Formální specifikace architektur informačních systémů

Marek Rychlý

Ústav informačních systémů,
Fakulta informačních technologií, Vysoké učení technické v Brně,
Božetěchova 2, Brno 612 66, Czech Republic
rychly@fit.vutbr.cz

Abstrakt

Návrh architektury informačního systému tvoří důležitou část vývojového procesu softwarového produktu. Se složitostí architektury stoupá také pravděpodobnost chybného návrhu a obtížnost dokazatelnosti správného chování systému. Formální specifikace architektury umožňuje použít na daný model architektury konkrétního informačního systému formální metody a dokázat tak vlastnosti požadované pro korektní chování systému. Tato práce pojednává o formálních přístupech k implementaci jazyka pro popis architektur (Architecture description language, ADL).

Klíčová slova:
informační systém, architektura, Architecture Description Language, Petriho sítě, temporální logiky, procesní algebry, UML

Obsah

1. Úvod

Softwarová architektura je definována[1] jako „základní organizace softwarového systému zahrnující jeho komponenty, jejich vzájemné vztahy a vztahy s okolím systému, principy návrhu takového systému a jeho vývoje“. Návrh softwarové architektury informačního systému je jedním klíčových kroků vývojového procesu softwarového produktu. Architektura informačního systému leží na vyšší úrovni abstrakce tak, že zahrnuje jak pohled na aplikační doménu (tj. „pohled zákazníka“), tak pohled vývojáře na globální strukturu systému a chování jeho částí, jejich propojení, synchronizaci, rozmístění zdrojů a schéma jejich rezervace, přístup k datům a toky dat v systému, fyzické rozmístění komponent a další.

Z výše uvedeného důvodu mají chyby v návrhu architektury informačního systému velký vliv na celý vývojový proces. Robustní softwarové architektura musí zahrnovat uživatelské i technologické požadavky a vykazovat optimální stupeň srozumitelnosti, udržovatelnosti, rozšiřitelnosti, dopředné i zpětné kompatibility a podporovat vývojový proces. Systém s nevhodně navrženou architekturou sice může pokrývat aktuální specifikaci systému, ale nebude flexibilní a neumožní přizpůsobit systém novým požadavkům (novým funkcím i technologickým změnám). Vzhledem k nutnosti učinit rozhodnutí o architektuře již v počátečních fázích návrhu systému, má chyba při návrhu architektury poměrně drastický dopad na zbytek procesu i na projektové řízení (architektura systému ovlivňuje jeho dekompozici a rozvržení aktivit v rámci projektu).

Tato práce představí různé přístupy k modelování softwarové architektury informačního systému a jejich formální specifikaci. Prezentované přístupy jsou vybrány s ohledem na rozmanitost (z didaktického hlediska) a na uplatnění při návrhu distribuovaných informačních systémů (z praktického hlediska, s respektem k současným trendům v softwarovém inženýrství).

1.1. Architektury informačních systémů

Softwarová architektura představuje především strukturu softwarového systému a pravidla jejího vývoje. Podle síly omezení kladených na vývoj struktury softwarového systému můžeme softwarové architektury rozdělit na tři druhy ([7], také dále):

1. *statická architektura* – umožňuje zachytit pouze pevnou strukturu softwarového systému bez možnosti změn, struktura systému je daná při návrhu a neměnná za běhu systému (např. architektura distribuované aplikace na předem známé konfiguraci sítě),
2. *dynamická architektura* – oproti statické architektuře navíc podporuje vznik a zánik komponent a vazeb za běhu systému podle pravidel určených při návrhu, struktura systému se dynamicky mění (např. architektura distribuované aplikace, která si dynamicky alokuje v síti dostupné zdroje během výpočtu),
3. *mobilní architektura* – rozšiřuje dynamickou architekturu o mobilní prvky, kdy se komponenty a vazby přesouvají za běhu systému podle stavu výpočtu (např. aplikace multiagentní architekturou, kde je změna struktury součástí výpočtu).

[1] dle standardu ANSI/IEEE 1471-2000 „Recommended Practice for Architectural Description of Software-Intensive Systems“

Stav struktury systému se podle softwarové architektury definuje pomocí tří typů entit:

1. *komponenty* – části dekomponovaného systému s daným rozhraním,
2. *konektory* – komunikační kanály pro propojení komponent s daným rozhraním,
3. *konfigurace* – konkrétní způsob vzájemného propojení komponent pomocí konektorů.

U statických architektur existuje pouze jeden stav struktury systému, tzn. především pouze jedna konfigurace. U dynamických a mobilních architektur dochází ke změnám množin komponent a konektorů a ke změnám konfigurace. U dynamických a mobilních architektur je tedy potřeba dále specifikovat, jak se mění konfigurace za běhu systému. Zde definujeme:

1. *akce*, na které systém reaguje (vnější podněty) a které vykonává (vlastní akce),
2. *vztahy* mezi akcemi udávající, na jaké vnější podněty systém reaguje vlastními akcemi,
3. *chování* komponent, konektorů a změny konfigurace, tj. jak se mění softwarová architektura v důsledku akcí.

1.2. Jazyk pro popis architektur (ADL)

Jazyk pro popis architektur (Architecture description language, ADL) je obecný[2] formální prostředek pro vyjádření modelů softwarové architektury na konceptuální úrovni [1]. Jazyk ADL musí být srozumitelný pro všechny strany zúčastněné ve vývoji softwarového produktu (zachycuje technologický i funkční koncept systému), musí podporovat změny architektury během vývojového cyklu, poskytovat prostředky pro snadné vyjádření obvyklých architektur a navazovat na pozdější fáze návrhu a implementace celého systému (např. pomocí softwarových rámců pro podporu architektury). Konkrétní implementace jazyků ADL mívají obvykle prostředky pro grafické znázornění modelu.

Minimální syntax jazyka ADL je dána prvky pro zachycení stavu systému v softwarové architektuře (viz kap. 1.1). Jazyk musí obsahovat prostředky pro vyjádření komponent, konektorů a konfigurace systému a u dynamických a mobilních architektur také prostředky pro akce, vztahy akcí a chování. Z hlediska sémantiky je nezbytné standardizované rozhraní mezi komponentami a konektory, typový systém, omezení konfigurace a přechody, určující možný vývoj konfigurace. Formální sémantika bývá často definována vyjádřením či převodem jazyka ADL na některý z klasických formalismů nebo na jiný „obecnější“ jazyk ADL (jak bude ukázáno dále v této práci).

Vzhledem k striktně formální definici modelů v jazyce ADL jsou výsledné modely automaticky strojově zpracovatelné. Tím může být umožněno generování kódu pro softwarový rámec (např. komunikační vrstva systému v jazyce Java), automatická příprava dokumentů pro následující fáze vývojového procesu (např. diagramy strukturovaných tříd v UML 2.0), překlad modelu na prototyp systému a jeho testování, simulace a verifikace chování architektury systému atd.

[2] konkrétní syntax a sémantika je určena až v konkrétní implementaci jazyka ADL a má úzkou vazbu na použitý formalismus (viz kap. 2)

Jazyky označované jako ADL mohou být rozděleny na tři kategorie, podle toho jak vyjadřují konfiguraci architektury [1]:

1. *jazyky s implicitní konfigurací* – konfigurace je vyjádřena pouze rozhraním komponent a konektorů, kde je jednoznačné jejich propojení (např. rozhraní instance komponenty se odkazuje přímo na konkrétní instanci konektoru),
2. *jazyky s in-line konfigurací* – konfigurace je vyjádřena explicitně (mimo komponenty a konektory), ale v rámci nějakého komunikačního protokolu (vzoru architektury), který určuje účastníky komunikace a tím také konkrétní konfiguraci (např. konfigurace je dána určením konkrétních komponent do rolí účastníků vztahu klient-server)
3. jazyky s *explicitní konfigurací* – konfigurace je vyjádřena odděleně od komponent i konektorů.

Abychom odlišili jazyky ADL od nástrojů, které pouze vykazují prvky ADL tím, že umožňují dekomponovat systém na komunikující části (např. prostředky UML 2.0 zmíněné v kap. 3.2), budeme za ADL považovat pouze jazyky pro popis architektury s explicitní konfigurací (tím zajistíme shodu s definicí architektury podle kap. 1.1).

2. Formalismy pro popis architektur

Formální specifikace softwarové architektury informačního systému umožňuje nad modelem aplikaci formálních metod, jako je dokazatelnost formulí temporálních logik, dokazatelnost či vyvratitelnost některých kritických vlastností (existence uváznutí apod.), simulace v určitém prostředí, různé procházení stavového prostoru atd. Pomocí těchto formálních metod lze validovat softwarovou architekturu a „zaručit se“ za její bezpečnost.

Pro formální specifikaci architektury existuje velké množství jazyků ADL, které využívají různé formalismy. Volba konkrétního ADL je dána především druhem architektury (např. ne všechny formalismy jsou vhodné mobilní architektury), mírou zkušeností návrháře s daným formalismem, ale v neposlední řadě také dostupností nástrojů pro podporu modelování pomocí daného ADL a návazností na ostatní fáze vývojového procesu (převoditelnost na standardizované ADL, možnost generování kódu, implementační podpora apod.).

Tato kapitola postupně představí tři nejrozšířenější formalismy používané pro implementaci jazyků ADL. V každé části kapitoly bude po stručném úvodu do použitého formalismu představen způsob, jak lze implementovat prvky popisující stav struktury softwarového systému a jeho dynamiku (viz kap. 1.1), a uvedeny konkrétní příklady implementací ADL pomocí daného formalismu. V poslední podkapitole budou stručně představeny některé další, méně obvyklé, přístupy.

2.1. Petriho sítě

Petriho sítě (Petri Nets, PN) jsou formální grafický jazyk pro modelování systémů se souběžností a sdílením zdrojů. Tento formalismus představil poprvé *Carl Adam Petri* počátkem 60. let ve své disertační práci. Jazyk je zobecněním teorie automatů o souběžné zpracování. Formálně

je Petriho síť definována jako uspořádaná šestice (S, T, F, M_0, W, K), kde jsou jednotlivé komponenty popsány takto[3]:

- množina S – *místa*,
- množina T – *přechody*,
- množina F – *orientované hrany* mezi místy a přechody, $F \subseteq (S \times T) \cup (T \times S)$,
- funkce M_0: $S \rightarrow N$ – *počáteční ohodnocení* každého místa určitým počtem tokenů,
- funkce W: $F \rightarrow N-\{0\}$ – *váhy hran* přiřazující každé orientované hraně kladné číslo udávající kolik tokenů je v jednom kroku výpočtu zkonzumováno (pokud hrana vede z místa do přechodu) nebo vyprodukováno (pokud hrana vede z přechodu do místa),
- funkce K: $S \rightarrow N-\{0\}$ – *kapacitní omezení* přiřazující každému místu kladné číslo udávající maximální kapacitu místa (maximální počet tokenů, které se mohou v místě nacházet).

Stav Petriho sítě je definován jako vektor $M \in N^n$, kde $n=|S|$ je počet míst sítě, který obsahuje postupně ohodnocení každého místa Petriho sítě (tzn. počet tokenů nacházejících se v daném místě). Uspořádaný seznam dvojic (M_i,t_i), kde M_i je výše definovaný stav Petriho sítě a $t_i \in T$ je přechod, se nazývá *spouštěcí sekvence* (angl. „firing sequence“), pokud splňuje podmínku, že ve stavu M_i je dost vhodně rozmístěných tokenů, aby mohl být uskutečněn přechod t_i. Přechod t_i *může být uskutečněn* ve stavu M_i, pokud pro každou orientovanou hranu $f=(s, t_i)$, kde $s \in S$ je libovolné místo (zde tzv. *vstupní místo*), je ohodnocení místa větší nebo rovno váze hrany, tj. $M_i[s] \geq W(f)$. Uspořádaný seznam stavů, vzniklý projekcí spouštěcí sekvence na první složku, se nazývá *trajektorie*. Následkem *uskutečnění (spuštění) přechodu* t_i ve stavu M_i, pokud může být ve stavu uskutečněn ve výše uvedeném významu, je pro každou orientovanou hranu $f=(t_i, s)$, kde $s \in S$ je libovolné místo (zde tzv. *výstupní místo*), zvýšeno ohodnocení místa s v následujícím stavu M_{i+1} o váhu hrany f, tj. $M_{i+1}[s] = M_i[s] + W(f)$. V jednom kroku výpočtu Petriho sítě může být uskutečněn libovolný (i nulový) počet přechodů.

Při spuštění PN se tedy souběžně přesouvají tokeny mezi vstupními a výstupními místy v celé síti. Protože běh PN je nedeterministický (může být spuštěn libovolný přechod, který má dostatek tokenů na vstupních místech, nehledě na pořadí, přičemž spuštění nemusí nastat v konečném čase) jsou PN vhodným formalismem pro modelování a simulaci distribuovaných systémů.

Petriho sítě bývají použity jako formální základ implementací jazyka ADL, zaměřených především na rozsáhlé distribuované architektury. Přiřazení prvků PN obecnému ADL je na obecné rovině následující [1]:

- *komponenty* ADL jsou rozlišeny na komponenty udržující stav (modelovány pomocí míst PN, kde je stav kódován ohodnocením místa) a komponenty provádějící operaci (modelovány pomocí přechodů PN, jejich uskutečnění mění stavy komponent předchozího druhu),
- *konektory* ADL jsou vyjádřeny orientovanými hranami PN, které spojují komponenty se stavem (místa) s komponentami s operací (přechody),

[3] existují i jiné definice, které např. nemusí obsahovat váhy hran a kapacitní omezení

- *konfigurace* ADL je dána konkrétním výčtem orientovaných hran PN (tedy propojením konektorů a komponent).

Problémem takového návrhu je, že PN umožňuje spojit pouze stavy a přechody, tedy v ADL pouze komponenty se stavem a komponenty s operací, nikoliv např. dvě komponenty s operací (tzn. sekvenční skládání operací). Toto omezení může způsobovat problémy při návrhu architektury, kdy musí být překonáno zavedením pomocných komponent.

Větší problém implementace ADL pomocí Petriho sítí způsobuje absence možnosti specifikovat rozhraní komponent, protože PN nerozlišuje mezi typy tokenů. Toto částečně řeší barvené Petriho sítě, které zavádějí různé druhy tokenů, ale jen v souvislosti s místy (omezující podmínky, že v daném místě smí být uloženy pouze tokeny daných barev) a ne pro přechody v PN (operace v ADL zůstávají tedy nadále beztypové, stejně jako konektory).

Z výše uvedených důvodů lze konstatovat, že formalismus Petriho sítí (bez výrazných úprav) nesplňuje obecné požadavky kladené na jazyk ADL tak, jak byly stanoveny v kapitole 1.2. Proto se pro implementaci ADL používají Petriho sítě v kombinaci s jiným (doplňujícím formalismem), jak bude ukázáno v následující části kapitoly.

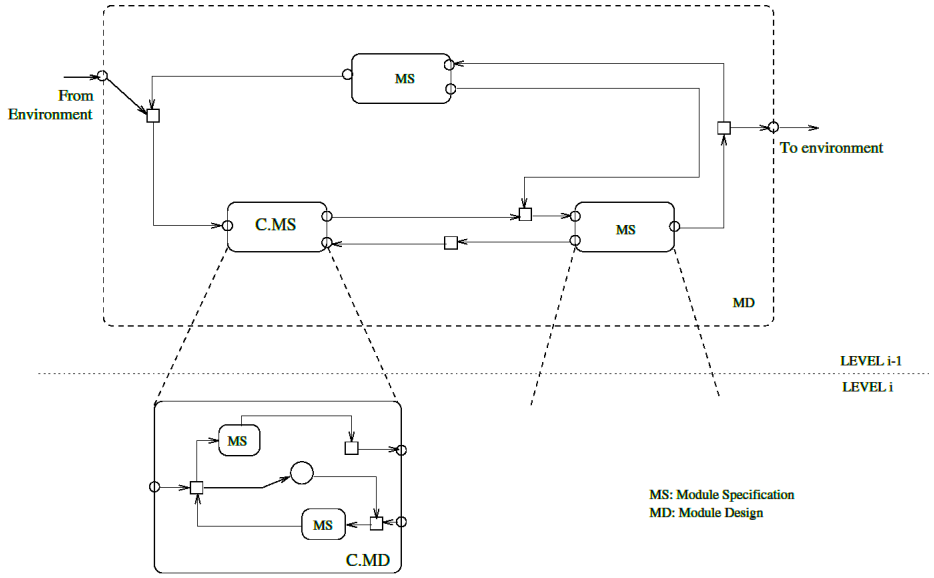
2.1.1. Model NOAM

Jako konkrétní příklad jazyka ADL využívající formalismu Petriho sítí představme *model NOAM* (Net-based and Object-based Architectural Model) publikovaný v [2], přestože v uvedené publikaci není model explicitně pojmenován jako jazyk ADL. Model NOAM umožňuje modelovat real-time systémy pomocí hierarchického objektového přístupu (objekty jsou níže uvedené moduly), do kterého je integrována notace a sémantika Petriho sítí.

Systém je v modelu NOAM vyjádřen jako síť *specifikací modulů* (module specications, MS, ve významu dříve uvedených komponent) propojených *komunikačními kanály* (ve významu dříve uvedených konektorů). Modul MS představuje samotnou komponentu, ale také její chování – popisuje rozhraní komponenty, sémantiku operací, které mění stav této komponenty, jejich závislost a časové pořadí, synchronizaci mezi jejich vstupy a časová omezení jejich běhu. Každý modul MS je možné na nižší úrovni opět dekomponovat na systém dílčích MS, který je ve vztahu k dekomponovanému MS souhrnně označován jako *návrh modulu* (module design, MD) a má jako celek s dekomponovaným MS stejné rozhraní (viz obrázek 1). Každému MS je možné definovat několik MD tak, že zachycují různé alternativy návrhu daného MS.

Grafická reprezentace modulů v NOAM vychází z běžné grafické reprezentace Petriho sítí. Moduly MS jsou graficky značeny jako místa PN a komunikační kanály jako přechody, v PN spojeny s místy orientovanými hranami[4]. Formální sémantika je definována rovněž pomocí Petriho sítí – konkrétně pomocí rozšíření PN o logické podmínky přechodů v podobě *Predicate/Transition-nets* a temporální rozšíření *ER-nets*. Na vyšších úrovních modelu, kdy jsou moduly MD dekomponovány na MS, se jedná o jednoduché mapování mezi modelem NOAM a Petriho sítí, kdy moduly MS jsou vyjádřeny jako místa PN a komunikační kanály jako přechody spojené

[4] existují tedy také kanály, které spojují orientovanými hranami více než jeden modul



Obrázek 1. Dekompozice MS komponenty C dle jejího MD v modelu NOAM (z [2])

s MS orientovanými hranami ve směru komunikace (toto je patrné z obrázku 1). Na nejnižší úrovni dekompozice modelu je modul MD vyjádřen jako plnohodnotná Petriho síť, kde rozhraní MD tvoří operace MS. Celý MD může mít prvky rozhraní (tzv. brány) čtyř druhů, z nichž jsou v [2] uvedeny jen nejdůležitější dva:

- *vstupní brána* (input gate) – pro vstup signálu nebo zprávy vyžadující operaci nebo službu do modulu,
- *odpovědní brána* (reply gate) – pro výstup odpovědi (výsledku operace) z modulu zpět ke klientovi.

Problematika typového omezení rozhraní komponent a konektorů NOAM není v [2] na formální úrovni řešena.

2.2. Temporální logiky

Temporální logika (Temporal Logic, TL) je rozšíření výrokové logiky o modální operátory umožňující pracovat s časem jako se sekvencí stavů. Pravdivost výroků je v temporální logice závislá na čase, resp. na aktuální pozici ve stavovém prostoru a jejím okolí. Kromě klasických čtyř operátorů výrokové logiky $\{\wedge, \vee, \neg, \rightarrow\}$ zahrnují různé varianty TL následující *modální operátory*[5]:

- unární $X\varphi$ – operátor „next“, výrok φ platí v následujícím stavu,
- unární $F\varphi$ – operátor „eventually“, výrok φ platí v některém z následujících stavů (v aktuální posloupnosti stavů),

[5] v některých definicích modální logiky jsou použity pouze základní operátory F a G, přičemž vždy platí rovnost $G\varphi = \neg(F(\neg\varphi))$

- unární Gφ – operátor „globally“, výrok φ platí v každém z následujících stavů (v celé aktuální posloupnosti stavů),
- unární Aφ – operátor „all“, výrok φ platí ve všech posloupnostech stavů, které začínají v aktuálním stavu,
- unární Eφ – operátor „exists“, výrok φ platí v některé z posloupností stavů, která začíná v aktuálním stavu,
- binární φUψ – operátor „until“, výrok ψ platí v tomto nebo budoucím okamžiku a do té doby platí výrok φ,
- binární φRψ – operátor „release“, výrok ψ platí až po první okamžik (včetně), kdy začne platit výrok φ.

Pro zachycení dynamických vlastností systému se používá nejčastěji *logika lineárního času* (Linear temporal logic, LTL) nebo některá její varianta obohacená o prostředky zjednodušující určité konstrukce LTL (např. definici real-time vlastností). Logika LTL umožňuje popsat vlastnosti běhů paralelních programů. *Pravdivost formule* φ logiky LTL pro program P je definována tak, že P splňuje φ právě když každý jeho běh splňuje φ.

Pro *formální definici pravdivosti* φ formule LTL v přechodovém systému T předpokládejme nechť:

- $T=(S, Act, \Rightarrow, s_0)$ je přechodový systém, kde S je množina stavů, Act je konečná množina akcí, $\Rightarrow \subseteq S\times Act\times S$ je přechodová relace, a $s_0 \in S$ je počáteční stav,
- $v: S\rightarrow P_{at}$ je valuace atomických predikátů, která každému stavu přechodového systému přiřazuje množinu v něm platných predikátů,
- α_φ je nekonečné slovo na abecedou formulí LTL tak, že $\alpha_\varphi(i)=At(\varphi)\cap v(\alpha(i))$, kde $\alpha \in T$ je běh, $\alpha_\varphi(i)$ je i-tý symbol slova α_φ, $At(\varphi)$ je množina atomických formulí obsažených ve φ a α(i) je i-tý stav běhu α,

pak LTL formule φ ve stavu $s\in S$ při valuaci v je pravdivá, psáno $s \models^v \varphi$ právě když[6] $\alpha_\varphi \models \varphi$ pro každý běh α taková, že α(0)=s.

Platnost formulí logiky LTL lze automaticky ověřovat pomocí algoritmů, které se opírají o schopnost automatické transformace formule LTL na konečný automat akceptující všechny modely formule φ (tzn. množiny všech formulí platných za předpokladu platnosti formule φ) a o rozhodnutelnost (v polynomiálním čase) problému prázdnosti jazyka akceptovaného takovým automatem. Jedná se o tzv. *Büchiho automat*, což je konečný automat akceptující nekonečná slova. Dokazatelnost platnosti formule LTL nad programem P probíhá tak, že se sestrojí automat daného programu, který se (za předpokladu konečnosti stavového prostoru) transformuje na příslušný Büchiho automat. Pak se řeší problém prázdnosti průniku jazyka akceptovaného uvedeným Büchiho automatem s jazykem všech modelů negace dokazované formule. Pokud je průnik prázdný, je formule pro daný program platná.

Vzhledem k tomu, že temporální logiky specifikují vlastnosti přechodových systémů, lze tyto logiky využít pro implementaci jazyka ADL. Architektura v takové jazyce ADL je popsána

[6] následující výraz je v běžném významu „splnitelnosti“ z výrokové logiky

deklarativním způsobem, kde způsob uplatnění logik TL v popisu závisí na konkrétní implementaci ADL (např. jak bude ukázáno v následující podkapitole). Nad takto definovaným modelem softwarové architektury systému lze provádět automatické důkazy platnosti formulí logiky LTL (jak je popsáno v předchozím odstavci), ale nelze např. simulovat chování systému (v kontrastu s ADL pomocí Petriho sítí z kap. 2.1, které architekturu systému definovaly přímo jako spustitelný automat).

2.2.1. Specifikace dynamického systému založeného na komponentách

Konstrukci konkrétního jazyka ADL pomocí temporálních logik popisuje publikace [3] (dalšími příklady mohou být [4] a [5]). Uvedená implementace jazyka ADL používá logiku LTL (predikátovou logiku prvního řádu obohacenou o modální operátory ve významu LTL). Základní stavební blokem prezentovaného přístupu je *třída*[7], která představuje komponentu s definovanými daty a chováním. Definice třídy je struktura skládající z následujících konečných množin:

- *atributy* – uchovávají informace o vnitřním stavu komponenty (např. počet úloh ve frontě ke zpracování u komponenty představující tiskový server),
- *čtecí proměnné* (read variables) – umožňují komponentě získávat určité informace o svém okolí (např. informace tiskového serveru o připravenosti externí tiskárny),
- *akce* – parametrické operace, které komponenta poskytuje (bez předávání návratových hodnot) a které mohou být zpřístupněny vnějšímu světu pomocí deklarace „exports“ v hlavičce třídy,
- *axiomy* – popisují vliv akcí na stav třídy (tj. na atributy) pomocí aplikace logických operátorů pouze na prvky třídy (není povoleno odkazování mimo třídu).

Všechny atributy, čtecí proměnné a formální parametry akcí mají přiřazen *datový typ* (povoleny jsou jen jednoduché datové typy) a typová kompatibilita je dodržena také v axiomech. Axiomy se mohou rovněž odkazovat na domény datových typů (např. $\forall s \in$ string).

V axiomech třídy se používají akce deklarativním způsobem (viz příklad[8] 1), tj. jako predikáty – takový predikát platí, pokud akce byla vykonána. Existuje zvláštní nulární predikát BEG, který nabývá pravdivé hodnoty tehdy a jen tehdy, když se daná komponenta (třída) inicializuje.

Další prvek prezentovaného jazyka ADL je *vztah* (association) představující konektor ADL. Vztah je opět struktura, jejímž obsahem jsou definice následujících konečných neprázdných množin:

[7] autoři v publikaci vysvětlují odlišnost jejich „třídy“ od běžné definice třídy v objektově orientovaném přístupu

[8] Příklad definuje třídu `Printer` s atributy `ready` a `job`, akcemi `print`, `load` a `print-el` (poslední akce tiskne jeden znak). První axiom říká, že po inicializaci je seznam tiskových úloh `job` prázdný. Axiomy 2 a 3 udávají, že po zavolání akce `load(s)` je úloha s přidána na další místo seznamu. Následující axiom definuje, že akce `print()` smí být vyvolána pouze při neprázdném `job`. Pak je podle dvou axiomů 6 a 7 každá úloha postupně po znacích předána akci `print-el`. Poslední axiomy říkají, že při neprázdném `job` musí někdy dojít k volání `print` a že `ready` je pravda při prázdném `job`.

Class Printer

Exports

print(), load(string), ready

Attributes

- ready : boolean
- job : string

Actions

- print()
- load(string)
- print-el(char)

Axioms

1. BEG → job=[]
2. ∀s∈ string: load(s) → job=[]
3. ∀s∈ string: load(s) → N (job=s)
4. print() → job≠[]
5. ∀j∈ string: print()∧job=j → print-el(head(j))
6. ∀j∈ string: print()∧job=j → N (job=tail(j))
7. job≠[] → F (print())
8. ready=T ↔ job=[]

EndofClass

Příklad 1. Definice třídy Printer (převzato z [3])

- *účastníci* (participants) – definuje formální parametry konektoru, tj. identifikátory, které budou zastupovat v definici konektoru účastníky spojení a jejich typ daný příslušností k třídě,
- *spojení* – pomocí formulí temporální logiky popisuje souvislost atributů a akcí (ve významu predikátů) účastníků spojení.

Konfiguraci částí systému pak zachycují *subsystémy*, které jsou definovány jako struktury následujících konečných množin:

- *počáteční stav* (initial state) – definuje, které instance kterých tříd (tj. instance komponent) jsou v subsystému (konfiguraci) propojeny a pomocí kterých vztahů (konektorů),
- *operace* – popisuje, které operace poskytuje subsystém svému okolí,

- *axiomy* – deklarativním způsobem pomocí temporální logiky popisuje omezení na třídy a vztahy v subsystému a vliv operací subsystému na jeho stav.

Z definice subsystému je patrné, že uvedená implementace jazyka ADL umožňuje hierarchickou dekompozici architektury a dynamickou změnu struktury systému (protože operace subsystému mají vliv na jeho stav, tj. na konfiguraci). Popsaná implementace ADL plně splňuje definice z 1.2.

2.3. Procesní algebry

Procesní algebra (procesní kalkul) je obecný formální model, který poskytuje nástroje pro popis systému souběžných procesů na vysoké úrovni abstrakce. Procesní algebry obsahují dva druhy prvků:

- *proces* (agent) – reprezentuje v systému komunikující entitu, která může být atomická (její obsah není předmětem zájmu) nebo vyjádřená výrazem procesní algebry (dekomponována na subprocesy),
- *jméno* (kanál, port) – prvek, pomocí něhož probíhá komunikace a který je v některých implementacích rovněž předmětem komunikace[9], tj. *zprávou*.

Formálně lze proces v procesní algebře definovat indukcí. V dalším značení bude bez újmy na obecnosti použita notace procesní algebry π-kalkulus a dále bude ukázáno, v čem se π-kalkulus liší o jiné procesní algebry. Nejprve definujme proces `0`, tzv. *prázdný proces* (null process). Pokud jsou P, Q procesy a x, y jména procesní algebry, pak následující konstrukce jsou procesy procesní algebry:

- x<y>.P pošle jméno y pomocí kanálu x a pokračuje jako proces P, tzv. *výstupní operátor*,
- x(y).P – obdrží jméno y po kanálu x a pokračuje jako proces P, tzv. *vstupní operátor*,
- τ.P – vykoná *vnitřní akci* a pokračuje jako proces P,
- (x)P – vytvoří nové jméno x platné v kontextu procesu P a pokračuje jako proces P, tzv. *operátor skrytí* (hiding),
- P|Q – vytvoří paralelní kompozici procesů P a Q, tzv. *operátor paralelní kompozice*,
- P.Q – vytvoří sekvenční kompozici procesů P a Q, kdy proces Q bude proveden po skončení procesu P, tzv. *operátor sekvenční kompozice*,
- !P – vytvoří paralelní kompozici procesu P a operátoru replikace aplikovaného na proces P (tzn. nekonečnou paralelní kompozici s „líným vyhodnocováním“), tzv. *operátor replikace*.

Jako uzávěr, který omezuje platnost jmen na určitý kontext (zapouzdřuje jména pro použití jen uvnitř následující části procesu), funguje vstupní operátor (jméno, do kterého se má uložit přijatá zpráva, nebude volné) a operátor skrytí (prostý uzávěr). Výpočet nad takto definovanou

[9] pro větší srozumitelnost budeme v dalším textu nazývat jména procesní algebry názvy entit, které tato jména představují (tzn. např. použijeme „kanál“ místo „jméno“)

procesní algebrou je realizován pomocí redukce definované na paralelní kompozici komunikujících procesů takto[10]: x<y>.P | x(z).Q → P | Q{z\y}

Při konstrukci jazyka ADL pomocí procesních algeber probíhá následující mapování mezi prvky ADL a prvky procesní algebry:

- *komponenty* ADL jsou v procesní algebře definovány jako procesy,
- *rozhraní komponent* ADL tvoří v procesní algebře všechny volná jména procesu představujícího danou komponentu, přičemž typové omezení rozhraní komponenty je v případě potřeby implementováno prostředky konkrétní procesní algebry (např. je použit typový π-kalkulus),
- *konektory* ADL jsou v procesní algebře vyjádřeny jako speciální procesy, které pouze přeposílají data mezi svými vstupními a výstupními porty (typové omezení rozhraní konektorů je opět implementováno prostředky konkrétní algebry),
- *konfigurace* ADL je definována v procesní algebře jako paralelní kompozice procesů, které představují v ADL komponenty a konektory, s přejmenováním volných jmen (kanálů) procesů tak, aby měly procesy představující komponenty a na ně navázané konektory vždy shodná pojmenování volných jmen.

Pokusme se objasnit převod konfigurace ADL na procesní algebru následujícím příkladem. Mějme dvě komponenty s rozhraním definované jako `A(x : byte, y : string)` a `B(z : byte)`. Dále mějme konektor `C`, který umožňuje propojit komponentu `A` s komponentou `B`. Pak na zvolené úrovni abstrakce představuje komponenta `A` proces P_A s volnými jmény `x`, `y` a komponenta `B` proces P_B s volným jménem `z`. Proces P_C představující konektor `C` má pak tři volná jména, která označme `e`, `f`, `g` (první dvě jsou pro komponentu typu `A` a třetí pro komponentu typu `B`). Konfiguraci systému, kde konektor `C` propojuje komponenty `A` a `B` zapíšeme jako paralelní kompozici procesů P_A, P_B a P_C, kde v procesu P_C přejmenujeme jméno `e` na `x`, `f` na `y` a `g` na `z`. Celou kompozici poté obalíme operátory skrytí aplikovanými po řadě na jména `x`, `y` a `z`.

2.3.1. Jazyk Wright

Příkladem implementace ADL pomocí procesní algebry *Communicating Sequential Processes* (CSP) je jazyk *Wright* [6]. Procesní algebra CSP, kterou zavedl Charles Hoare v roce 1978, obsahuje primitiva *události* (ve významu výše zmíněných jmen), *primitivní procesy* (ve významu atomických základních procesů) a operátory:

- α→P – operátor *prefix*, provede se vstupní nebo výstupní událost α a poté se přejde do procesu P,
- (α→P)□(β→Q) – *deterministický výběr* větvení běhu podle událostí (musí platit α≠β),
- (α→P)○(β→Q) – *nedeterministický výběr* větvení běhu podle událostí (může platit α=β),
- P|||Q – operátor *prokládání*, nezávislá paralelní kompozice,
- P||{α}||Q – operátor *paralelní rozhraní*, paralelní kompozice se sdílenou událostí α,
- (α→P)\{α} – operátor *skrytí* (hiding) události α.

[10] výraz Q{z\y} značí substituci y za z v Q

Jazyk Wright zavádí entity ADL podobným způsobem, jak bylo popsáno na konci kapitoly 2.3. Sémantika takových entit je zachycena pomocí CSP. Konkrétně definuje jazyk *komponentu* (struktura „component“), jako strukturu obsahující:

- definované *rozhraní* (atributy „port“) – vstupní a výstupní rozhraní, která budou odpovídat v CSP událostem,
- *sémantiku* (část ozn. „computation“) – popis procesu v CSP, který komponenta vykonává (s odkazy na definované rozhraní).

Dále zavádí jazyk Wright *konektor* (struktura „connector“, viz příklad[11] 2), který zahrnuje:

- obsluhu rozhraní (atributy „role“) – proces v CSP, který popisuje chování očekávané od komponenty v dané roli (odkazuje se na rozhraní takové komponenty),
- spojovací proces (atribut „glue“) – proces v CSP spojující procesy jednotlivých rolí (odkazuje se na události na procesy definované u rolí).

Konfigurace ADL je vyjádřena strukturou *konfigurace* („configuration“) udávající, které instance komponent a konektorů se účastní konfigurace a jak je propojeno jejich rozhraní. Jazyk Wright umožňuje hierarchickou dekompozici komponent (na vyšší úrovni) na konfigurace (na nižší úrovni), které je implementují.

Pro zobecnění konfigurací zavádí popisovaný jazyk tzv. *architektonické styly*, které představují samostatné obecné komponenty (ne jejich instance) vyjádřené jako vzorové konfigurace spojující dané typy komponent a konektorů (opět ne instance). Styly dále obsahují volitelně integritní omezení zúčastněných komponent a konektorů a povinně rozhraní komponent publikovaná jako rozhraní celé konfigurace.

2.3.2. Jazyk π-ADL

Druhým příkladem modernější implementace ADL pomocí procesní algebry π-kalkulus je *jazyk π-ADL* [7] vyvinutý v rámci projektu *ArchWare European Project*. Procesní algebra π-kalkulus umožňuje zachytit mobilní systémy souběžných procesů a její popis téměř přesně odpovídá modelu procesní algebry definovanému na konci kapitoly 2.3. V π-kalkulu jsou jména v rolích komunikačních kanálů i zasílaných zpráv. To umožňuje předávání komunikačních kanálů mezi jednotlivými procesy, tzv. mobilitu.

Model architektury v jazyce π-ADL se skládá z následujících entit:

- *komponenty* – funkční prvky softwarového systému ve významu komponent podle ADL,
- *konektory* – speciální druh komponent vyhrazený pro komunikaci (pouze vzájemně spojuje své porty do komunikačních kanálů),
- *porty* – rozhraní komponent (včetně konektorů), které umožňuje jejich komunikaci s okolím,

[11] Proces § znamená přerušení. Caller se tedy nedeterministicky rozhoduje mezi voláním a nečinností. Definer se deterministicky čeká na volání nebo končí. Spojovací proces definuje, že když Caller vypustí událost volání (call), tak Definer tuto událost přijme, a opačný směr pro událost konce volání (return).

Connector Procedure-call

Role

- Caller = (out call→in return→Caller) ○ §
- Definer = (in call→out return→Definer) □ §

Glue

(in Caller.call→out Definer.call→Glue) □ (in Definer.return→out Caller.return→Glue) □ §

Příklad 2. Definice konektoru Procedure-call v jazyce Wright (převzato z [6])

- *spojení* – komunikační kanály mezi komponentami (včetně konektorů) navázané na jejich porty.

Platí, že komponenty mohou být navzájem propojeny jen pomocí konektorů a s konektory prostřednictvím spojení. Množina všech spojení v určité části modelu pak tvoří *konfiguraci* této části dle ADL. Komponenty i konektory jsou definovány svým rozhraním (porty) a *chováním*, které udává jejich sémantiku v architektuře. Chování komponent je v jazyce π-ADL zapsáno vlastní syntaxí[12] prostřednictví typového π-kalkulu (viz komponenta v příkladu[13] 3). Tím jsou také definovány typy rozhraní komponent (a konektorů). Shluky propojených komponent a konektorů mohou být zapouzdřeny do větších celků – komponent, kde jsou určité vnitřní porty zpřístupněny vnějšímu prostředí. Tím je umožněna hierarchická dekompozice modelu architektury.

Míra formalismu v návrhu architektury je podle uplatnění typovosti rozdělena do tří úrovní:

1. *základní vrstva* obsahuje jen prvky čistého (netypového) π-kalkulu,
2. *vrstva prvního řádu* rozšiřuje základní vrstvu o datové typy a typové konstruktory (typový π-kalkulus),
3. *vrstva vyššího řádu* umožňuje hierarchické typy a třídy (typy chování a spojení, viz dále).

Kromě prostých typů jsou v π-ADL definovány jako typy také chování a spojení. Ve vrstvě vyššího řádu to umožňuje v modelu manipulovat s komponentami[14] a s konfigurací, jako se zprávami.

[12] místo matematických symbolů je použit zápis ve strukturované angličtině

[13] V komponentě v příkladu je definován vstupní port `in` a výstupní port `toLink`, oba pro složený typ `Entry`. Vstupem i výstupem procesu je hodnota typu `Data`, vstupující jako druhá složka `in` a po zpracování vystupující jako druhá složka `toLink` (pak následuje ostatní chování zastoupené voláním `behaviour()`). V poslední části definice komponenty je formálně popsán protokol, tj. očekávané chování.

[14] zde podobnost s π-kalkulem vyššího řádu, který považuje procesy za jména

```
component SensorDef is abstraction() {
 type Key is Any. type Data is Any. type Entry is tuple[Key, Data].
 port incoming is { connection in is in(Entry) }.
 port outgoing is { connection toLink is out(Entry) }.
 behaviour is {
  process is function(d: Data) : Data { unobservable }.
  via incoming::in receive entry : Entry.
  project entry as key, data.
  via outgoing::toLink send tuple(key, process(data)).
  behaviour()
 }
} assuming {
 protocol is { (
   via incoming::in receive any. true*.
   via outgoing::toLink send any
  )* }
}
```

Příklad 3. Definice komponenty Sensor v jazyce π-ADL (fragment převzat z [7])

Celý model π-ADL jazyka je podpořen nástroji pro vizuální modelování – existuje UML profil, pomocí kterého lze vytvářet π-ADL modely v klasických nástrojích pro UML. Je dostupný také překladač a virtuální stroj (π-ADL je spustitelné), verifikační a upřesňovací nástroje a generátor systémů v jazyce Java.

2.4. Další formální přístupy

Existuje velké množství publikací využívajících jiné formalismy a přístupy k návrhu jazyka ADL. Většinou se však jedná ADL zaměřený na konkrétní oblast architektur (např. real-time systémy) nebo využívající „nestandardní“ formalismy.

Mezi takové přístupy k ADL můžeme zařadit např. model CHAM (Chemical Abstract Machine) založený na abstraktním modelu chemických reakcí, systém *Rapide* (Executable Architecture Definition Language) navržený jako simulační systém pro analýzu běhů, nebo přístupy založené na *jazyku Z*, jazyk *xADL* využívající strukturovaný zápis pomocí XML a jiné.

3. Obecný jazyk

Jazyky ADL představené v předchozích kapitolách této práce se zaměřovaly na omezenou aplikační doménu, která byla nepřímo určena vlastnostmi použitých formalismů. V důsledku těchto omezení prezentovaných ADL, stejně jako kvůli nejednotné syntaxi a sémantice, nebyl žádný z jazyků globálně rozšířen a běžně používán při návrhu informačních systémů.

Obecné jazyky ADL se snaží popsat softwarovou architekturu nezávisle na formálním modelu. Formalismus je tedy ponechán konkrétním „formálním“ ADL. Obecný ADL pak poskytuje pouze modelovací prostředky, případně dále algoritmus obousměrného překladu modelu mezi obecným a konkrétním ADL (toto je však ponecháno v kompetenci autorů konkrétního

ADL). Při vhodně implementovaném obecném ADL je dokonce možno převádět jeho prostřednictvím daný model softwarové architektury mezi různými konkrétními ADL a na model tak uplatnit různé formalismy.

V této kapitole budou stručně představeny dva obecné ADL přistupující k problematice různými způsoby. První bude *jazyk ACME*, který se soustřeďuje na kompatibilitu s konkrétními ADL (v praxi se používá právě jako „převodník“ mezi různými konkrétními ADL). Jak druhý představíme *jazyk UML 2.0*, který poskytuje nástroje pro modelování architektur, co nejsnadnějším a návrháři i uživateli nejsrozumitelnějším způsobem. Nástroje UML 2.0 mají některé vlastnosti jazyku ADL, přestože tak nejsou explicitně pojmenovány.

3.1. Jazyk ACME

Jazyk ACME je ADL jazyk druhé úrovně (tj. obecný[15] ADL jazyk) navržený zobecněním principů použitých v konkrétních jazycích ADL. V následujících odstavcích budou postupně popsány prostředky, které jazyk ACME poskytuje pro vyjádření architektury [9]. Jsou to struktura, vlastnosti, omezení, typy a styly.

Struktura popisuje dekompozici softwarové architektury na jednotlivé části. Entity, které mohou být v takové struktuře uplatněny, jsou následující:

- *komponenty* – reprezentují funkční prvky systému, které mohou vykonávat operace a které mají vnitřní stav,
- *porty* – rozhraní mezi komponentou a jejím okolím, při výstupním portu definované jako emitory událostí a při vstupním portu jako operace komponenty nebo kolekce operací komponenty s daným pořadím provádění aktivované přijatými událostmi,
- *konektory* – prvky pro interakci mezi komponentami,
- *role* – rozhraní mezi konektorem a jeho okolím definující typy komponent a jejich role v konektoru[16],
- *systémy* – konfigurace vyjádřené jako orientované grafy konkrétního propojení komponent pomocí konektorů,
- *reprezentace* – hierarchická dekompozice komponent (na vyšší úrovni) jako systémů s definovaným rozhraním (na nižší úrovni),
- tzv. *rep-maps* (tj. „mapy reprezentací“) – spojující vnitřní porty reprezentace s vnějšími porty reprezentované komponenty.

Vlastnosti umožňují specifikovat upřesňující vlastnosti jednotlivých prvků výše popsané struktury. Každý prvek struktury může mít více vlastností, kde každá vlastnost je dána svým názvem (udává sémantiku, např. „zdrojový kód“), typem (např. „string“) a hodnotou (např. „read(request); write(hash(request+secretKey));“). Vlastnosti mohou být využity v následujících

[15] za jazyky první úrovně můžeme považovat „konkrétní“ formální jazyky ADL, představené v kapitole 2

[16] např. binární konektor s rolemi Caller a Definer (viz 2), nebo n-ární konektor s rolemi EventAnnouncer a EventReceiver

```
Component Type Client = {
 Port sendRequest = {Property protocol: CSPprotocolT};
 Properties {
  requestRate : float;
  sourceCode : externalFile;
 }
 Invariant Forall p in self.Ports @ p.protocol = rpc-client;
 Invariant size(self.Ports) <= 5;
 Invariant requestRate >= 0;
 Heuristic requestRate < 100;
}
```

Příklad 4. Definice typu komponenty Client v jazyce ACME (kompozice příkladů z [9])

fázích procesu vývoje softwarového produktu (např. u výše uvedených příklad to může být fáze generování zdrojového kódu systému).

Omezení vyjadřují, jak se konfigurace softwarové architektury vyvíjí v čase, tzn. umožňují definovat vlastnosti dynamické softwarové architektury. Jsou to omezení kladená na stavy prvků struktury ACME (např. hodnoty vlastností komponent, druhy konektorů apod.) zapsaná v predikátové logice prvního řádu. Jazyk ACME umožňuje definovat predikáty pro některé běžné vlastnosti, jako je např. binární predikát `connected` aplikovatelný na komponenty, který nabývá pravdivé hodnoty, pokud jsou dané komponenty propojeny nějakým konektorem.

Důležitou vlastností jazyka ACME je schopnost definovat architektonické *styly*, což jsou zobecnění konfigurací (s tímto prostředkem jsme se již setkali u jazyka Wright v kap. 2.3.1). S tímto souvisí také *typy* prvků struktury ACME (např. komponenta typu „klient“, konektor typu „roura“ atd.).

Jazyk ACME byl použit v celé řadě případových studií (např. pro specifikaci systému navádění řízených střel pro americké ministerstvo obrany) a stal se de facto standardem pro zápis architektur. Při návrhu nového jazyka ADL je nyní klasicky jednou z fází tvorba překladače do jazyka ACME a zpět, případně stojí ACME přímo na začátku návrhu a jednotlivé prvky struktury ACME a vlastnosti jazyka se postupně implementují v navrhovaném ADL. Proces návrhu v jazyce ACME je podpořen různými vizuálními nástroji (např. *AcmeStudio*) a simulátory.

Na závěr uveďme příklad (viz příklad 4) definice typu komponenty `Client` z běžného architektonického stylu *Client-Server*. V definici je obsažen jeden port nazvaný `sendRequest` s vlastností `protocol` udávající protokol požadavku (na tom může záležet způsob implementace odkazovaný ve vlastnosti `sourceCode`). Mezi vlastnosti samotné komponenty patří `requestRate` udávající četnost dotazů za jednotku času a zmiňovaná `sourceCode`. Následují tři invarianty specifikující nutná omezení komponenty typu `Client` – taková komponenta musí obsahovat nejméně jeden port typu `rpc-client`, nejvýše pět portů, a hodnota vlastnosti `requestRate` musí být nezáporná. Nakonec je tu doporučení, že by hodnota vlastnosti `requestRate` měla být menší než 100 (toto doporučení je zřejmě dáno výkonem předpokládané implementace).

3.2. Jazyk UML 2.0

Unified Modelling Language (UML) je produkt skupiny *Object Management Group* (OMG) pro modelování softwarových produktů. V současné době je UML de facto standardem v průmyslové i akademické sféře. Nová verze UML 2.0 přináší některé změny a nové nástroje usnadňující modelování architektury.

Architektury softwarových systémů mohou být v UML modelovány staticky pomocí diagramu tříd. Dynamiku architektur lze částečně popsat pomocí diagramu spolupráce a sekvenčního diagramu (na úrovni instancí tříd). UML 2.0 zavádí některé typické prvky a principy ADL [8]:

- *strukturované třídy*[17] – popisují vnitřní strukturu třídy pomocí chování a spolupráce částí spojených konektory za účelem hierarchické dekompozice systému a prosazení architektury již ve fázi návrhu (uplatnění strukturovaných tříd se promítne do rozdělení zdrojového kódu, který pak odpovídá architektuře),
- *porty* – popisují rozhraní mezi strukturovanou třídou (komponentou) a okolním systémem ve smyslu portů ADL.

Komponenty ADL jsou v UML 2.0 třídy a strukturované třídy. Rozhraní komponent a konektorů ADL je dáno porty UML 2.0. Problém je nalézt entity UML 2.0, které by odpovídaly konektorům ADL. Zde připadají v úvahu asociace, asociativní třídy a obyčejné třídy, ale ani jedna z nich nesplňuje požadavky ADL. Konektory je tedy nutné definovat na úrovni konkrétního modelu (např. pomocí rámce nebo návrhového vzoru). Konfiguraci ADL lze v UML 2.0 vyjádřit vzájemným napojením komponent pomocí portů (spojení portu typu „provided“ s portem typu „required“) v diagramu spolupráce.

Jiný pohled na architekturu systému v UML nabízejí přístupy *Aspect-Oriented Modeling* (AOM) and *Model-Driven Architecture* (MDA). Architektura MDA zahrnuje „business“ model, výpočetní model a implementační rozhodnutí do různých částí vývojového procesu s definovanými produkty každé fáze a jejich využitím ve fázi následující. Modelování AOM čistě odděluje funkční požadavky (aspekty) od aplikačně závislých (a implementačních) požadavků, které se v pokročilé fázi vývojového procesu kontrolovaně spojí. Oba dva přístupy ovlivňují proces návrhu architektury (problematické jsou zejména otázky „kdy začít navrhovat architekturu“ a „jak dodržet stejnou architekturu v různých modelech“).

Protože „čisté“ UML 2.0 není dostačující pro návrh ADL, je nutné UML rozšířit – toto lze udělat pomocí *profilů*. Profil umožňuje klasifikovat prvky UML pomocí *stereotypů*, které upřesňují jejich sémantiku. Grafické podobě modelu v UML (vizuálně shodné s UML bez profilu) se tak přiřadí jiný význam (nejčastěji bývá takto rozšiřován stavový diagram a diagram tříd). Tímto způsobem je například vytvořen profil pro modelování π-ADL [7] v UML 2.0.

[17] strukturované třídy (diagram složených struktur) vznikly z diagramu komponent v UML 1.x

4. Závěr

V této práci byly stručně představeny různé přístupy k formální specifikaci architektur informačních systémů pomocí různých implementací jazyka ADL, který byl popsán v úvodní části práce.

Modelování architektury informačního systému pomocí ADL zažilo velký rozmach během druhé poloviny devadesátých let 20. století. V té době také vzniklo největší množství různých přístupů k implementaci ADL, které podporovaly především statické a později dynamické architektury (mezi takové přístupy patří také aplikace formalismů představené v kapitolách 2.1 a 2.2). Žádný z projektů se však dostatečně nerozšířil a v navržených ADL byly implementovány většinou pouze případové studie. Možným důvodem malého rozšíření může být neúměrná složitost formalismu při jednoduchosti architektury. Později byly navrženy přístupy umožňující zachycení mobilních architektur (jazyky z kapitoly 2.3), které jsou výrazně složitější než dynamické architektury.

Na základě zkušeností s příliš úzkou aplikační orientací konkrétních jazyků ADL a přílišnou složitostí použitých formalismů byly zavedeny obecné jazyky ADL (představené v kapitole 3). Obecné ADL stanovily de facto standardní části ADL a jejich vlastnosti a umožnily propojit existující formální ADL. Součástí mnoha konkrétních ADL se pak staly převodníky na standardní ADL. Tento trend „standardizace ADL“ se promítl také do jazyka UML, kde zejména ve verzi UML 2.0 jsou patrné některé prvky ADL, přestože nástroje UML 2.0 nepokrývají všechny nezbytné komponenty ADL (viz kapitola 3.2).

Současný trend je integrace jazyků ADL pro mobilní architektury (tedy „nejsilnější“ verze ADL) do standardu UML a jejich podpora pomocí vhodných implementačních nástrojů (generátory kódu, rámce pro programovací jazyky, integrace se spustitelným UML atd.).

Bibliografie

1. Nenad Medvidovic a Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76, Zurich, Switzerland. Springer-Verlag New York, Inc., 1997.
2. Yi Deng, Shengkai Lu a Michael Evangelist. A Formal Approach for Architectural Modeling and Prototyping of Distributed Real-Time Systems. In *HICSS '97: Proceedings of the 30th Hawaii International Conference on System Sciences*, pages 481–491, Los Alamitos, CA, USA. IEEE Computer Society, 1997.
3. Nazareno Aguirre a Tom Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *ASE '02: Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 271–275, Los Alamitos, CA, USA. IEEE Computer Society, 2002.
4. Matteo Pradella, Matteo Rossi, Dino Mandrioli a Alberto Coen-Porisini. A formal approach for designing CORBA based applications. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 188–197, New York, NY, USA. ACM Press, 2000.
5. Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding a Yi Deng. Formally analyzing software architectural specifications using SAM. *Journal of Systems and Software*, 71(1–2):11–29, 2004.

6. Robert Allen a David Garlan. *The Wright Architectural Specification Language*. Technical Report CMUCS-96-TB. School of Computer Science, Carnegie Mellon University. Pittsburgh. URL [http://www.cs.cmu.edu/afs/cs/project/able/ftp/wright-tr.ps]. 1996.
7. Flavio Oquendo. π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.
8. Paris Avgeriou, Nicolas Guelfi a Nenad Medvidovic. Software Architecture Description and UML. In *UML 2004: Modeling Languages and Applications*, Lecture Notes in Computer Science, pages 23–32, Lisbon, Portugal. Springer, 2005.
9. David Garlan, Robert T. Monroe a David Wile. *Acme: Architectural Description of Component-Based Systems*. 47–68. *Foundations of Component-Based Systems*. Cambridge University Press. URL [http://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf]. 2000.