace nad uzly nižší úrovně. Vyhodnocení výrazu s operátory `forall` a `exists` probíhá zvlášť pro každou kopii vlastnosti uvedenou jako kontext (viz logický výraz 2.2 v kapitole 2.8).

V případě, že je výsledkem vyhodnocení nepravda (dané omezení je porušeno), je skrze rozhraní *IEvaluationResult* předán seznam vlastností, jejichž výběr zapříčinil tento výsle-

[8] https://code.google.com/a/eclipselabs.org/p/yakindu/

dek. Tedy například, pokud bylo porušeno omezení `(not A) and B`, je za chybnou označena vybraná vlastnost A. Vzhledem k tomu, že vlastnost B není v konfiguraci přítomna (a nelze ji tedy označit), je chyba přiřazena k vlastnosti v jejímž kontextu byl celý výraz vyhodnocen. Takto specifikované vlastnosti jsou potom v editoru pro uživatele viditelně označeny.

Implementace jazyka BoolCL i jeho editoru jsou rozděleny do samostatných zásuvných modulů *cz.zcu.yafmt.clang.bcl* a *cz.zcu.yafmt.clang.bcl.ui*. Ty pak obsahují ve stejnojmenných balíčcích i třídy *BooleanConstraintLanguage* a *BooleanConstraintLanguageEditingSupport* implementující rozhraní *IConstraintLanguage* a *IEditingSupport*. Gramatika, ze které bylo provedeno generování kódu, se nachází v souboru *BooleanConstraintLanguage.xtext*, jež je součástí prvního uvedeného modulu.

6.4 Editor modelu vlastností

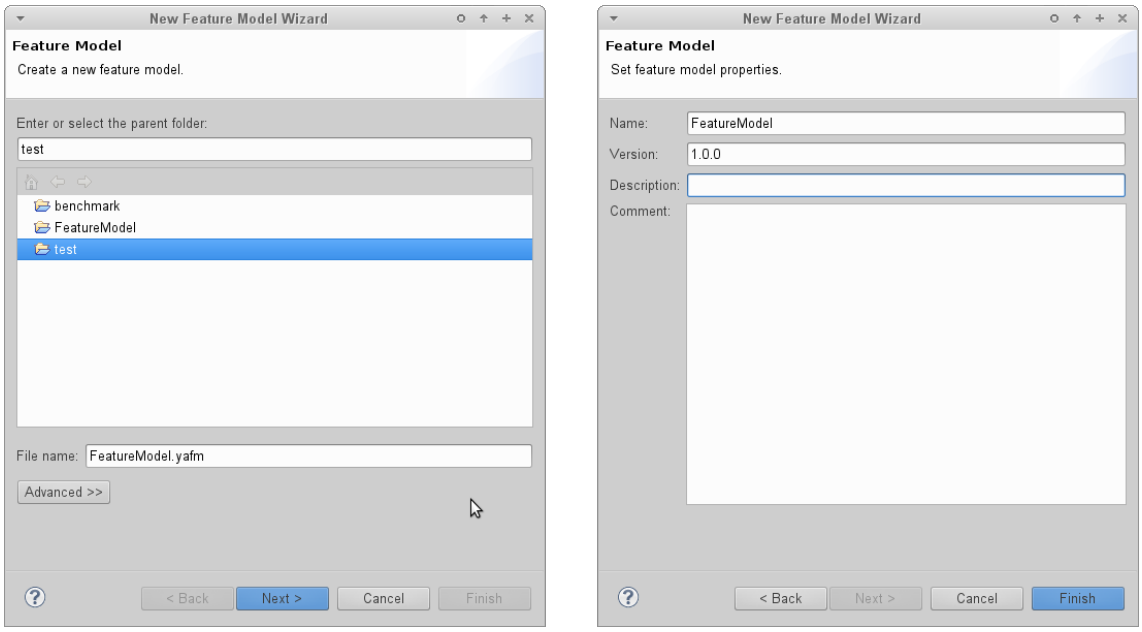
Editor modelu vlastností je realizován v podobě zásuvných modulů *cz.zcu.yafmt.ui* a *cz.zcu.yafmt.ui.editors.fm*, kde první jmenovaný obsahuje třídy sdílené společně s ostatními komponentami uživatelského rozhraní. Samotný editor je představován třídou *FeatureModelEditor* odvozenou od bázové třídy *ModelEditor*, jež je součástí právě zmíněného *cz.zcu.yafmt.ui*.

Pro implementaci editoru byl použit framework GEF, popsaný v kapitole 5.3.2. Způsob této implementace je poměrně přímočarý a víceméně odpovídá tomu, co jsme uvedli ve zmíněné kapitole. Proto bude dále uveden hlavně popis funkcionality s případnými důležitými implementačními detaily. Pro přehled zde ještě však uvedeme strukturu balíčků zásuvného modulu *cz.zcu.yafmt.ui.editors.fm* a jejich popis:

cz.zcu.yafmt.ui.editor.fm	- základní třídy editoru
cz.zcu.yafmt.ui.editor.fm.actions	- akce uživatelského rozhraní
cz.zcu.yafmt.ui.editor.fm.commands	- příkazy zapouzdřující operace nad modelem
cz.zcu.yafmt.ui.editor.fm.figures	- obrazce pro vykreslení diagramu
cz.zcu.yafmt.ui.editor.fm.layout	- třídy pro uchování rozložení diagramu
cz.zcu.yafmt.ui.editor.fm.parts	- implementace *editparts*
cz.zcu.yafmt.ui.editor.fm.policies	- implementace editačních politik
cz.zcu.yafmt.ui.editor.fm.utils	- pomocné třídy
cz.zcu.yafmt.ui.editor.fm.wizards	- okna průvodců

6.4.1 Vytvoření nového modelu vlastností

Aby mohl uživatel vytvořit soubor s novým modelem vlastností, má k dispozici připraveného průvodce (obr. 6.8) implementovaného třídou *NewFeatureModelWizard*. Tento průvodce se skládá ze dvou stránek, z nichž první slouží ke specifikaci jména souboru a nadřazeného projektu a druhá je pak určena ke specifikaci dodatečných parametrů. Po dokončení tohoto průvodce je uživateli zobrazen editor s vytvořeným souborem (obr. 6.9).



Obrázek 6.8: Okno průvodce pro vytvoření nového modelu vlastností.

6.4.2 Základní funkcionalita editoru

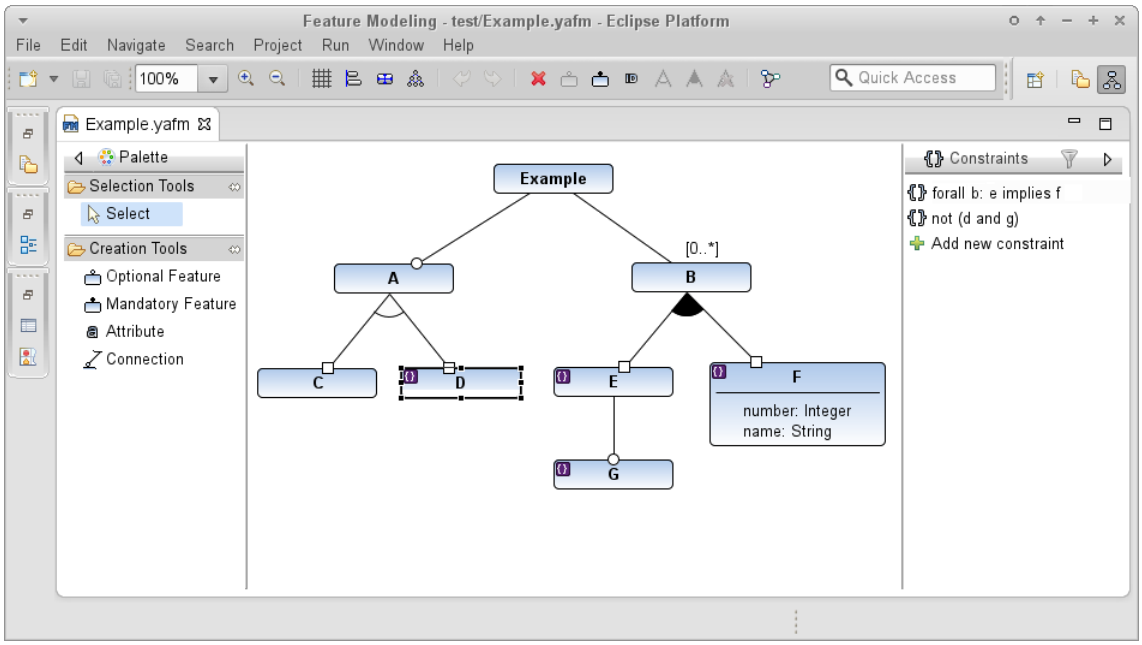
Okno editoru (obr. 6.9) je rozděleno do tří částí:

1. Paleta nástrojů v levé části.
2. Hlavní editační plocha uprostřed.
3. Editor omezení v pravé části.

Paleta nástrojů slouží uživateli k přidávání nových vlastností a jejich atributů do digramu, čehož je možné docílit jejich přetažením do editační plochy. Součástí palety je také nástroj pro propojování jednotlivých vlastností ve vztahu rodič-potomek. Samotnou paletu nástrojů i editor omezení lze pomocí příslušného tlačítka dočasně skrýt, takže uživatel může využít celý prostor obrazovky.

Pro práci s hlavním částí editoru jsou k dispozici všechny obvyklé prvky, jako je nástrojová lišta, kontextové menu či klávesové zkratky, pomocí kterých lze například seskupit vybrané vlastnosti nebo změnit jejich kardinalitu. Kromě nich má editor navíc podporu i pro následující prvky, jejichž cílem je usnadnit práci uživatele:

- *Přímá editace* - Uživateli se po dvojkliku na vlastnost v diagramu objeví pole ve kterém může zadat jméno této vlastnosti. Obdobně může uživatel zadat jméno a typ atributu.
- *Drag and drop* - Pokud uživatel uchopí pomocí myši atribut, může měnit jeho pozici v rámci dané vlastnosti či ho přesunout k vlastnosti zcela jiné. Obdobným způsobem lze přepojovat i konce spojení mezi jednotlivými vlastnostmi. Pokud tedy uživatel

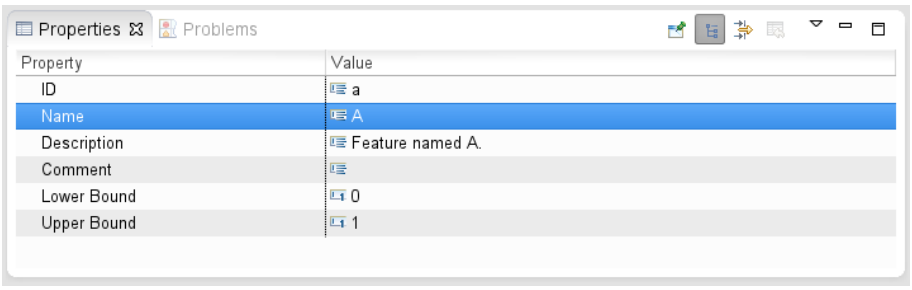


Obrázek 6.9: Editor modelu vlastností.

potřebuje například změnit rodiče vlastnosti, nemusí mazat stávající spojení a vytvářet nové, ale provede pouze přepojení na straně rodiče.

Pro pokročilejší úpravy je využito standardního pohledu Eclipse s názvem *properties view*, který umožňuje editovat parametry vybraného objektu (viz obr. 6.10). Implementace obsahu *properties view* je realizována třídou *EditorPropertySheetPage* z *cz.zcu.yafmt.ui*, jež je pak použita i pro editor konfigurace vlastností.

U všech doposud i dále uvedených uživatelských akcí platí, že je lze vždy vrátit nebo opětovně vykonat pomocí příslušných ovládacích prvků (*undo*/*redo*). Toto je možné zejména díky použití návrhového vzoru *command* pro zapouzdření veškerých operací prováděných nad modelem, jak jsme již uvedli v kapitole o frameworku GEF. Na tomto místě se ukázala další výhoda EMF, neboť umožňuje automaticky zaznamenávat veškeré změny prováděné na jednom či více objektech EMF modelu pomocí třídy *ChangeRecorder*. Implementace *undo* a *redo* pak spočívala pouze v zavolání příslušných metod této třídy.



Obrázek 6.10: Pohled pro editaci parametrů vybraného objektu (*properties view*).

6.4.3 Rozvržení prvků diagramu

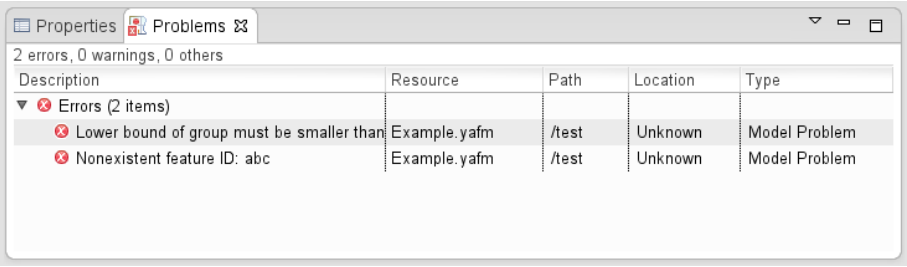
Jednotlivé prvky diagramu lze v rámci editační plochy libovolně pozicovat či měnit jejich velikost. Pro urychlení této činnosti jsou uživateli (skrze nástrojovou lištu) k dispozici akce pro nastavení optimální velikosti objektu a pro automatické rozvržení prvků diagramu do stromové struktury. Případně může uživatel také povolit automatické zarovnávání objektů do mřížky či automatické zarovnávání vůči ostatním objektů.

Údaje o tom, jaká je pozice a velikost každého objektu, nejsou uloženy jako součást modelu vlastností, ale jsou umístěny v odděleném souboru[9], aby nedocházelo k promíchání datových a prezentačních informací. Pokud není při otevření editoru tento soubor nalezen, jsou prvky diagramu automaticky uspořádány do stromu. Pro uchování těchto prezentační informací byl použit samostatný EMF model *ModelLayout.ecore*, který je provázán s modelem vlastností[10].

6.4.4 Validace modelu

Na rozdíl od jiných nástrojů, kde musí uživatel proces validace spustit ručně, provádí editor nástroje YAFMT validaci automaticky. Tedy, pokud uživatel upraví část modelu (např. změní kardinalitu vlastnosti), je tato část modelu ihned revalidována a uživateli jsou zobrazeny případné chyby. Pokud uživatel používá k editaci *properties view* nebo editor omezení, je validace prováděna už při zadávání vstupu a příslušná chyba je zobrazována ve stavové liště okna Eclipse. Ačkoliv je tento automatický způsob validace náročnější na implementaci, nabízí pro uživatele lepší zpětnou vazbu. Pro samotnou detekci změny modelu před validací je použit návrhový vzor *observer*, jež je standardní součástí EMF.

Veškeré chyby modelu jsou zobrazitelné skrze standardní pohled Eclipse jménem *problems view* (obr. 6.11). Pokud uživatel provede dvojklik na řádek s chybou v tomto pohledu, je v okně editoru vybrán odpovídající (nevalidní) objekt. Veškeré nevalidní objekty jsou navíc zřetelně označeny i v samotném diagramu. Pro správu validačních chyb používá editor rozhraní *IProblemManager*, jehož implementaci *ResourceProblemManager* uchovává bázová třída editorů *ModelEditor*.



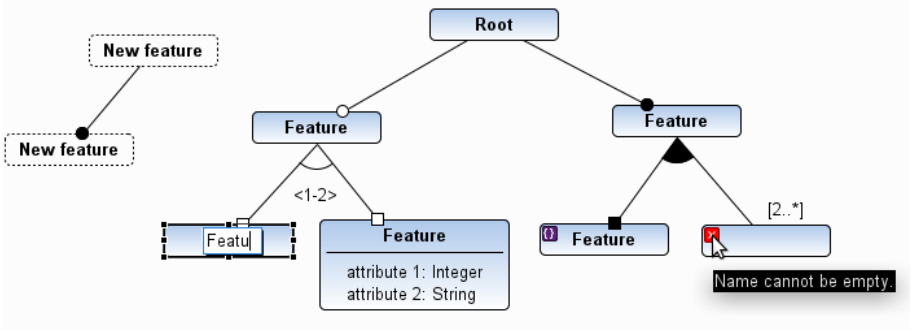
Obrázek 6.11: Pohled pro zobrazení chyb v modelu (*problems view*).

[9] Jeho jméno je odvozeno od jména souboru modelu vlastností, přidáním přípony *.layout*.
[10] Toto provázání je automaticky zajištěno frameworkem EMF a je pro programátora zcela transparentní.

6.4.5 Vizuální nápověda

Aby se mohl uživatel při tvorbě diagramu lépe zorientovat, používá editor různé vizuální prvky, jež můžeme vidět na obrázku 6.12. Všechny vlastnosti, jež nejsou součástí hlavního stromu modelu vlastností, jsou vykresleny bez výplně a s čárkovaným okrajem. Uživatel tak snadno například pozná, že danou vlastnost zapomněl do hlavního stromu připojit. Pokud je vlastnost ovlivněna nějakým omezením, objevuje se v jejím levém horním rohu dekorace v podobě symbolu `{}` na fialovém pozadí. Obdobně jsou označeny i všechny chybné objekty, a to pomocí červené dekorace s křížkem. Po najetí kurzorem na tuto dekoraci je navíc uživateli zobrazen i popisek obsahující text dané chyby.

Kromě výše zmíněných prvků můžeme na obrázku také vidět, jak vypadá použitá notace diagramu vlastností. Tuto notaci jsme již dříve popsali v kapitole 2.5, avšak zde k ní přibyly drobné modifikace, jako například zobrazení atributů vlastností podobným stylem, jako je tomu u diagramu tříd UML. Na uvedeném obrázku je také vidět, jak vypadá přímá editace jména vlastnosti (první modrý uzel zleva).



Obrázek 6.12: Vizuální prvky používané v editoru modelu vlastností.

6.4.6 Pomocné pohledy

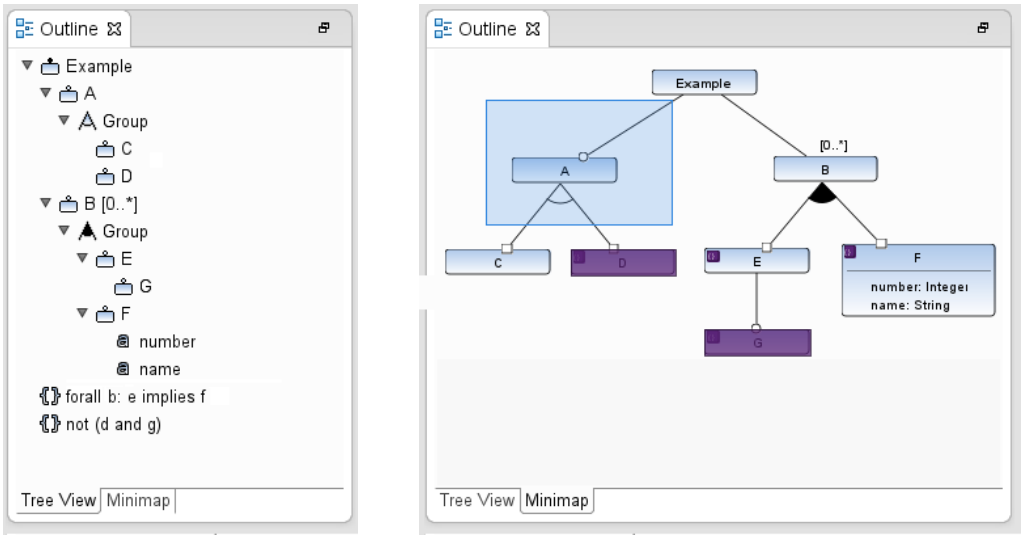
Jeden z požadavků, který jsme kladli na vytvářený editor, byla dostupnost pomocných pohledů, které by uživateli umožnily rychle se zorientovat v rozsáhlém diagramu. Součástí nástroje YAFMT jsou dva takové pohledy, a to pohled pro vizualizaci modelu vlastností, jež bude popsán samostatně v podkapitole 6.6, a tzv. *outline view*.

Druhý jmenovaný je opět standardním pohledem v rámci Eclipse a slouží jako náhled na aktuální obsah editoru. To, jak vypadá tento pohled pro model vlastností, můžeme vidět na obrázku 6.13. První karta pohledu zobrazuje stromovou strukturu editovaného diagramu s možností zabalování a rozbalování jeho jednotlivých částí. Výběry mezi tímto stromovým pohledem a hlavním editorem jsou navíc synchronizovány, takže když uživatel vybere objekt v editoru, je vybrán i v pohledu a naopak. Pokud byl vybrán objekt, jež je mimo viditelnou část diagramu, dojde na něj v editoru k automatickému vycentrování.

Druhá karta *outline view* pak obsahuje náhled v podobě zmenšeného diagramu, jež je vhodný zejména v případě, že uživatel použil v editoru funkci přiblížení (*zoom*), nebo v případě že se celý diagram do okna editoru nevejde. Tento náhled má také jednu speciální funkci, a to že dokáže barevně vyznačit které vlastnosti jsou ovlivněny zvoleným omezení

(viz fialově obarvené vlastnosti na obr. 6.13b). Tato funkcionalita je poměrně zásadní, neboť omezení jsou často definována pro vlastnosti umístěné v různě vzdálených částech diagramu a není snadné je poté dohledat.

Způsob, jakým editor přispívá do *outline view*, je třída *FeatureModelEditorContentOutlinePage*. Ta je odvozena od bázové třídy *EditorContentOutlinePage* z *cz.zcu.yafmt.ui*, jež je používána i pro editor konfigurace.



(a) Stromový pohled. (b) Zmenšený náhled na diagram.

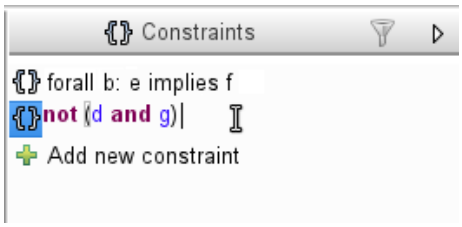
Obrázek 6.13: Pohled na editovaný model vlastností v *outline view*.

6.4.7 Editor omezení

Součástí hlavního editoru je také editor omezení, realizovaný jako samostatný panel na jeho pravé straně (viz předchozí obrázek 6.9). Jednotlivá omezení jsou v tomto editoru zobrazená v podobě seznamu a je možné je editovat po dvojkliku myší. Stejným způsobem lze přidat i nové omezení, a to dvojklikem na poslední (vyhrazenou) položku seznamu. Při editaci je na daném místě seznamu zobrazen řádkový editor obsahující text omezení. Tento editor je možné pro zvolený jazyk omezení nahradit specializovaným editorem pomocí implementace rozšíření popsaného v podkapitole 6.3. Ukázku, jak takový specializovaný editor s podporou zvýrazňování syntaxe vypadá, můžeme vidět na obrázku 6.14.

Editor kromě výše popsaného také podporuje standardní klávesové zkratky a má vlastní kontextové menu. Detaily vybraných omezení se navíc dají také editovat přes *properties view*. Zajímavou funkcí editoru omezení je možnost nechat si vyfiltrovat seznam omezení, dle vybrané vlastnosti z hlavním editoru. Pokud je tento filtr povolen, jsou zobrazena pouze omezení ovliňující vybrané vlastnosti, což může být výhodné, je-li kompletní seznam omezení velmi dlouhý.

Samotný editor je implementován v podobě třídy *ConstraintsEditor*, jež pak třída *FeatureModelEditor* interně používá. Pří práci s omezeními je velmi často potřeba přistupovat k informacím o souvisejících vlastnostech a omezeních (viz například zmíněná filtrační



Obrázek 6.14: Editor omezení s podporou zvýraznění syntaxe.

funkce editoru omezení). Z výkonnostních důvodu byla proto vytvořena třída *ConstraintCache*, jež zmíněné informace uchovává. *ConstraintCache* sleduje veškeré změny v modelu a v případně některých změn znevalidní svůj obsah. K obnovení jejího obsahu pak dojde automaticky až při prvním dotazu na data o souvisejících vlastnostech a omezeních.

6.4.8 Ostatní funkcionalita

Jak jsme v předchozím textu viděli, při editaci modelu vlastností jsou často používány některé specifické pohledy (*properties view*, *problems view* a *outline view*). Tyto pohledy však nebývají obvykle zobrazeny jako výchozí, a proto je součástí nástroje YAFMT i nová perspektiva, nazvaná *Feature Modeling*, která nabízí standardní rozložení okna obsahující výše zmíněné pohledy. Tato perspektiva je vytvářena tovární třídou *FeatureModelingPerspectiveFactory* z *cz.zcu.yafmt.ui*.

Poměrně důležitou vlastností z hlediska uživatele je také možnost vyexportovat zobrazený diagram v podobě obrázku. Tato funkcionalita je v editoru dostupná skrze kontextové menu a umožňuje export diagramu ve formátech *jpg*, *pgn* a *gif*. Zajištění exportu do vektorového formátu by bylo poměrně problematické a znamenalo by závislost na několika dalších (poměrně rozsáhlých) knihovnách a modulech, a proto není implementováno. Uživatel může nicméně použít například specializovaný zásuvný modul *eclipse-gef-imageexport*[11], jež umožňuje export obrázku z libovolného GEF editoru do celé škály formátů.

6.5 Editor konfigurace vlastností

Stejně jako editor vlastností, i editor konfigurace byl vytvořen za pomoci frameworku GEF. Tento editor je implementován v podobě zásuvného modulu *cz.zcu.yafmt.ui.editors.fc*, jež opět využívá sdílený modul *cz.zcu.yafmt.ui*. Hlavní třída editoru *FeatureConfigurationEditor* je, obdobně jako u předchozího editoru, odvozena od bázové *ModelEditor*. Samotný způsob implementace je také v podstatě identický, včetně struktury balíčků, kterou zde proto nemusíme uvádět.

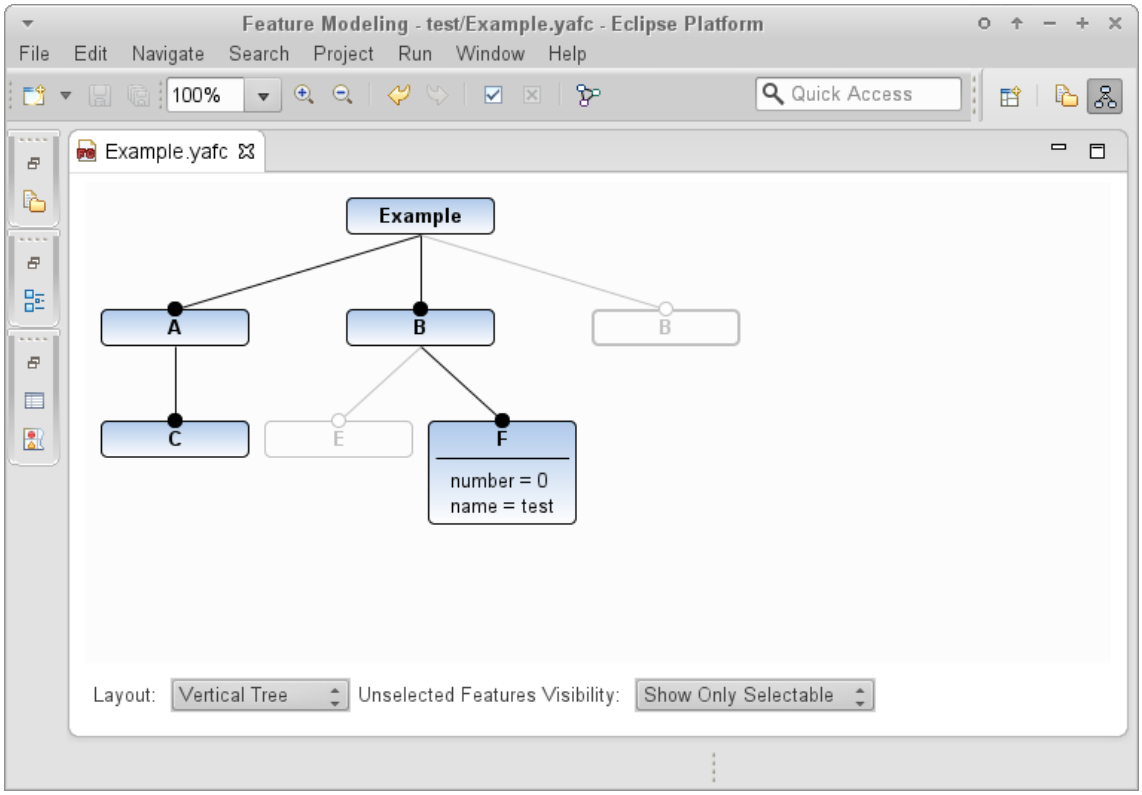
Na rozdíl od většiny ostatních nástrojů, které používaly pro tvorbu konfigurace vlastností stromový editor se zatrháváním voleb, je náš editor postaven na konstrukci grafického diagramu, jak uvidíme dále v textu.

[11] `https://github.com/veger/eclipse-gef-imageexport/`

6.5.1 Vytvoření nové konfigurace vlastností

Tvorba nové konfigurace vlastností probíhá opět pomocí průvodce (třída *NewFeatureConfigurationWizard*), ne nepodobnému tomu, jaký jsme viděli na obrázku 6.8. Jediným rozdílem je zde nutnost specifikovat odpovídající model vlastností, pro který bude konfigurace vytvořena. Po vytvoření souboru s výchozí konfigurací vlastností je uživateli otevřen samotný editor, který můžeme vidět na obrázku 6.15.

6.5.2 Základní funkcionalita editoru



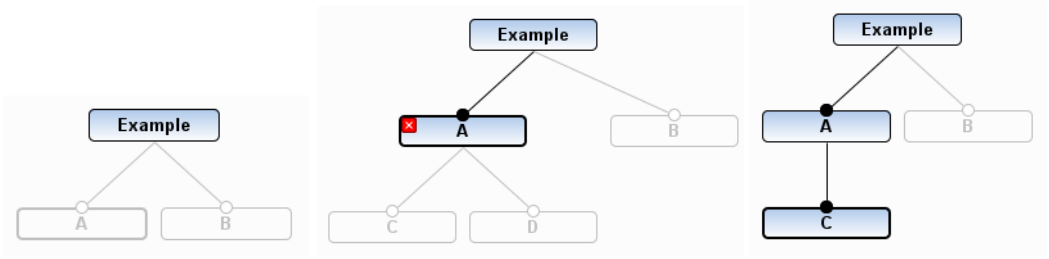
Obrázek 6.15: Editor konfigurace vlastností.

Struktura tohoto editoru je poměrně jednoduší než tomu bylo u editoru modelu vlastností a je tvořena pouze hlavní editační plochou a dolní panelem, obsahujícím různé volby pro úpravu zobrazení. Editace konfigurace probíhá skrze konstrukci diagramu, jež svojí strukturou odpovídá diagramu vlastností. Uživateli je zde povoleno pouze přidávat a ubírat jednotlivé vlastnosti tak, aby nebyla porušena pravidla specifikovaná původním modelem vlastností.

Ukázku tvorby konfigurace[12] můžeme vidět na obrázku 6.16. Na začátku konfigurace (obr. 6.16a) má uživatel možnost vybrat pouze mezi dvěma volitelnými vlastnostmi *A* a *B*. Po přidání *A* do konfigurace jsou mu okamžitě ukázány další možnosti, kterými může

[12]Model vlastností, pro který je konfigurace vytvářena, jsme mohli vidět na obrázku 6.9.

pokračovat (obr. 6.16b). Po zvolení jedné ze dvou alternativních vlastností C a D je mu druhá (nevalidní) možnost automaticky zablokována (obr. 6.16c)[13].



(a) (b) (c)

Obrázek 6.16: Způsob úpravy konfigurace v nástroji YAFMT.

Tento způsob konfigurace je pro uživatele poměrně výhodný, neboť mu postupně odkrývá povolené možnosti a skrývá před ním ty nevalidní, takže i konfigurace rozsáhlého modelu vlastností je poměrně intuitivní. To, které volby jsou při konfiguraci skryty, je dáno kardinalitou jednotlivých vlastností a skupin (lokální omezení). Skrývání neplatných voleb nevyhovujících globálním omezením není podporováno z důvodu netriviální implementace. Pokud by totiž například globální omezení definovalo cyklickou závislost mezi vlastnostmi, bylo by poměrně obtížné říci, zda tyto vlastnost před uživatelem skrýt či ne.

Samotné přidávání a ubírání vlastností v konfiguraci probíhá dvojklikem myší na daný uzel diagramu. K této činnosti lze také použít i kontextové menu, nástrojovou lištu či klávesové zkratky, kdy lze manipulovat i s více objekty najednou. Pokud má vybraná vlastnost nějaké atributy, dá se jejich hodnota po dvojkliku myší přímo editovat. Parametry jednotlivých objektů se také dají nastavovat skrze *properties view*.

Poměrně zásadním problémem při takovéto konstrukci editoru bylo, jakým způsobem uchovávat informace o vlastnostech, jež má uživatel možnost teprve vybrat (zašedlé volby), neboť ty ještě nejsou přímou součástí EMF modelu[14]. Z tohoto důvodu byla vytvořena třída *FeatureConfigurationManager*, jež na základě lokálních omezení tuto množinu vlastností vytváří a uchovává. Tato třída navíc zapouzdřuje veškeré operace prováděné nad konfigurací vlastností z důvodu ošetření přístupu k těmto vlastnostem.

Validace konfigurace probíhá stejným způsobem jako u tvorby modelu vlastností, tedy automaticky po každé provedené úpravě. Vlastnosti, jejichž výběr porušil definovaná omezení, jsou viditelně označeny symbolem křížku v červeném pozadí (viz obr. 6.16b), jež po zaměření kurzorem myši poskytne místní nápovědu s popisem dané chyby. Způsob určení těchto chybných elementů jsme již diskutovali v podkapitolách 6.3 a 6.3.2. Správa validačních chyb je opět řešena skrze rozhraní *IProblemManager*.

Stejně jako pro editor konfigurace, i zde lze použít identický pohled *outline view* (třída *EditorContentOutlinePage*) a pohled pro vizualizaci modelu vlastností.

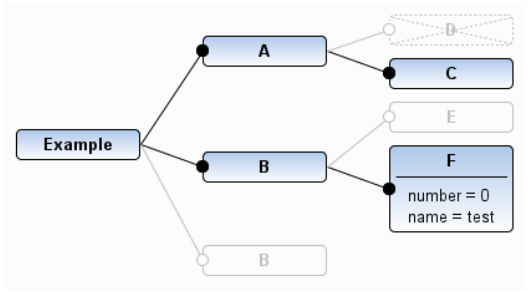
[13]Červené chybové zvýraznění na prostředním obrázku upozorňuje uživatele na nutnost vybrat alespoň jednu z těchto alternativních vlastností.

[14]Toto je právě jeden z důvodů proč bylo poněkud problematické použít framework GMF namísto GEF.

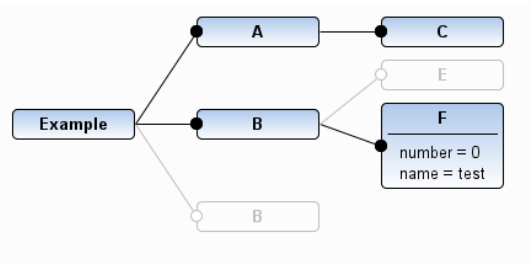
6.5.3 Rozložení prvků diagramu

Jak jsme na uvedeném obrázku 6.16 již viděli, jednotlivé prvky diagramu jsou jsou automaticky pozicovány dle zvoleného schématu (vertikálně orientovaný strom). Nepředpokládáme zde, že by měl uživatel potřebu rozmísťovat objekty dle svého vlastního uvážení, neboť mu jde hlavně o samotný proces konfigurace. Kromě zmíněného schéma rozvržení je k dispozici i druhé (horizontálně orientovaný strom), jež je vhodné zejména pokud je strom diagramu velmi široký. Mezi oběma schématy se dá přepnout pomocí rozbalovacího seznamu v dolní části editoru.

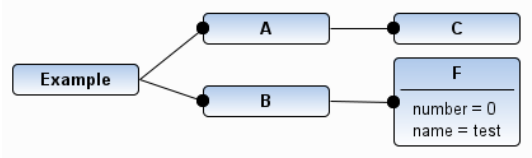
Druhé schéma rozložení je možné vidět na obrázku 6.17, jež také ukazuje různé možnosti zobrazení dostupných voleb konfigurace. Standardně jsou v diagramu zobrazeny pouze validní volby (obr. 6.17b), avšak uživatel může pomocí příslušného rozbalovacího seznamu povolit zobrazení voleb všech (obr. 6.17a) či tyto volby zcela skrýt (obr. 6.17c).



(a) Zobrazení všech voleb.



(b) Skrytí nevalidních voleb.



(c) Skrytí všech voleb.

Obrázek 6.17: Změna viditelnosti jednotlivých prvků diagramu při konfiguraci.

Zmíněná schémata rozložení jsou představována třídami *VertivalTreeLayout* a *HorizontalTreeLayout*, odvozených od bázové třídy *TreeLayout*. Jejím dalším odvozením by bylo možné (zatím pouze programově) přidat i jiná rozložení, například radiálně orientovaný strom. Tato funkcionalita je tedy dobrým kandidátem na místo rozšíření (*extension point*) do budoucna. Různá viditelnost jednotlivých voleb konfigurace je řešena jejich povolením či zneplatněním uvnitř již zmíněné třídy *FeatureConfigurationManager*.

6.5.4 Synchronizace s modelem vlastností

Konfigurace vlastností je vždy vytvářena dle odpovídajícího modelu vlastností. Pokud by došlo k tomu, že by uživatel tento model modifikoval, měly by se provedené změny promítnout i existující konfigurace. Navíc je nutné brát v úvahu, že se model i konfigurace vlastností se nacházejí v oddělených souborech.

Způsob propojení těchto souborů jsme již popsali v kapitole 6.2.2, kde jsme uvedli,

že konfigurace vlastností obsahuje odkaz[15] na soubor s modelem vlastností a zároveň také obsahuje plnou kopii tohoto modelu. K zajištění konzistence pak dochází vždy při otevření souboru ve *FeatureConfigurationEditor*.

1. Je zkontrolováno, zda odkazovaný soubor s modelem vlastností opravdu existuje. Pokud ne, je na to uživatel pomocí dialogového okna upozorněn a je mu dána možnost tento soubor vybrat.
2. Pokud je výše zmíněný soubor dostupný, je jeho obsah porovnán s uchovanou kopií modelu vlastností. V případě, že se obě dvě verze modelu vlastností liší, je na to uživatel pomocí dialogového okna upozorněn a je mu dána možnost tyto změny buď zahrnout nebo je ignorovat. Pokud obsahuje nová verze modelu vlastností validační chyby, je součástí zobrazeného dialogového okna také výstraha.
3. Jako poslední krok je zkontrolována i validita aktuální verze modelu vlastností. V případě validačních chyb jsou tyto chyby uživateli zobrazeny v dialogovém okně dohromady s varováním.

Promítnutí změn z nového verze modelu vlastností do stávající konfigurace je řešeno poměrně primitivním algoritmem. Ten, velmi zjednodušeně řečeno, porovnává ID vlastností (*Feature*) a jejich protějšků v konfiguraci (*Selection*). V případě že se vlastnost s daným ID v modelu vlastností již nevyskytuje, je daná vlastnost odstraněna i z konfigurace. Obdobně je řešen případ, kdy k došlo ke snížení horní hranice kardinality vlastnosti. Jednoduché změny, jako rozdílný typ atributu, se řeší pouhým přepsáním v konfiguraci. Tento algoritmus také zajišťuje, že počet kopií vlastnosti v konfiguraci není menší než dolní hranice její kardinality.

Zmíněný algoritmus promítnutí změn je součástí pomocné třídy *FeatureConfigurationUtil*. Algoritmus pro porovnání dvou modelů vlastností je implementován v jejím protějšku jménem *FeatureModelUtil*. Pro samotné porovnání modelů by šlo také použít nadstavbu EMF, framework EMF Compare, nicméně by to pro nástroj představovalo poměrně zbytečnou závislost, neboť samotná implementace ve *FeatureModelUtil* je velmi triviální.

6.6 Vizualizér modelu vlastností

Vizualizér modelu vlastností představuje poslední z komponent nástroje YAFMT. Tato komponenta, jež je dostupná v podobě zásuvného modulu *cz.zcu.yafmt.ui.views.fm*, představuje alternativní pohled na editovaný model vlastností. Základním účelem tohoto pohledu je zejména pomoc uživateli zorientovat se v rozsáhlém modelu vlastností a umožnit mu v něm nalézt vztahy, které nebyly na první pohled zřejmé. Pro tvorbu tohoto pohledu byla použita knihovna Zest, jež jsme popsali v kapitole 5.3.3.

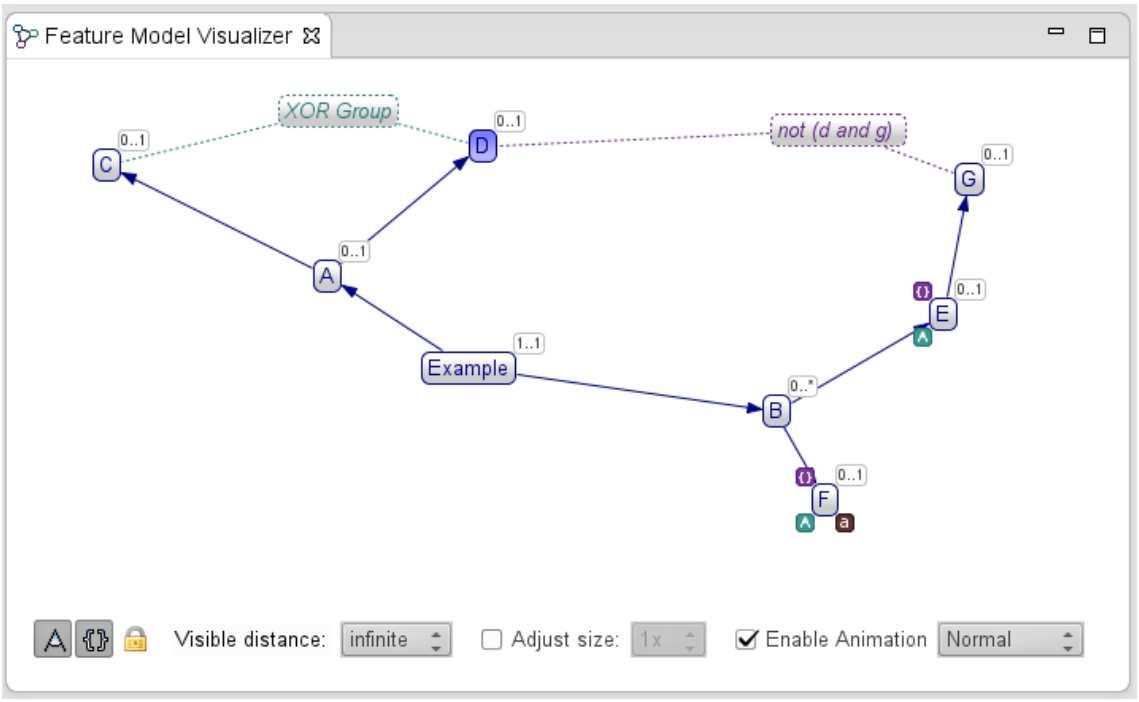
Hlavní třídou vizualizéru je třída *FeatureModelVisualizer*, nacházející se v balíčku *cz.zcu.yafmt.ui.views.fm*. Detailní struktura zmíněného balíčku vypadá následovně:

[15] Odkazem je myšlena relativní cesta v souborovém systému k danému souboru.

cz.zcu.yafmt.ui.views.fm	- základní třídy vizualizace
cz.zcu.yafmt.ui.views.fm.actions	- akce uživatelského rozhraní
cz.zcu.yafmt.ui.views.fm.decorations	- dekorace pro uzly grafu
cz.zcu.yafmt.ui.views.fm.figures	- grafické obrazce uzlů grafu
cz.zcu.yafmt.ui.views.fm.filters	- filtry viditelných částí grafu
cz.zcu.yafmt.ui.views.fm.graph	- nadstavba tříd pro zobrazení grafu
cz.zcu.yafmt.ui.views.fm.setttings	- nastavení vizualizace
cz.zcu.yafmt.ui.views.fm.util	- pomocné třídy

6.6.1 Základní funkce

To, jak vypadá okno vizualizéru, můžeme vidět na obrázku 6.18. Hlavní část tohoto okna tvoří plocha, kde je zobrazen aktuálně editovaný model vlastností. V dolní části je pak umístěn samostatný panel obsahující různé prvky pro nastavení pohledu.



Obrázek 6.18: Vizualizace modelu vlastností.

Co jsi můžeme na uvedeném obrázku všimnou je, že model vlastností zde není zobrazen jako strom, ale jako grafová struktura, jejíž hlavní kostra (tmavě modré hrany a vrcholy) představuje diagram vlastností. Jednotlivé uzly této kostry odpovídají vlastnostem, ostatní uzly mimo kostru pak odpovídají skupinám (zelené obarvení) a omezením (fialové obarvení). Uzly jsou v tomto grafu propojeny hranou pouze tehdy, pokud je mezi nimi v diagramu vlastností nějaká souvislost. Tedy například pokud jde o rodičovskou vlastnost a jejího potomka či o skupinu a v ní nacházející se vlastnost.

Každý uzel grafu zobrazuje relevantní informace o reprezentovaném objektu, jako například jméno vlastnosti, typ skupiny nebo hodnotu daného omezení. Kromě toho může

Dekorace	Pozice	Význam
	Vpravo nahoře	Kardinalita vlastnosti.
	Vpravo dole	Vlastnost má jeden nebo více atributů.
	Vlevo dole	Vlastnost je součástí skupiny, jež však není v pohledu zobrazena.
	Vlevo nahoře	Vlastnost je ovlivněna omezením, jež však není v pohledu zobrazeno.

Tabulka 6.1: Popis grafických symbolů ve vizualizaci modelu vlastností.

být navíc každý uzel označen různými symboly (dekoracemi), jejichž význam je uveden v tabulce 6.1. Po zaměření kurzorem myši na odpovídající prvek je uživateli zobrazena i místní nápověda s krátkým popisem prvku.

Standardně tento pohled zobrazuje pouze základní strom diagramu vlastností. Pokud však uživatel vybere v hlavním editoru například vlastnost, jež je součástí nějaké skupiny, stane se tato skupina také součástí pohledu. Obdobné pravidlo platí i pro všechna omezení ovlivňující vybranou vlastnost. Jinými slovy, obsah vizualizéru je vždy odvozen od aktuálního výběru v editoru nebo jiné aktivní komponenty (např. *outline view*). Toto je důležité zejména proto, že zobrazení kompletního grafu, obsahujícího všechny skupiny a omezení, by bylo pro uživatele poměrně nepřehledné.

Samotná synchronizace výběrů mezi vizualizérem a ostatními komponentami funguje i opačným směrem, tedy pokud uživatel vybere v rámci vizualizace uzel, je odpovídající prvek vybrán i v hlavním editoru či v *outline view*.

Pro zobrazení grafu je v rámci Zest používána třída *GraphViewer*, nicméně ta má několik zásadních omezení. Jedním z nich je například nemožnost vykreslovat vlastní obsah mimo plochu zobrazených vrcholů[16], což byl problém pro zajištění vykreslování jejich dekorací. Proto byla odvozením ze zmíněné třídy vytvořena nadstavba jménem *DecoratableGraphViewer*, která do kreslící plochy přidává speciální vrstvu[17] pouze pro samotné dekorace. Jednotlivé dekorace jsou potom vytvářeny jako libovolné obrazce (*figures*), které implementují rozhraní *IDecoration*. Obrazce, jimž je možné tyto dekorace přiřazovat, jsou odvozovány od třídy *DecoratableFigure*.

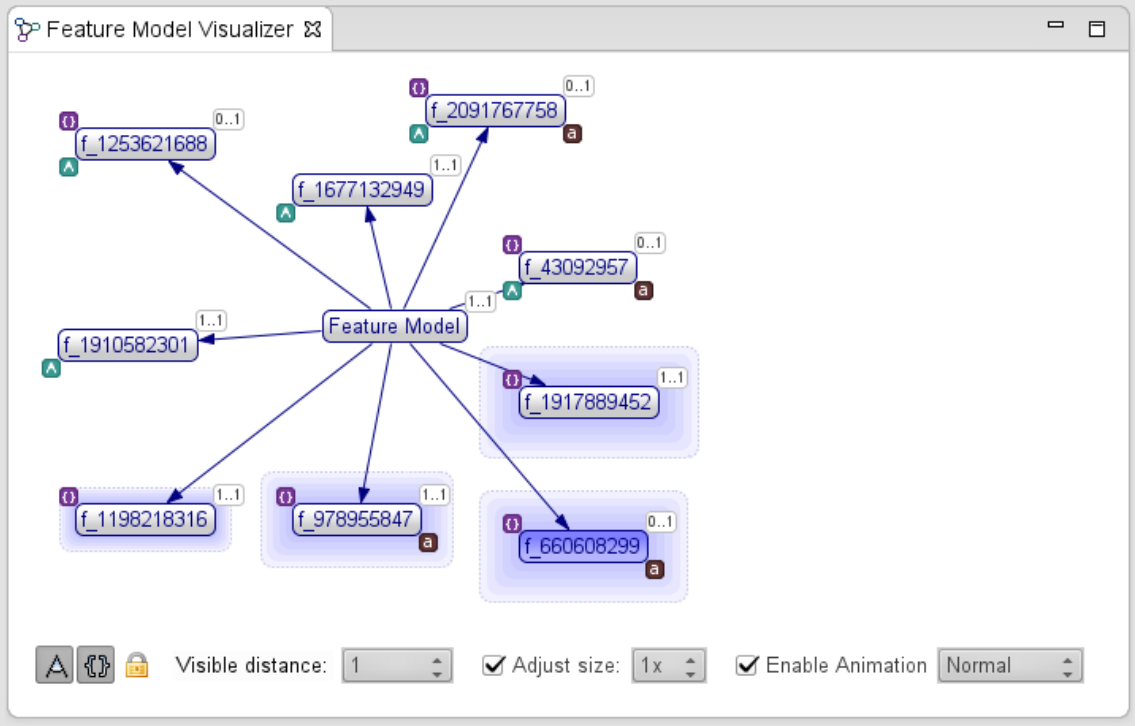
6.6.2 Pokročilé funkce

Dolní panel okna vizualizace obsahuje některé ovládací prvky, jimiž se dá ovlivnit zobrazovaný obsah pohledu. První dvě tlačítka slouží k povolení či zákazu zobrazení uzlů skupin a omezení. Tento zákaz je realizován aktivací tříd filtrů *GroupFilter* a *ConstraintFilter* při zobrazení grafu. Třetí tlačítko pak umožňuje celý pohled uzamknout (výběr z hlavního editoru neovlivní zobrazované objekty).

Poměrně důležitým ovládacím prvkem je první rozbalovací seznam, který umožňuje změnit zobrazovaný rozsah grafu. Standardně není tento rozsah nijak omezen, což však představuje zásadní problém při vizualizaci rozsáhlého modelu vlastností. Pomocí zmíně-

[16] Zde jde spíše o omezení Draw2d, které se tímto omezením kreslící plochy snaží zvýšit výkon.
[17] Ve skutečnosti jsou tyto vrstvy dvě. Jedna pro dekorace na popředí a druhá pro dekorace na pozadí.

ného menu je tedy možné nastavit, v jaké vzdálenosti od vybraného uzlu budou ještě zobrazeny uzly sousední. Ukázku toho, jak vypadá omezení viditelné vzdálenosti na hodnotu 1, můžeme vidět na obrázku 6.19. Všimněme si zde hlavně modré obálky kolem některých uzlů, která označuje existenci jejich skrytých sousedů. Velikost této obálky je navíc přímo úměrná jejich celkovému počtu a dává tak uživateli dobrou představu jak asi vypadá skrytá část grafu. Samotné snížení viditelnosti je realizováno třídou filtru *DistanceFilter*.



Obrázek 6.19: Omezení zobrazované části rozsáhlého grafu.

Pokud dojde k jakékoliv akci, která vyvolá změnu struktury grafu, je přeskupení jeho uzlů provedeno s animovaným přechodem. V případě, že došlo ke vzniku uzlů nových, je jejich přidání také animováno, a to postupný objevením ze ztracena. Účelem těchto animačních prvků je zejména zabránit zmatení uživatele při rozsáhlých změnách v zobrazovaném obsahu. Samotná doba trvání této animace je nastavena na standardní hodnotu 500 ms, nicméně ta se dá pomocí posledního rozbalovacího seznamu změnit, případně lze efekt animace i zcela vypnout. Zde se ukázal další nedostatek knihovny Zest, neboť ta umožňovala animovat pouze posuv prvků a ne změnu jejich viditelnosti. Z toho důvodu bylo nutné implementovat vlastní animační třídu *GraphAnimator*, nahrazující standardní *LayoutAnimator*.

Posledním ovládacím prvkem, který jsme ještě nezmínili, je pak rozbalovací seznam uprostřed, který umožňuje nastavit velikost plochy, jež vizualizace zabírá.

Poměrně zásadním je pro celou vizualizaci způsob, jakým jsou jednotlivé uzly grafu rozmísťovány, neboť to přispívá k celkové přehlednosti. K tomuto účelu byly použity dva základní algoritmy dostupné v knihovně Zest:

1. *Radial Layout Algorithm* - Umožňuje uspořádat uzly do radiálně orientovaného stromu. Tento algoritmus je vždy aplikován jako první, a to na základní kostru grafu, jež tvoří uzly vlastností.

2. *Spring Layout Algorithm* - Jeho fungování je založeno na simulaci přitažlivých sil mezi jednotlivými vrcholy, díky čemuž umožňuje vytvořit poměrně přirozeně vypadající rozložení grafu. Tento algoritmus je aplikován jako druhý v pořadí, a to na uzly omezení a skupin, včetně jejich sousedů.

6.6.3 Integrace s ostatními komponentami

Ačkoliv dokáže vizualizér zobrazit obsah editoru modelu či konfigurace vlastností, není na implementaci těchto komponent nijak závislý. Způsob, jakým bylo docíleno jejich vzájemné integrace, je rozhraní *IAdaptable*, jež je v rámci platformy Eclipse široce používáno jako jednoduchá alternativa k místům rozšíření (*extension points*).

Rozhraní *IAdaptable* definuje pouze jedinou metodu s názvem *getAdapter*, jejíž parametrem je třída na kterou chceme daný objekt adaptovat. Pokud objekt implementující *IAdaptable* takovou adaptaci umožňuje, vrátí její výsledek jako návratovou hodnotu této metody. Příkladem použití *IAdaptable* v rámci Eclipse je například třída *FileEditorInput* (vstup editoru tvořený souborem), kterou lze pomocí *getAdapter* transformovat na rozhraní *IFile* (soubor).

Obdobně je toto rozhraní použito i ve vizualizéru, který se při detekci změny aktivního editoru pokusí o jeho adaptaci na třídu *FeatureModel*. Pokud byl daným editorem náš editor modelu či konfigurace vlastností, získá vizualizér model, který může zobrazit. Tento přístup je výhodný zejména proto, že vizualizér dokáže model vlastností získat v podstatě od libovolného editoru, který jej skrze *IAdaptable* zpřístupní.

Samotná detekce změny aktivního editoru je zajištěna implementací rozhraní *IPartListener* a zaregistrováním se v rámci *workbench page* jako odběratelem této událostí. Obdobně je detekována i změna výběru z ostatních komponent skrze rozhraní *ISelectionListener*.

7 Výsledný nástroj

Tato kapitola shrnuje výsledky dosažené při implementaci nástroje YAFMT. Na konci kapitoly je pak provedeno jeho zhodnocení a porovnání s ostatními nástroji, zmíněnými v kapitole 3.

7.1 Distribuce nástroje

Výsledný nástroj YAFMT byl zpřístupněn na internetu prostřednictvím hostingové služby *bitbucket*[1], na adrese `https://bitbucket.org/jpikl/yafmt/`. Zde se nachází nejen repozitář zdrojových kódů nástroje, ale také jeho veškerá uživatelská dokumentace či jeho binární distribuce. Samotný zdrojový kód nástroje je licencován pod *Eclipse Public License v1.0*, jak je tomu například standardně u všech projektů Eclipse.

Pro instalaci a běh nástroje je zapotřebí Eclipse (verze 4.2 nebo vyšší) spolu s běhovým prostředím jazyka Java (Java Runtime Environment) verze 6 nebo vyšší. Kromě toho vyžaduje nástroj přítomnost následujících frameworků či knihoven:

- *Eclipse Modeling Framwork* verze 2.8.3 nebo vyšší.
- *Graphical Editing Framework* verze 3.9 nebo vyšší.
- *Zest* verze 1.5 nebo vyšší.
- *Xtext* verze 2.4.1 nebo vyšší.
- *Xtext JFace Integration* verze 2.1 nebo vyšší.

Funkčnost nástroje byla ověřena na operačních systémech Windows 7, Linux a OS X. Hardwarové nároky nástroje jsou víceméně identické s prostředím Eclipse. Minimální množství potřebné operační paměti závisí na počtu ostatních zásuvných modulů, které uživatel používá, nicméně jako doporučenou minimální hodnotu lze uvést 1 GB. Dále je pro běh doporučen procesor alespoň se 1,7 GHz.

Uživatelé si mohou nástroj do prostředí Eclipse nainstalovat buď manuálně (prostým zkopírováním distribuovaných zásuvných modulů) nebo pomocí aktualizačního mechanismu Eclipse, tzv. *update site*. Tento mechanismus spočívá v umístění veškerých zásuvných modulů a jejich meta-dat na server, odkud jsou přístupné pomocí protokolu HTTP. Uživatel pak uvnitř okna správce aktualizací Eclipse zadá adresu dané *update site*[2] a dostupné zásuvné moduly jsou mu nabídnuty ke stažení[3]. Výhodou tohoto přístupu je jeho uživatelská přívětivost a to, že dokáže rozpoznat závislosti instalovaných modulů a případně tyto závislosti doinstalovat také.

[1] `https://bitbucket.org/`

[2] V našem případě `https://bitbucket.org/jpikl/yafmt/downloads/`.

[3] Ve formě tzv. *features*, což je označení skupiny společně distribuovaných zásuvných modulů.

Označení	Komponenta	Popis
Op1	Editor modelu vlastností	Současné smazání všech prvků diagramu.
Op2	Editor konfigurace vlastností	Přidání jedné vlastnosti do konfigurace.
Op3	Vizualizér modelu vlastností	Změna rozvržení prvků (měřena doba výpočtu nového rozvržení).

Tabulka 7.1: Označení a popis měřených operací.

Operace	Počet vlastností/omezení					
	32/16	64/24	128/32	256/48	512/64	1024/96
Op1	146 ms	229 ms	461 ms	1061 ms	3214 ms	11269 ms
Op2	25 ms	48 ms	56 ms	60 ms	76 ms	104 ms
Op3	210 ms	679 ms	2571 ms	9139 ms	35373 ms	140868 ms

Tabulka 7.2: Doba běhu měřených operací.

7.2 Výkonnostní testování

Vzhledem k tomu, že YAFMT není dávkově pracující, ale interaktivní nástroj, bylo obtížné objektivně změřit celkovou výkonnost jeho implementace. V rámci testování byla proto vybrána množina (výpočetně náročných) uživatelských operací a byla programově změřena[4] doba jejich běhu pro různě velké modely vlastností. Zmíněné testovací modely byly automaticky vygenerovány pomocí třídy *FeatureModelGenerator*, nacházející se v modulu *cz.zcu.yafmt.model*. Charakter testovacích modelů víceméně odpovídá těm reálným, neboť je jejich strom vlastností rozsáhlý zejména do šířky[5]. Pro každý model vlastností byla také vygenerována skupina jednoduchých omezení v jazyce BoolCL[6].

Označení jednotlivých testovaných operací a jejich popis je uveden v tabulce 7.1. Pro každou operaci bylo provedeno celkem devět měření, z nichž byl posléze vybrán medián. Výsledky tohoto měření nalezneme v tabulce 7.2. Testování bylo prováděno na sestavě CPU Intel Celeron 1.7 GHz (1 jádro), 2 GB RAM, OS Linux 3.8.7. Před konzultováním samotných výsledků je nejprve nutné dodat, že nástroj není zamýšlen pro tvorbu rozsáhlých modelů o více jak 100 vlastnostech, neboť takto extrémně rozsáhlý diagram se již stává velice nepřehledným a nepraktickým. Druhou polovinu tabulky proto považujme spíše za zátěžový test nástroje.

Odstranění všech prvků diagramu můžeme považovat za jednu z časově nejnáročnějších operací, neboť ovlivňuje každý prvek modelu. Jak vidíme z prvního řádku tabulky, závislost doby jejího trvání na velikosti diagramu má přibližně polynomiální charakter. Příčinou takto dlouhé doby trvání je zejména nutnost zaznamenat původní a budoucí stav každé operace kvůli *undo*/*redo* akcím, což je v tomto případě bohužel výpočetně náročné.

[4] Jako časový rozdíl vrácený mezi dvěma voláními standardní knihovní funkce jazyka Java `System.currentTimeMillis()`.

[5] Na každý uzel stromu připadlo přibližně 1 až 12 potomků.

[6] Každé omezení tvořilo 1 až 5 ID vlastností propojených operátorem `or`.

V případě, že byla daná operace znovu vyvolána pomocí *redo*, byl tento čas již daleko menší[7]. Operace přidání vlastnosti do konfigurace se naproti tomu ukázala být jako velice nenáročná, neboť je doba jejího trvání víceméně závislá pouze na počtu lokálních a globálních omezení modelu, které nutno vždy ověřit. Jako nejkritičtější operací se ukázala být změna rozvržení prvků ve vizualizaci, jejíž doba výpočtu rostla polynomiálně v závislosti na velikosti grafu. Zde se jako úzké hrdlo ukázal být použitý algoritmus výpočtu rozložení *Sprint Layout*, který pracuje iteračně a je tak bohužel více výpočetně náročný než ostatní podobné algoritmy.

7.3 Uživatelské testování

Ve spolupráci s 56 uživateli bylo provedeno testování výsledného nástroje. Uživatelé si při něm měli nástroj sami nainstalovat a vyzkoušet si postupně práci se všemi jeho komponentami (editor modelu i konfigurace vlastností a vizualizér) při tvorbě jednoduchého modelu modelu vlastností dle jejich uvážení (15-20 vlastností).

Uživatelé byli na začátku nejprve krátce seznámeni s problematikou modelování vlastností a byl jim dám jednoduchý návod[8] pro instalaci a základní ovládání nástroje. Cílem bylo hlavně zjistit, jak moc je nástroj pro nového uživatele intuitivní a to, zda uživatele dokáží sami najít a použít některé jeho funkce. Dalším účelem testování bylo pak nalézt případné chyby či nedokonalosti nástroje.

Jednotliví testeři byli převážně pokročilí uživatelé nástroje Eclipse, jen zhruba pětina z nich uvedla, že jsou začátečníci či nemají s Eclipse žádné zkušenosti. Testování bylo zpravidla prováděno na operačním systému Windows 7, několik uživatelů pak testovalo i na Windows 8 či na nějaké Linuxové distribuci. Samotná instalace nástroje (pomocí *update site*) proběhla ve většině případů bez problémů, jen několik uživatelů muselo nástroj nainstalovat ručně, zkopírování zásuvných modulů do instalačního adresáře Eclipse. Případné další problémy byly způsobeny pokusem o instalaci na nepodporovanou verzi Eclipse (starší než 4.2). Jednomu uživateli se nástroj nepovedlo nainstalovat vůbec.

7.3.1 Editor modelu vlastností

Velká část testerů neměla s ovládáním editoru vlastností žádné závažné problémy a hodnotila ho kladně. Více než polovina jich pak označila editor za intuitivní pro neznalého uživatele. Největší problém dělalo testerům ze začátku přijít na to, jak vytvářet skupiny, neboť hledali danou funkcionalitu v paletě nástrojů a nevšimli si její existence v nástrojové liště (až čtvrtina testerů). Po nalezení této funkcionality už pak pro ně nebyl problém ji používat. Většina uživatelů oceňovala možnost automatického rozvržení prvků, některým však chyběla možnost zarovnávat jednotlivé prvky do mřížky[9]. Kromě toho navrhovali testeři nejčastěji následující úpravy a vylepšení:

[7] Například u modelu s 1024 vlastnostmi to byly jen 2 sekundy oproti původním 11 sekundám.

[8] Tento návod je k dispozici na CD přiloženém k této práci. Zmíněné CD pak dále obsahuje i zprávy o testování od jednotlivých uživatelů.

[9] V té době nebyla ještě tato funkcionalita implementována.

- Podpora přibližování/oddalování pomocí kolečka myši a jeho jemnější rozsah.
- Posuv po editační ploše pomocí prostředního tlačítka myši.
- Přidání některých neexistujících klávesových zkratek.
- Citlivější výběr prvků pomocí obdélníkového výběru (aby stačilo označit jen část prvku pro jeho výběr).
- Možnost opakovaného použití nástroje z nástrojové lišty (aby se po jeho použití nenastavil zpět standardní nástroj výběru).
- Možnost zalamování spojení mezi jednotlivými prvky.
- Podpora pro kopírování a vkládání ze schránky.
- Možnost úprav z *outline view* pomocí kontextového menu.

Až na poslední tři se podařilo všechna tato vylepšení dodatečně implementovat. Poměrně velké potíže dělalo některým uživatelům (skoro až třetině z nich) použití integrovaného editoru omezení. Problémem byl zejména nedostatečný popis jazyka BoolCL v dodaném manuálu a také ID vlastností, která jsou automaticky generována v poměrně nevhodném tvaru *f_NáhodnéČíslo*. Uživatelé často nenašli způsob jak tato ID změnit či považovali nutnost jejich ručního zadání za nepraktickou. Proto byla do nástroje dodatečně implementována funkce, která automaticky generuje validní ID vybraných vlastností a atributů na základě jejich názvu.

Dalším problémem byla pak poměrně „agresivní“ validace vstupů, která nedovolila zadat neplatnou hodnotu. Uživatelům se tak často stávalo, že se přepsali a při potvrzení byl jejich zadaný vstup smazán. Toto chování bylo tedy upraveno a chyby vstupu jsou nyní uživateli pouze oznamovány ve stavové liště (při zadávání) či ikonou červeného křížku u příslušného prvku (po zadání).

Kromě výše zmíněného našli uživatelé také několik drobných chyb, jako například špatný výpočet optimální velikosti jednotlivých prvků, neprováděnou validaci některých změn či špatnou aktualizaci některých popisků. Tyto chyby byly opraveny. Kromě toho byly také reportovány problémy zapříčiněné pouze nedostatečným pochopením principu modelování vlastností. Většina uživatelů použila funkci pro exportování diagramu jako obrázku, avšak vadil jim, někdy příliš velký a nepřesný, rozsah vytištěné oblasti. Toto chování již bylo opraveno. Dostupné *outline view* hodnotilo jen malé množství uživatelů, avšak kladně.

7.3.2 Editor konfigurace vlastností

Editor konfigurace byl uživateli hodnocen podstatně lépe než editor modelu vlastností, zejména kvůli jeho jednoduchému ovládání pomocí dvojkliků myší. Více jak polovina uživatelů opět označila editor za velice intuitivní. Testeři zejména oceňovali automatické schovávání neplatných voleb, případně i možnost změny automatického rozvržení prvků (vertikálně a horizontálně orientovaný strom), kde druhé z nich využili zejména při rozsáhlém

modelu vlastností. Někteří uživatelé také ocenili i možnost změny viditelnosti nabízených voleb.

Co však některým testerům chybělo, bylo automatické řešení konfliktů, zejména pro omezení typu vlastnost A vyžaduje vlastnost B. Uživatelé zde předpokládali, že bude B automaticky vybráno poté, co vyberou A. Někteří by také ocenili možnost schovávat volby, jež jsou neplatné dle definovaných globálních omezení. Další věc, která uživatele trápila (až čtvrtinu z nich), bylo promítnutí změn z modelu vlastností do vytvářené konfigurace. Uživatelé očekávali, že budou tyto změny detekovány automaticky, či že bude alespoň dostupné tlačítko pro jejich načtení. Nutnost zavřít a znovu otevřít soubor považovali za nepraktickou. Kromě toho se také dvěma testerům povedlo nalézt v editoru poměrně kritickou chybu (zablokování validních voleb), která je již opravena.

7.3.3 Vizualizér modelu vlastností

Vizualizér představoval z pohledu testerů nejrozporuplnější součást celého nástroje. Ačkoliv někteří hodnotili vizualizér kladně, jiní nechápali jeho použití a považovali ho zcela za zbytečný. Většina testerů se však k užitečnosti této komponenty nijak nevyjádřila.

Co uživatelé často oceňovali, byla animace při změně struktury zobrazovaného grafu, kterou někteří považovali za „efektní“. Dále byly oceňovány dostupné možnosti nastavení vizualizace, ačkoliv několik málo uživatelů nepochopilo jejich význam. Poměrně zásadním problémem pro více jak třetinu testerů bylo nepřehledné rozložení prvků grafu, kdy se (při jeho velkém rozsahu) tyto prvky často překrývaly. Některým uživatelům v takovém případě pomohlo zvětšení vykreslované plochy grafu. Synchronizaci výběru prvků mezi hlavním editorem a vizualizací okomentovalo jen pár uživatelů, avšak kladně. Dále bylo s pomocí testerů nalezeno a opraveno několik drobných chyb.

7.4 Srovnání s ostatními nástroji

Nástroj YAFMT umožňuje, stejně jako většina těch uvedených v kapitole 3, tvorbu modelů vlastností a z nich odvozených konfigurací. Rozdílem oproti některým z nich je možnost specifikace kardinality jak pro skupiny, tak i pro vlastnosti. Tuto schopnost mělo jen několik málo nástrojů (CaptainFeature, FMP, XFeature), jiné pak umožňovaly specifikovat kardinalitu pouze pro skupiny (pure::variants). Dle vzoru některých, i YAFMT umožňuje pro vlastnost specifikovat neomezené množství atributů, avšak pouze několika málo datových typů. Schopnost specializace modelu vlastností jako u FMP podporována není. Na rozdíl od nástrojů typu FeatureIDE a pure::variants, YAFMT nijak nepodporuje propojení vytvořených modelů s ostatními částmi architektury rodiny produktů.

Samotná editace modelu vlastností skrze tvorbu diagramu je podobná přístupu nástrojů CaptainFeaute, XFeature a FeatureIDE, nicméně je oproti nim uživatelsky přístupnější. Zatímco XFeature umožňoval editaci prakticky pouze přes kontextové menu, YAFMT umožňuje používat paletu nástrojů, nástrojovou lištu, klávesové zkratky či *drag and drop*. Víceméně unikátní je i způsob tvorby konfigurace skrze diagram, neboť zde většina ostatních nástrojů použila stromový editor s možností zatrhávání položek. Podobně jako například pure::variants, i YAFMT nabízí pomocné pohledy pro zobrazení editova-

ného modelu, včetně pohledu vizualizace, který je, ve srovnání s tím co nabízely ostatní nástroje, poměrně netradiční.

Tam, kde některé nástroje používaly dávkovou validaci editovaného modelu (po stisknutí příslušného tlačítka), používá YAFMT automatickou revalidaci upravené části modelu a nabízí tak uživateli lepší zpětnou vazbu. Při tvorbě konfigurace jsou pak, stejně jako u ostatních nástrojů, blokovány neplatné volby. YAFMT nicméně nedokáže automaticky řešit konflikty v konfiguraci či zobrazit počet zbývajících validních konfigurací, jako je tomu u FMP či FeatureIDE.

Stejně jako většina dalších nástrojů, i YAFMT umožňuje definovat omezení v textové formě. Zásadním rozdílem i výhodou oproti ostatním je zde však nezávislost na konkrétním jazyce omezení. Pomocí příslušného místa rozšíření lze do YAFMT přidat podporu pro teoreticky libovolný jazyk, například pvSCL z pure::variants.

7.5 Zhodnocení nástroje

V kapitole 4 jsme uvedli celou řadu různých požadavků, jež jsme kladli na vytvářený nástroj. Jak jsme mohli následně vidět v kapitole 6, většinu z těchto požadavků se podařilo úspěšně naplnit a výsledkem je tak multiplatformní nástroj umožňující grafickou editaci i konfiguraci modelů vlastností s kardinalitou a atributy, který je navíc i rozšiřitelný přidáním nových jazyků omezení. Samotný nástroj má však několik nedostatků, které zde, spolu s nerealizovanými požadavky, uvedeme.

Možná největším omezením nástroje je nemožnost dekomponovat model vlastností do více menší modelů, na než by se pak dalo z toho hlavního odkazovat a jež by mohly být i případně uloženy v samostatných souborech. Tento požadavek nebyl realizován z důvodu jeho obtížné implementace, kdy by bylo například nutné kontrolovat to, zda mezi odkazovanými částmi nevzniká cyklus, či nutnost práce s modelem rozkládajícím se do několika souborů. Díky nemožnosti této dekompozice tak není nástroj vhodný pro tvorbu rozsáhlých modelů vlastností (100 a více prvků), neboť je jejich editace v rámci jediného diagramu velice nepřehledná a tím i nepraktická.

Obdobný problém nastává při i vizualizaci rozsáhlého modelu vlastností, kdy je zobrazený graf velice nepřehledný. Zde se však jako poměrně vhodné řešení ukázalo být omezení zobrazovaného rozsahu grafu, které má pozitivní dopad nejen na jeho celkovou přehlednost, ale také i na rychlost odezvy (výpočet rozvržení uzlů grafu). Z tohoto důvodu je tato funkce tedy automaticky aktivována vždy, když je zobrazován graf o 100 a více vrcholech.

Dalším nerealizovaným požadavkem je pak automatické řešení konfliktů při tvorbě konfigurace vlastností, nebo zobrazení počtu zbývajících validních konfigurací. Obě tyto funkce byly vynechány z důvodu obtížnosti a časové náročnosti jejich implementace. Z časových důvodů také nedošlo k realizaci některých jiných požadavků, jako je například import/export modelu vlastností do datového formátu jiných nástrojů. Zde je však nutné poznamenat, že samotný datový formát nástroje YAFMT je postaven na XML a je tedy snadno strojově čitelný a zpracovatelný (viz použití XSLT v ukázkových příkladech).

Jako poslední nedostatek bychom měli zmínit poměrně malou komplexnost dostupného jazyka BoolCL, který nedokáže pracovat s atributy. Neexistuje tak například možnost

jak zajistit, že hodnota číselného atributu nebude záporná, či že daný řetězec bude ve správném formátu.

Na tomto místě se ještě vrat'me k výsledkům výkonnostního testování nástroje, uvedeným v tabulce 7.2. Ačkoliv bychom z některých výsledků mohli nabýt dojmu, že je práce s nástrojem pro rozsáhlé modely vlastností velmi pomalá, není tomu tak zcela úplně pravda. Uvedený příklad, kdy došlo ke smazání všech prvků v diagramu, byl ukázkou prakticky nejnáročnější operace prováděné nad modelem. Doba odezvy obvyklých operací (přidání a ubírání jednotlivých prvků a jejich editace) je i pro rozsáhlé modely vlastností poměrně nízká (viz *Op2* v tabulce 7.2).

Z výsledků uživatelského testování vyplývá, že nástroj je poměrně intuitivní i pro nové uživatele. Většina testerů hodnotila celkový nástroj i se všemi jeho komponentami kladně, pouze několik málo uživatelů bylo frustrováno z problémové instalace a tím i zapřičiněné nefunkčnosti některých součástí nástroje. Ve spolupráci s uživateli se podařilo nalézt celou řadu nedostatků a chyb, které byly následně opraveny.

7.6 Náměty na další rozšíření nástroje

V předchozím textu jsme uvedli několik nedostatků, jejichž náprava by měla být zajištěna v budoucí verzi nástroje. Jako nejdůležitější se zde jeví zejména podpora pro dekompozici modelu vlastností do více menších celků, jež by usnadnila práci v případě rozsáhlého modelu vlastností. Dalším důležitým vylepšením je pak import a export datových formátů jiných nástrojů, zejména komerčního pure::variants. Zásadní je pro nás hlavně možnost importu, která by usnadnila přechod uživatelů od těchto nástrojů.

V rámci editoru konfigurace vlastností by bylo vhodné implementovat podporu pro automatické řešení konfliktů či asistenci uživateli ze strany nástroje při jejich řešení.

Stávající jazyk omezení BoolCL by bylo dobré rozšířit o možnost práce s hodnotami atributů, či případně rovnou implementovat jazyk zcela nový. Zde se jako vhodný kandidát jeví jazyk pvSCL nástroje pure::variants, neboť by jeho podpora usnadnila import z tohoto nástroje.

Jako možné vylepšení vizualizace modelu vlastností bychom mohli uvést použití lepšího algoritmu rozvržení uzlů grafu než je ten stávající[10], který se ukázal být ne příliš vhodný pro rozsáhlé grafy. Možným kandidátem by zde mohl být algoritmus *hypertree*[11].

Kromě výše zmíněného by také bylo možné lokalizovat nástroj i do jiných jazyků než angličtiny. Za zmínku dále stojí prozkoumání možností budoucích verzí frameworku GEF (GEF 4) a knihovny Zest (Zest 2), jež by mohly přinést některá zajímavá vylepšení.

[10] Kombinace algoritmů *radial tree layout* a *spring layout*.
[11] `http://hypertree.sourceforge.net/`

8 Ukázkové příklady

Tato kapitola obsahuje ukázku použití nástroje na dvou praktických příkladech. Z důvodu rozsahu zde bude ke každému příkladu uveden pouze jeho základní popis a motivace, včetně obrázku výsledného modelu uvnitř okna editoru. Pro bližší informace odkážeme čtenáře na již zmíněné webové stránky nástroje, kde jsou tyto příklady, spolu s jejich podrobnějším popisem, dostupné ke stažení. Na těchto stránkách se také ještě nachází třetí (elementární) příklad, který slouží spíše jako úvod pro práci s nástrojem a který zde není pro svou jednoduchost popsán.

8.1 Konfigurace Apache Tomcat

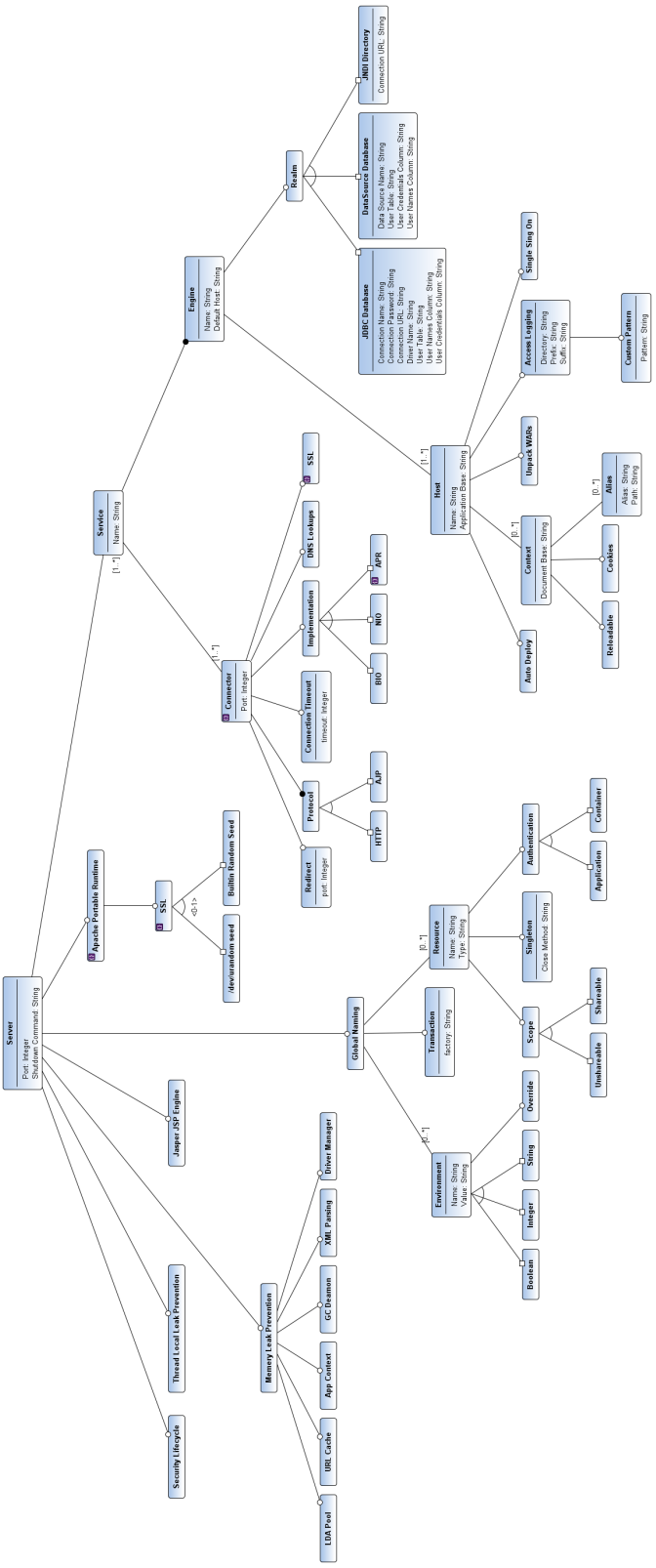
Apache Tomcat[1] je open-source webový server a servletový kontejner, vyvíjený Apache Software Foundation. Konfigurace serveru probíhá vytvořením XML souboru *server.xml* s definovanou strukturou, jež může obsahovat až poměrně velký počet různých parametrů. Cílem tohoto příkladu bylo vytvoření modelu vlastností, jež reprezentuje malou podmnožinu ze všech možných konfiguračních voleb tohoto serveru (konkrétně jeho 7. verze).

Ačkoliv byla vybrána pouze malá část konfiguračních voleb, je výsledný diagram vlastností poměrně rozsáhlý (celkem 58 vlastností). Přibližná struktura zmenšeného diagramu je pro představu uvedena na obrázku 8.1. Samotné vlastnosti uvnitř diagramu odpovídají většinou přímo jednotlivým prvkům konfiguračního souboru, jako je například *Engine* (komponenta zpracovávající požadavky) či *Context* (webová aplikace). Některé vlastnosti mají však poněkud obecnější charakter, jako například podpora *SSL*[2], která zároveň ovlivňuje několik odlišných konfiguračních voleb. Atributy vlastností pak představují různé textové či číselné parametry, jako je například číslo portu. Pro ukázku zde uveďme také dvě omezení v jazyce BoolCL, jež byla nad modelem vlastností definována:

- `conn_apr implies apr` - Pokud je vybrán *Connector* používající APR[3], musí být podpora APR globálně povolena.
- `(exists connector: conn_apr and conn_ssl) implies apr_ssl` - Pokud existuje *Connector* používající zároveň APR i SSL, musí být podpora SSL u APR globálně povolena.

Model vlastností je uložen v souboru *Tomcat7Config.yafm*, ukázková konfigurace vlastností pak v souboru *Tomcat7Config.yafc*. Kromě toho obsahuje příklad také *XSLT* styl *server.xsl*, který dokáže z dané konfigurace vlastností vygenerovat cílový konfigurační soubor *server.xml*, obsahující vybrané volby.

[1] `http://tomcat.apache.org/`
[2] Secure Socket Layer.
[3] Apache Portable Runtime.



Obrázek 8.1: Ukázka modelu vlastností reprezentující část konfiguračních voleb Tomcat 7.

8.2 Sestavení knihovny jQuery UI

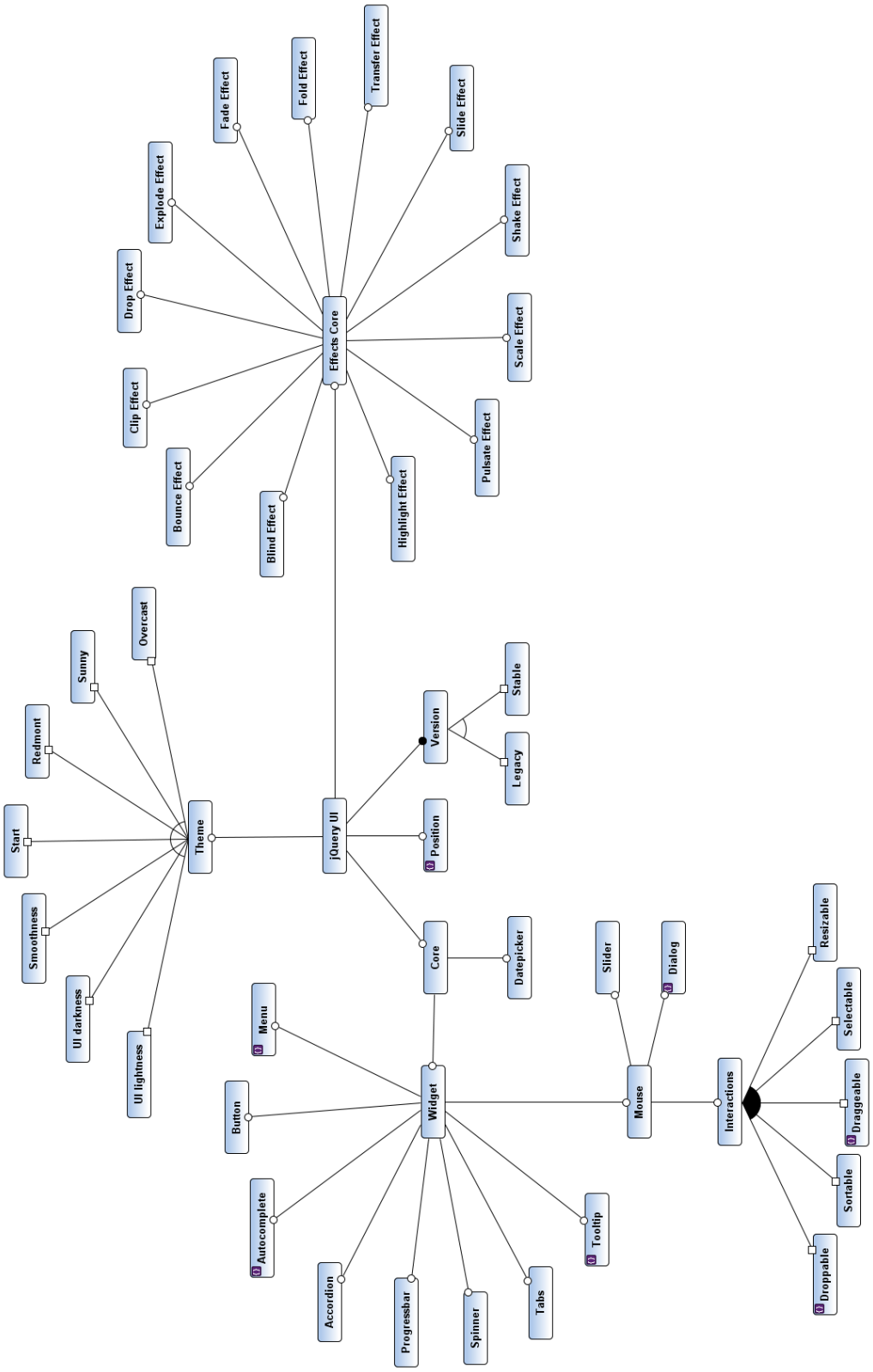
jQuery UI[4] je JavaScriptová knihovna pro tvorbu interaktivních uživatelských rozhraní webových aplikací. Na oficiálních stránkách `http://jqueryui.com/download/` se nachází konfigurátor, pomocí kterého si mohou uživatelé vytvořit vlastní sestavení této knihovny, obsahující jimi vybrané komponenty a téma vzhledu. Zmíněné komponenty pro výběr mají mezi sebou navzájem různé závislosti, které jdou vyjádřit pomocí modelu vlastností, jež je součástí tohoto příkladu.

Navržený diagram vlastností je vidět na obrázku 8.2. Struktura tohoto diagramu většinou odpovídá logickému členění jednotlivých komponent (*widgety*, efekty, témata apod.) a z části také jejich vzájemným závislostem. Závislosti, které nešly vyjádřit pomocí vazeb v diagramu, byly zavedeny pomocí omezení v jazyce BoolCL:

- `droppable implies draggeable` - Interakce upuštěním vyžaduje interakci tažením.
- `autocomplete implies position` - Autokompletace vyžaduje pozicování.
- `dialog implies (draggeable and droppable)` - Dialogové okno vyžaduje interakci tažením a upuštěním.
- `menu implies position` - Menu vyžaduje pozicování.
- `tooltip implies position` - Místní nápověda vyžaduje pozicování.

Model vlastností je uložen v souboru *jQueryUIBuilder.yafm*, ukázková konfigurace vlastností pak v souboru *jQueryUIBuilder.yafc*. Kromě toho obsahuje příklad také *XSLT* styl *request.xsl*, který dokáže z dané konfigurace vlastností vygenerovat tělo HTTP požadavku pro stažení nakonfigurované knihovny ve formě ZIP archivu z adresy `http://download.jqueryui.com/download/`.

[4] `http://jqueryui.com/`



Obrázek 8.2: Ukázka modelu vlastností reprezentující sestavení knihovny jQuery UI.

9 Závěr

V této práci byly diskutovány otázky modelování vlastností a jeho uplatnění v oblasti softwarového inženýrství. Na začátku byl uveden přehled historického i současného vývoje v oblasti modelování vlastností a byly představeny některé dostupné modelovací nástroje z této oblasti. Tyto nástroje byly pak následně porovnány a zhodnoceny z hlediska jejich zaměření, funkcionality, ovládání, uživatelské přívětivosti a použitých technologií.

Na základě požadavků identifikovaných v teoretické části práce byl navržen a implementován nástroj YAFMT, který umožňuje tvorbu a konfiguraci modelů vlastností s kardinalitou, ve formě grafických diagramů. Tento nástroj byl realizován jako sada zásuvných modulů pro platformu Eclipse, s využitém dostupných technologií a knihoven EMF, GEF, Zest a Xtext. S nástrojem je standardně dodávána implementace jazyka BoolCL, který umožňuje nad modelem vlastností definovat různá globální omezení. V případě jeho nedostatečnosti má nástroj přímou podporu pro integraci v podstatě libovolného dalšího jazyka omezení. Důležitou součástí nástroje je pak i komponenta pro vizualizaci modelu vlastností, která má uživateli usnadnit orientaci při tvorbě komplikovaného modelu.

Hlavní výhodou YAFMT oproti jeho konkurentům je jeho uživatelská přívětivost a jednoduchost na použití. Nedostatkem nástroje je pak nemožnost dekompozice vytvářeného modelu do menších částí, takže se příliš nehodí pro tvorbu rozsáhlých modelů vlastností. Nástroj YAFMT je dostupný pro operační systém Windows, GNU/Linux a OS X a je distribuován spolu s otevřeným zdrojovým kódem pod licencí EPL. Funkčnost tohoto nástroje byla ověřena pomocí rozsáhlého uživatelského testování a byla ukázána i na souboru ukázkových příkladů. Výsledný nástroj je se svými zdrojovými kódy, uživatelskou dokumentací i zmíněnými ukázkovými příklady dostupný online a splňuje tedy všechny předpoklady této práce.

Při vývoji nástroje se nepodařilo dokončit některé plánované požadavky, jako je například zmíněná možnost dekompozice modelu vlastností, automatické řešení konfliktů při konfiguraci či podpora pro export a import datových formátů jiných nástrojů. Přidání této funkcionality by tedy bylo dobrým rozšířením nástroje do budoucna. Dalším možným rozšířením je pak i zdokonalení jazyka BoolCL či přidání podpory zcela jiného, komplexnějšího jazyka omezení.

Přínosem výsledného nástroje není jen jeho samotná funkcionalita, ale také to, že může sloužit jako praktická ukázka použití technologií postavených na platformě Eclipse.

Seznam zkratek

AES	Advanced Encryption Standard
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
APR	Apache Portable Runtime
CPU	Central Processing Unit
CSV	Comma Separated Values
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
FMP	Feature Modeling Plug-in
FODA	Feature-Oriented Domain Analysis
FOSD	Feature-Oriented Software Development
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
IDE	Integrated Development Environment
JAR	Java Archive
JDT	Java Development Tooling
MVC	Model-View-Controller
OCL	Object Constraint Language
OSGi	Open Services Gateway initiative
pvSCL	pure::variants Simple Constraint Language
RPC	Rich Client Platform
SDK	Software Development Kit
SPL	Software Product Line
SPLOT	Software Product Line Online Tools
SSL	Secure Socket Layer
SW	Software
SWT	Standard Widget Toolkit
TVL	Text-based Variability Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
XJI	Xtext JFace Integration

XMI XML Metadata Interchange
XML eXtensible Markup Language
XPath XML Path Language
XSD XML Schema Definition
XSLT eXtensible Stylesheet Language Transformations

Literatura

[Ant04] ANTKIEWICZ, M. – CZARNECKI, K. *FeaturePlugin: Feature Modeling Plug-In for Eclipse*. Canada, Waterloo: University of Waterloo, 2004. 6 s.

[Bea06] BEATON, W. – DES RIVIERES, J. *Eclipse Platform Technical Overview* [online]. 2001, poslední revize 19.4.2006 [cit. 2013-03-14]. Dostupné z: `http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html`.

[Bouch10] BOUCHER, Q., et al. *Introducing TVL, a Text-based Feature Modelling Language*. Belgium, Namur: University of Namur, 2010. 4 s.

[Cech04] CECHTICKY, V., et. al. *XML-Based Feature Modelling*. Switzerland, Zürich: Institut für Automatik, ETH-Zentrum, 2004. 15 s.

[Cza98] CZARNECKI, K. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Germany, Ilmenau, 1998. 449 s. PhD thesis, Technische Universit at Ilmenau, Department of Computer Science and Automation. Vedoucí práce Prof. Dr. U. W. Eisenecker.

[Cza00] CZARNECKI, K. – EISENECKER, U. *Generative programming: methods, tools, and applications*. 1 vyd. Boston: Addison-Wesley, 2000. 832 s. ISBN 0-201-30977-7.

[Cza02] CZARNECKI, K., et al. Generative programming for embedded software: An industrial experience report. *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*. Germany, Heidelberg: Springer-Verlag, 2002. s. 156–172 (Vol. 2487 of Lecture Notes in Computer Science).

[Cza04] CZARNECKI, K. – HELSEN, S. – EISENECKER, U. *Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models*. Canada, Waterloo: University of Waterloo, 2004. 28 s.

[Cza05a] CZARNECKI, K. – HELSEN, S. – EISENECKER, U. *Formalizing Cardinality-based Feature Models and their Specialization*. Canada, Waterloo: University of Waterloo, 2005. 25 s.

[Cza05b] CZARNECKI, K. – KIM, Chang H. P. *Cardinality-Based Feature Modeling and Constraints: A Progress Report*. Canada, Waterloo: University of Waterloo, 2005. 9 s.

[Griss98] GRISS, M. L. – FAVARO, J. – D'ALESSANDRO, M. Integrating Feature Modeling with the RSEB. *Software Reuse, 1998. Proceedings. Fifth International Conference on Software Reuse (ICSR)n.* California, Los Alamitos: IEEE Computer Society Press, 1998. s. 76–85.

[Kang90] KANG, K. C., et al. *Feature-Oriented Domain Analysis (FODA): Feasibility Study.* Pennsylvania, Pittsburgh: Carnegie Mellon University, 1990. 161 s.

[Kru13] KRUEGER, W. C. *The Systems and Software Product Line Engineering Lifecycle Framework.* [s.l.]: BigLever Software, 2013. 10 s. Dostupné z: `http://www.biglever.com/learn/whitepapers.html`.

[Lee02] LEE, K. – Kang. K. – LEE, J. *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering.* Korea, Pohang: Pohang University of Science and Technology, 2002. 16 s.

[Mca10] MCAFFER, J – LEMIEUX, J.-M. – ANISZCZYK, C. *Eclipse Rich Client Platform.* 2. vyd. New Jersey, Upper Saddle River: Addison-Wesley, 2010. 553 s. The eclipse series. ISBN 0-321-60378-8.

[Pas05] PASETTI, A. – Rohlík, O. *Technical Note on a Concept for the XFeature Tool.* [s.l.]: P&P Software, 2005, 44 s. Dostupné z: `http://www.pnp-software.com/XFeature/pdf/XFeatureToolConcept.pdf`.

[Pure13] pure-systems GmbH. *pure::variants User's Guide.* [s.l.]: pure-systems, 2013. 166 s. Dostupné z: `http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`.

[Rieb02] RIEBISH, M., et al. *Eetending Feature Diagrams with UML Multiplicities.* Germany, Ilmenau: Ilmenau Technical University, 2002. 7 s.

[Rub11] RUBEL, D. – WREN, J. – CLAYBERG, E. *The Eclipse Graphical Editing Framework (GEF).* 1. vyd. New Jersey, Upper Saddle River: Addison-Wesley, 2011. 312 s. The eclipse series. ISBN 0-321-71838-0.

[She10] SHE, S., et al. *The Variability Model of The Linux Kernel.* Canada, Waterloo: University of Waterloo, 2010. 7 s.

[Scho06] SCHOBBENS, P.-Y. – HEYMANS, P. – TRIGAUX, J.-C. *Feature Diagrams: A Survey and a Formal Semantics.* Belgium, Namur: University of Namur, 2006. 10 s.

[Stein08] STEINBERG, D., et al. *EMF: Eclipse Modeling Framework.* 2. vyd. New Jersey, Upper Saddle River:Addison-Wesley, 2008. 744 s. The eclipse series. ISBN 0-321-33188-5.

[Thum12] THÜM, T., et al. *FeatureIDE: An Extensible Framework for Feature-Oriented Software Development.* Germany, Mandenburg: University of Magdeburg, 2012. 33 s.

A Použitá terminologie

Vzhledem k neexistenci ustálené české terminologie byl autor této práce nucen přeložit některé pojmy z oblasti modelování vlastností. V této příloze je uveden návrh překladu jednotlivých pojmů.

Anglický termín	**Navrhovaný překlad**
Feature model	Model vlastností
Cardinality-based feature model	Model vlastností s kardinalitou
Feature diagram	Diagram vlastností
Feature	Vlastnost
Mandatory feature	Povinná vlastnost
Optional feature	Volitelná vlastnost
Alternative features	Alternativní vlastnosti
Or-features	Slučitelné vlastnosti
Solitary feature	Samostatná vlastnost
Grouped feature	Seskupená vlastnost
Feature group	Skupina vlastností
OR group	OR skupina
XOR group	XOR skupina
Feature model configuration	Konfigurace modelu vlastností
Feature model specialization	Specializace modelu vlastností
Staged configuration	Konfigurace ve fázích
Multi-level configuration	Víceúrovňová konfigurace

B Gramatika jazyka BoolCL

Níže uvedená gramatika je zapsána ve tvaru EBNF. Terminál ID představuje identifikátor vlastnosti, jež může být složen z libovolné kombinace symbolů a-z, A-Z, 0-9 a podtržítka. ID nesmí navíc začínat číslicí.

```
omezeni = kontextualni vyraz;

kontextualni vyraz = [('forall' | 'exists'), ID, ':'], ekvivalence;

ekvivalence = implikace, {'equals', implikace};

implikace = disjunkce, {'implies', disjunkce};

disjunkce = konjunkce, {'or', konjunkce};

konjunkce = negace, {'and', negace};

negace = ['not'], zakladni vyraz;

zakladni vyraz = ID | '(', kontextualni vyraz, ')'
```