Použití π-kalkulu v UML

Marek Rychlý
<rychly@fit.vutbr.cz>
Ústav informačních systémů,
Vysoké učení technické v Brně

Abstrakt

Unified Modeling Language (UML) je v současné době rozšířeným *de facto* standardem pro specifikaci a návrh softwarových systémů. Přestože UML poskytuje některé prostředky pro formální zápis vlastností modelovaných systémů (Object Constraint Language, OCL), je obtížné definovat systémy vyžadující čistě formální návrh a následnou verifikaci nutných podmínek (vlastností). Jedním z řešení formálního návrhu dynamické části systémů může být procesní kalkul π-kalkul. Tato práce pojednává o formálním převodu modelů systémů ve stavovém diagramu a diagramu aktivit UML do procesů vyjádřených pomocí π-kalkulu.

Klíčová slova:
procesní kalkul, π-kalkul, Unified Modeling Language, stavový diagram, diagram aktivit

Obsah

1. Úvod

1.1. Procesní kalkul

S příchodem pokročilých architektur výpočetních systémů je kladen důraz na nové optimalizované algoritmy využívající paralelismu a souběžnosti. Procesy implementující algoritmy se rozpadají na více paralelních podprocesů, které navzájem spolupracují, komunikují a synchronizují se. Pro modelování a verifikaci takových systémů souběžných procesů jsou nezbytné striktně formální prostředky.

Teoretická informatika poskytuje pro modelování souběžnosti různé nástroje, jako jsou Petriho sítě[1] nebo „Actor model“[2]. Mezi takové nástroje patří také *procesní kalkul*, který využívá pro komunikaci mezi procesy pojmenované kanály. Procesní kalkul obvykle umožňuje paralelní a sekvenční kompozici procesů, definici komunikačních kanálů a jejich omezení (vyhrazení pro určitou skupinu procesů), interakci, rekurzi a replikaci procesů.

Existuje mnoho konkrétních variant implementujících procesní kalkul. Mezi nejvýznamnější patří jistě *Communicating Sequential Processes* (CSP), *Calculus of Communicating Systems* (CCS), *Calculus of Broadcasting Systems* (CBS), a v neposlední řadě *π-kalkul* (*π-calculus for mobile processes*), kterému bude věnována tato práce.

1.2. Unified Modeling Language (UML)

UML je specifikační a modelovací nástroj využívaný zejména v softwarovém inženýrství pro modelování business procesů a (zejména vizuální) specifikaci, návrh a dokumentaci objektově orientovaných softwarových produktů. Jazyk UML vznikl jako neproprietární jazyk kombinující vlastnosti mnoha významných modelovacích jazyků, jako jsou *Booch*, *OMT* nebo *OOSE*.

Nástroje, které jazyk UML poskytuje, můžeme rozdělit na tři části:

a. *funkční model* – zachycuje funkcionalitu systému z vnějšího pohledu (pohled aktérů). Sem patří diagramy případů užití (use-case).
b. *objektový model* – zobrazuje statickou objektovou strukturu systému. UML poskytuje diagramy tříd.
c. *dynamický model* – vyjadřuje dynamické chování systému z vnitřního pohledu. Tento model zahrnuje sekvenční diagramy, diagramy aktivit a stavové diagramy.

Pro porozumění dále prezentovaného použití π-kalkulu v UML bude důležitý dynamický model, a to zejména stavový diagram a diagram aktivit. *Stavový diagram* popisuje deterministické přechody systému nebo jeho části mezi různými stavy, včetně počátečního a koncového stavu. Prvky stavového diagramu jsou:

- stav – Popisuje stav systému v jednom okamžiku. Platí, že část systému se smí nacházet v jednom okamžiku jen v jednom stavu.
- přechod – Přechody jsou definovány jako pětice obsahující událost E_x, podmínku stráže $[guard_x]$, akci $Action_x$, zdrojový stav `S1` a cílový stav `S2`. Pokud se nachází popisovaná část systému ve stavu `S1` a nastane událost E_x a je splněna podmínka stráže $[guard_x]$, přejde do stavu `S2` a zavolá v něm akci $Action_x$.
- vstupní a výstupní akce – Spustí se před vstupem do stavu (vstupní akce), resp. před opuštěním stavu (výstupní akce).
- vnořené stavy – Vyjádření chování systému v daném stavu pomocí dalšího stavového diagramu vnořeného do daného stavu (tzv. „složený stav“). Složený stav může obsahovat více souběžných oddělených stavových diagramů. Při zpracování události mají přednost přechody mezi vnořenými stavy (tzv. „vnitřní přechody“), před přechody vedoucími ze složeného stavu. Při přechodu do složeného stavu přejde systém do iniciálního stavu složeného stavu nebo do pseudostavu historie, pokud takový existuje.

[1] Petriho sítě definoval Carl Adam Petri v roce 1962, jako rozšíření stavových automatů o souběžnost.
[2] „Actor model“ byl publikovaný v roce 1973, jako matematický model pro souběžné výpočty.

- pseudostav historie – Reprezentuje poslední vnořený stav, ve kterém se nacházel složený stav před jeho opuštěním, nebo iniciální stav, pokud historie neexistuje (systém se do složeného stavu dostal poprvé).

Diagram aktivit je varianta stavového diagramu, kde stavy reprezentují operace a přechody aktivity, které nastanou, když jsou operace dokončeny. Diagram aktivit smí obsahovat následující prvky:

- iniciální a finální stav – Jsou to stavy kde systém začíná a končí svou činnost.
- akční stav – Reprezentuje činnost, kterou vykonává systém nacházející se v tomto stavu.
- stav s podaktivitami – Umožňuje vyjádřit akční stav vnořeným diagramem aktivit. Při přechodu do stavu s podaktivitami systém přejde do iniciálního stavu vnořeného diagramu. Po dosažení finálního stavu vnořeného diagramu systém opustí stav s podaktivitami.
- přechod – Přechody jsou většinou definovány jako trojice obsahující podmínku stráže $\texttt{[guard}_x\texttt{]}$, zdrojový stav `S1` a cílový stav `S2`. Přechody probíhají automaticky při splnění podmínky stráže.
- rozhodnutí a sloučení (decision and merge) – Umožňují výlučné větvení toku řízení v diagramu aktivit a jeho opětovné asynchronní sloučení.
- paralelní větvení a synchronizační závora (fork and join) – Umožňují paralelní větvení toku řízení na souběžná vlákna a jejich opětovné synchronní sloučení.
- plavecké dráhy (swimlanes) – umožňují rozdělit diagram aktivit na třídy podle zodpovědnosti za průběh dané části diagramu (tj. v ní obsažených aktivit).

V grafu diagramu aktivit budeme jako „uzly stavů“ označovat akční stavy a stavy s podaktivitami, „uzly pseudostavů“ pak budou rozhodnutí, sloučení, paralelní větvení a synchronizační závora, a pojmem „speciální uzly“ budeme označovat iniciální a finální stavy.

2. Co je π-kalkul?

π-kalkul je procesní kalkul velice blízký *Calculus of Communicating Systems* (CCS). Na rozdíl od CCS používá π-kalkul pouze dva základní pojmy (viz [Milner92] a [Milner93]).

proces (agent)
: Proces v systému reprezentuje komunikující entitu, která může být atomická (její obsah není předmětem zájmu) nebo vyjádřená výrazem π-kalkulu. Celý systém komunikujících procesů lze vyjádřit pomocí vhodných operátorů jako jeden proces.

jméno (kanál, port)
: Jméno je prvek pomocí něhož probíhá komunikace. Jména slouží jako vlastní data (např. proměnné a konstanty) i jako komunikační kanál. V procesu vyššího řádu mohou jména představovat také samotné procesy (nižšího řádu). Pro větší srozumitelnost budeme v dalším textu nazývat jména π-kalkulu názvy entit, které představují (např. kanál).

Ve výrazech π-kalkulu se procesy označují velkými písmeny a jména malými písmeny. Proces π-kalkulu je definován indukcí:

- `0` je proces π-kalkulu (tzv. nulární operátor)
- pokud `P` a `Q` jsou procesy, `x` a `y` jména, pak následující konstrukce jsou také procesy:
  1. $\overline{x}\langle y\rangle.P$
  2. $x(y).P$

3. $\tau.P$
4. $(x)P$
5. $[x = y]P$
6. $P|Q$
7. $P + Q$
8. $A(y_1, \ldots, y_n)$

Ve stručnosti si neformálně vysvětlíme sémantiku výše uvedených konstrukcí:

1. asynchronně pošli `y` po kanálu `x` a pak přejdi na proces `P`,
2. synchronně přijmi data po kanálu `x` do nové proměnné `y`, pak přejdi na proces `P`,
3. přejdi na proces `P` (něco jako je ε-krok v teorii automatů),
4. definuj nové `x` s platností omezenou[3] na proces `P`,
5. pokud je splněna podmínka `x=y`, tak přejdi na proces `P`,
6. souběžně spusť procesy `P` a `Q`,
7. přejdi právě do jednoho z procesů `P` a `Q`,
8. přejdi na procesu `A`, kde substituuj skutečné parametry za formální $\texttt{y}_1$ až $\texttt{y}_n$.

Komunikace procesů (redukce výrazů) probíhá přes společné kanály v daném rozsahu (tj. s respektem k sémantice definované v krocích 2 a 4).

$$(\ldots + \overline{x}\langle y\rangle.P) \mid (\ldots + x(z).Q) \xrightarrow{\tau} P \mid Q\,[y/z]$$

Pro stručnější vyjádření budeme v následujícím textu používat rozšířenou verzi π-kalkulu. Jedná se o tzv. *polyadický π-kalkul* (viz [Milner93]), který umožňuje v jednom kroku po kanálu vyslat více jmen. Rozšíření je bez újmy na obecnosti, neboť procesy polyadického π-kalkulu lze převést na procesy monoadického π-kalkulu, kde je hromadný přenos jmen realizován přenesením soukromého kanálu `l` a následným postupným přenosem jmen v daném pořadí po kanálu `l`.

3. Použití π-kalkulu v UML

V této kapitole budou prezentovány způsoby vyjádření semi-formálních UML diagramů do formálního zápisu v π-kalkulu pomocí procesů a jejich přechodů. Vyjádření systémů pomocí π-kalkulu umožní jejich formální analýzu, verifikaci a důkazy vlastností.

3.1. Vyjádření stavového diagramu pomocí π-kalkulu

Následující poznatky vychází z [Lam01]. Z didaktických důvodů a z důvodu omezeného prostoru budou prezentovány pouze základní poznatky – zájemce o podrobnější popis odkazujeme na výše uvedenou literaturu.

Ještě než přistoupíme k vlastnímu převodu stavového diagramu UML na výrazy π-kalkulu, je třeba upozornit na některé „zvláštnosti“. Při formulaci výrazů čistého π-kalkulu nebudeme používat konstanty (jako [Milner92]). Místo konstant definujeme nové kanály (viz 2), jejichž význam získáme zasláním jinému procesu v určitém pořadí. Například pro testování splnění

[3] Proces `P` může tuto platnost rozšířit tím, že pošle `x` jinému procesu.

podmínky stráže (rozhodnutí, kterým procesem pokračovat) máme kanál `guard` na vykonavatele stráže a postupujeme takto:

$$(tf)(\overline{guard}\langle tf\rangle.(t.P_{true} + f.P_{false}))$$

Každý stav ve stavovém diagramu definujeme jako proces π-kalkulu. Při definici stavu `S1` jako pětice (viz 1.2) můžeme definovat proces `S1` s parametry $\texttt{state}_S$ a $\texttt{event}_S$. Význam parametrů je následující:

- Kanál $\texttt{state}_S$ slouží k dotazu na identifikaci procesu. Proces přijme na tomto kanálu vektor a odpoví na kanál reprezentovaný tolikátou složkou vektoru, kolik je číslo procesu (identifikátor procesu).
- Kanál $\texttt{event}_S$ slouží k příjmu kanálu události. Na takto přijatý kanál vyšle proces vektor a událost pak procesu odpoví na kanál reprezentovaný tolikátou složkou vyslaného vektoru, kolik je číslo události (identifikátor události).

3.1.1. Přechody

Mějme stavy `S1` a `S2` a *přechod* aktivovaný událostí `E1` s podmínkou stráže $[\texttt{guard}_1]$ a akcí $\texttt{Action}_1$, jdoucí ze stavu `S1` do stavu `S2`. Proces `S1` odpovídající příslušnému stavu můžeme s využitím kanálu vykonavatele stráže (jako další parametru procesu) definovat takto:

$$\begin{aligned} S_1(state_S, event_S, guard_1) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(tf)(\overline{guard_1}\langle tf\rangle. \\ & (t.(S_2(state_S, event_S, guard_1)|Action_1)+ \\ & f.S_1(state_S, event_S, guard_1)))+ \\ & \Sigma_{i\neq 1}\, c_i.S_1(state_S, event_S, guard_1)))+ \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S, guard_1) \end{aligned}$$

Výše definovaný proces `S1` nejprve přijme kanál `x` po kanálu událostí $\texttt{event}_S$. Poté vytvoří nový vektor `c`, který vyšle po kanálu `x`. Pokud nastala událost `E1`, tak proces přijme signál po kanálu $\texttt{c}_1$ a vytvoří kanály `t` a `f`, které vyšle na kontrolu stráže po kanálu $\texttt{guard}_1$. Pokud byla kontrola úspěšná tak přijme signál po kanálu `t` a přejde do procesu `S2`, přičemž souběžně spustí proces $\texttt{Action}_1$ (vykonání akce spojené s přechodem). V případě, že byla kontrola stráže neúspěšná (přijme signál po kanálu `f`) nebo nastala jiná událost (přijme signál po kanálu $\texttt{c}_\texttt{i}$, kde $\texttt{i}\neq 1$) proces zůstane nezměněn. Poslední řádek definice procesu slouží k identifikaci procesu po kanálu $\texttt{state}_S$.

Přechod mezi stavy `S1` a `S2` můžeme rozšířit o *parametry události*, které mohou být využity při kontrole stráže, při volání akce spojené s přechodem a při vstupu do dalšího stavu. Následující definice procesu implementuje uvedené rozšíření pomocí kanálu $\texttt{param}_S$ v parametrech procesu.

$$\begin{aligned} S_1(state_S, event_S, param_S, guard_1) = \; & event_S(x).(\overrightarrow{c}) \\ & (\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(l)(\overline{param_S}\langle l\rangle|l(\overrightarrow{plist}).(tf)(\overline{guard_1}\langle tf\overrightarrow{plist}\rangle. \\ & (t.(S_2(state_S, event_S, param_S, guard_1, \overrightarrow{plist})|Action_1(\overrightarrow{plist}))+ \\ & f.S_1(state_S, event_S, param_S, guard_1))))+ \\ & \Sigma_{i\neq 1}\, c_i.S_1(state_S, event_S, param_S, guard_1)))+ \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S, param_S, guard_1) \end{aligned}$$

Při přechodu událostí s parametry vytvoří proces kanál `l` a vyšle ho po kanálu parametrů $\texttt{param}_S$. Na vyslaném kanálu `l` pak přijme vektor `plist` reprezentující seznam hodnot parametrů události. Tento vektor se předává při kontrole stráže, přechodu do dalšího procesu a vykonání akce přechodu.

Definice procesu implementujícího *vnitřní události* je podle [Lam01] téměř shodná s výše uvedenou definicí, kde se nahradí volání procesu `S2` procesem `S1`. Triviální je také úprava definice procesu pro přechody bez událostí, které vyjádříme v π-kalkulu pomocí přechodu τ:

$$\begin{aligned} S_1(state_S, event_S) = \; & \tau.S_2(state_S, event_S) + event_S(x).S_1(state_S, event_S) + \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S) \end{aligned}$$

Pokud ze stavu ve stavovém diagramu vede *více přechodů*, můžeme rozlišit z hlediska provádění přechodů dvě situace:

1. výběr přechodu je uskutečněn událostí (tzn. neexistují dva přechody spouštěné stejnou událostí)
2. výběr přechodu závisí také na splnění podmínky stráže (tzn. ze stejného stavu existuje více přechodů volaných stejnou událostí – provede se přechod, který bude mít splněnu podmínku stráže)

První případ více přechodů lze implementovat pomocí výběru události podle hodnoty vektoru `c` přijatého po kanálu událostí $\texttt{event}_S$. Definice takového procesu `S1`, který při události `Ei` s podmínkou stráže $[\texttt{guard}_i]$ a akcí $\texttt{Action}_i$ přejde do procesu $\texttt{S}_{i+1}$ pro $1 \leq i < n$, je následující[4]:

$$\begin{aligned} S_1(state_S, event_S, \overrightarrow{guard}) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(\Sigma_{i=1}^{n-1} c_i.(t_i f_i)(\overline{guard_i}\langle t_i f_i\rangle. \\ & (t_i.(S_{i+1}(state_S, event_S, \overrightarrow{guard}) | Action_i) + f_i.S_1(state_S, event_S, \overrightarrow{guard}))) + \\ & \Sigma_{i \geq n} c_i.S_1(state_S, event_S, \overrightarrow{guard}))) + \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S, \overrightarrow{guard}) \end{aligned}$$

Druhý případ více přechodů z jednoho stavu musíme řešit odlišným způsobem. Nechť při události `E1` existují přechody ze stavu `S1` do stavů `S2` až `Sn` hlídané podmínkami stráže $\texttt{guard}_1$ až $\texttt{guard}_{n-1}$. V procesu implementujícím stav `S1` při testujeme souběžně podmínky všech stráží dané události (tj. všech přechodů stejné události). Po přijetí výsledku každé podmínky stráže je signalizován kanál `ack` (celkem tedy `n-1` signálů). Pokud je podmínka stráže nepravdivá, tak je očekáván signál na kanálu `stop`. Je-li `i`-tá podmínka pravdivá tak je buď signalizován kanál $\texttt{cont}_i$, nebo je očekáván signál po kanálu `stop`. První možnost (tj. signalizace $\texttt{cont}_i$) smí nastat jen jednou, protože může být vybrán pouze jeden přechod do dalšího stavu (je-li úspěšných více podmínek stráží, tak je tímto uskutečněn nedeterministický výběr jedné). Do dalšího procesu (stavu) $\texttt{S}_{i+1}$ se přejde, pokud skončily všechny kontroly podmínek stráží (bylo obdrženo `n-1` signálů `ack`), `i`-tou podmínkou stráže byl signalizován kanál $\texttt{cont}_i$ a od zbylých podmínek signalizován kanál `stop` (obdrženo `n-2` signálů). Proces je tedy definován takto:

[4] pro zjednodušení nebudeme dále používat události s parametry

$$S_1(state_S, event_S, \overrightarrow{guard}) = event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(\overrightarrow{t}\ \overrightarrow{f}\ \overrightarrow{cont}\ ack\ stop)(\Pi_{i=1}^{n-1}\overline{guard_i}\langle t_i f_i\rangle|$$
$$\Pi_{i=1}^{n-1} t_i.\overline{ack}.(\overline{cont_i} + stop) + f_i.\overline{ack}.stop)|$$
$$ack^{n-1}.(\Sigma_{i=1}^{n-1} cont_i.\overline{stop}^{n-2}.(S_{i+1}(state_S, event_S, \overrightarrow{guard})|Action_i)))+$$
$$\Sigma_{i\neq 1} c_i.S_1(state_S, event_S, \overrightarrow{guard}))+$$
$$state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S, \overrightarrow{guard})$$

3.1.2. Akce a aktivity

Ve stavovém diagramu UML může obsahovat stav vstupní a výstupní akce. Vstupní akce se spustí před vstupem do stavu v němž je obsažena, výstupní akce pak před opuštěním daného stavu.

Mějme stav `S1`, který přejde při události `E1` do stavu `S2` a ten přejde při události `E2` do stavu `S3`. Nechť stav `S2` má vstupní akci $\texttt{Action}_1$ a výstupní akci $\texttt{Action}_2$. Pak v π-kalkulu budeme *vstupní akci* modelovat v procesu `S1`, ze kterého se přechází do procesu `S2` obsahujícího vstupní akci. Přechod do procesu `S2` se dokončí až po signálu vstupní akce na vyhrazeném kanálu `entry`.

$$S_1(state_S, event_S) = event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(entry)(Action_1(entry)|entry.S_2(state_S, event_S))+$$
$$\Sigma_{i\neq 1} c_i.S_1(state_S, event_S))) + state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S)$$

Výstupní akci stavu `S2` definujeme podobně v procesu `S2`, tak aby k ní došlo před přechodem procesu po události `E2` do procesu `S3`.

$$S_2(state_S, event_S) = event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_2.(exit)(Action_2(exit)|exit.S_3(state_S, event_S))+$$
$$\Sigma_{i\neq 2} c_i.S_2(state_S, event_S))) + state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_2}.S_2(state_S, event_S)$$

Aktivita stavu je činnost, která je vykonávána od vstupu do daného stavu až po její ukončení nebo opuštění stavu, a to souběžně s jinou činností realizovanou v rámci stavu (např. souběžně s vnitřními přechody).

Mějme stav `S1`, který přejde při události `E1` do stavu `S2` a ten přejde při události `E2` do stavu `S3`. Stav `S2` vykonává aktivitu $\texttt{Activity}_1$, kterou si vyjádříme jako posloupnost akcí $\texttt{Action}_1$ až $\texttt{Action}_n$. V π-kalkulu budeme modelovat aktivity procesy s parametry `abort` a `complete`. Pokud aktivita obdrží signál po kanálu `abort`, skončí svoji činnost ihned po dokončení právě vykonávané akce (aktivita je přerušena). Pokud je aktivita dokončena bez přerušení, vyšle signál po kanálu `complete`. Proces `S1` pak můžeme definovat takto:

$$S_1(state_S, event_S) = event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(abort\ complete)$$
$$(Activity_1(abort, complete)|S_2(state_S, event_S, abort, complete))+$$
$$\Sigma_{i\neq 1} c_i.S_1(state_S, event_S))) + state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S)$$

Proces aktivity je definován jako navazující běh akcí a možností přerušení provádění.

$$Activity_1(abort, complete) = (\overrightarrow{cont}\ stop)((Action_1(cont_2) + abort.\overline{stop}^n)|$$
$$\Pi_{i=2}^{n} \overline{cont_i}.(Action_i(cont_{i+1}) + abort.\overline{stop}^{(n+1)-i} + stop)|$$
$$(cont_{n+1}.\overline{complete} + stop))$$

Stav `S2`, v rámci kterého běží aktivita, je modelován dvěma procesy `S2` a `S2k`. První proces reprezentuje stav, při kterém běží souběžně aktivita – pokud takový proces `S2` skončí, tak vyšle ukončovací signál běžící aktivitě.

$$\begin{aligned} S_2(state_S, event_S, abort, complete) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_2.\overline{abor}.S_3(state_S, event_S)+ \\ & \Sigma_{i\neq 2}\, c_i.S_2(state_S, event_S, abort, complete)))+ \\ & complete.S_{2k}(state_S, event_S)+ \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_2}.S_2(state_S, event_S, abort, complete) \end{aligned}$$

Pokud skončí dříve proces modelující aktivitu, tak proces `S2` modelující stav přejde do klasického druhého procesu `S2k`, kde není potřeba hlídat žádnou aktivitu.

$$\begin{aligned} S_{2k}(state_S, event_S) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_2.S_3(state_S, event_S) + \Sigma_{i\neq 2}\, c_i.S_{2k}(state_S, event_S)))+ \\ & complete.S_{2k}(state_S, event_S) + state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_2}.S_{2k}(state_S, event_S) \end{aligned}$$

3.1.3. Vnořené stavy

Mějme stav `S1`, který přejde při události $\texttt{E}_1$ do stavu `S2`, přesněji do jeho vnořeného stavu $\texttt{V}_1$. Vnořený stav $\texttt{V}_\texttt{i}$, kde $\texttt{1}\leq\texttt{i}\leq\texttt{n-1}$, přejde do stavu $\texttt{V}_{\texttt{i+1}}$ při události $\texttt{E}_{\texttt{i+1}}$. Stav $\texttt{V}_\texttt{n}$, který je poslední vnořený stav stavu `S2`, přejde do stavu `S3` (tj. opustí stav `S2`) při události $\texttt{V}_{\texttt{n+1}}$.

Proces `S1`, implementující stav `S1`, můžeme vyjádřit v π-kalkulu klasickým způsobem, kde navíc zavedeme kanály stavu a událostí pro procesy vnořených stavů a při přechodu z procesu `S1` do `S2` spustíme souběžně také vnořený proces $\texttt{V}_1$.

$$\begin{aligned} S_1(state_S, event_S) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_1.(state_V\ event_V) \\ & (S_2(state_S, event_S, event_V)|V_1(state_V, event_V, event_S))+ \\ & \Sigma_{i\neq 1}\, c_i.S_1(state_S, event_S))) + state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_1}.S_1(state_S, event_S) \end{aligned}$$

Při přijetí události proces `S2`, který představuje stav `S2`, nejprve předá kanálem $\texttt{event}_\texttt{V}$ událost aktivnímu procesu vnořeného stavu. Pokud vnořený proces událost obslouží, tak toto signalizuje procesu `S2` kanálem `nack`, v opačném případě signalizuje kanál `ack`. Proces `S2` může událost zpracovat jen pokud nebyla obsloužena jeho vnořeným procesem (tzn. obdržel `ack`).

$$\begin{aligned} S_2(state_S, event_S, event_V) = \; & event_S(x).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_{n+1}.(ack\ nack)(\overline{event_V}\langle x\ ack\ nack\rangle| \\ & (ack.S_3(state_S, event_S) + nack.S_2(state_S, event_S, event_V)))+ \\ & \Sigma_{i\neq n+1}\, c_i.(ack\ nack)(\overline{event_V}\langle x\ ack\ nack\rangle| \\ & (ack.S_2(state_S, event_S, event_V) + nack.S_2(state_S, event_S, event_V)))))+ \\ & state_S(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{s_2}.S_2(state_S, event_S, event_V) \end{aligned}$$

Vnořený stav $\texttt{V}_\texttt{i}$, kde $\texttt{1}\leq\texttt{i}\leq\texttt{n-1}$, je realizován proces, který po přijetí události $\texttt{E}_{\texttt{i+1}}$ přejde na proces $\texttt{V}_{\texttt{i+1}}$ a signalizuje `nack`, jinak pouze signalizuje `ack`.

$$\begin{aligned} V_i(state_V, event_V, event_S) = \; & event_V(x\ ack\ nack).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_{i+1}.\overline{nack}.V_{i+1}(state_V, event_V, event_S)+ \\ & \Sigma_{j\neq i+1}\, c_j.\overline{ack}.V_i(state_V, event_V, event_S)))+ \\ & state_V(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{v_i}.V_i(state_V, event_V, event_S) \end{aligned}$$

Poslední vnořený stav `V_n` ve stavu `S2`, který přejde do stavu `S3` (tj. opustí stav `S2`) při události `E_{n+1}`, modelujeme následujícím procesem. Proces při události `E_{n+1}` pouze signalizuje `ack` a skončí – samotný přechod vykoná proces `S2`.[5]

$$V_n(state_V, event_V, event_S) = event_V(x\ ack\ nack).(\overrightarrow{c})(\overline{x}\langle\overrightarrow{c}\rangle.(c_{n+1}.\overline{ack}+ \Sigma_{j\neq n+1}\, c_j.\overline{ack}.V_n(state_V, event_V, event_S)))+ state_V(\overrightarrow{s}, \overrightarrow{v}, \overrightarrow{w}).\overline{v_n}.V_n(state_V, event_V, event_S)$$

3.1.4. Pseudostav historie a souběžné skupiny vnořených stavů

V [Lam01] jsou dále prezentovány způsoby vyjádření stavového diagramu s pseudostavy historie a způsob implementace souběžných skupin vnořených stavů v π-kalkulu. Vzhledem k tomu, že se jedná o rozšíření výše popsaných konstrukcí, nebudeme zde jejich rozsáhlý formální popis uvádět.

Pseudostav historie je realizován pomocným procesem, který umožňuje uložit hodnotu přijatou po kanálu v parametru a na dotaz přijatý jiným kanálem tuto hodnotu odeslat (při vlastních přechodech si proces uloženou hodnotu předává ve svém parametru). Při vstupu do pseudostavu historie se vstupující proces dotáže pomocného procesu na hodnotu historie a podle ní pak spustí proces vnořeného stavu souběžně s cílem přechodu (viz 3.1.3). Při přechodu mezi procesy vnořených stavů a při přechodu mimo proces (tj. ze stavu, ve kterém jsou vnořené stavy) je pomocným procesem uložena hodnota historie (kanál identifikující vnořený stav[6]).

Souběžné skupiny vnořených stavů (souběžné v jednom stavu) jsou modelovány podobně souběžnými procesy, které řízené procesem obsahujícího stavu. Každá skupina pak má vlastní kanál `event`, na kterém přijímá události podobně, jak bylo popsáno v 3.1.3.

3.1.5. Korektnost modelů stavového diagramu v π-kalkulu

Pro *korektnost* je třeba dokázat následující (podrobně viz [Lam01]):

- každý stav je jedinečný (identifikovatelný) – zřejmé z obsluhy kanálu `state`,
- v každé části systému je vždy pouze jeden aktivní stav – pokud je v některém procesu souběžnost, tak je vždy kontrolována a při přechodu do jiného stavu (procesu) je předchozí stav opuštěn (proces ukončen),
- události jsou zpracovávány po jedné – opět zřejmé z realizace jednotlivých typů procesů,
- neimplementovaná událost nemá na systém vliv – zřejmé z předposlední části procesů (ignorování událostí),
- při přechodu je (pouze) opuštěn starý stav a aktivován nový – nikdy nespustíme souběžně více procesů představujících stavy ve stejné části systému,
- při nedeterministické konfliktu je vybrána jen jedna možnost (přechod) – viz synchronizace kritických míst pomocí `ack` kanálů,
- životnost vnořeného stavu nepřesáhne životnost stavu, který ho obsahuje – viz obsluhy kanálů `ack` a `nack`,
- priorita obsluhy události je vyšší u vnořených stavů než u stavu, který je obsahuje – opět viz obsluhy kanálů `ack` a `nack`,

[5] Pokud by možnost opustit stav `S2` měl každý vnitřní proces (tzn. existoval by přechod z `S2`), tak mírně modifikovala předchozí definice procesu o konec signalizací `ack` při akci zmiňovaného přechodu (viz definice `V_n`).

[6] implementující identifikátor vnořeného stavu (protože nemáme konstanty)

- pseudostav historie si pamatuje poslední konfiguraci složeného stavu (z vnořených stavů) – takto je definováno volání pomocného procesu pro zapamatování historie při vnitřním přechodu nebo opuštění složeného stavu.

3.2. Vyjádření diagramu aktivit pomocí π-kalkulu

Podle [Dong03] lze vyjádřit diagram aktivit UML pomocí π-kalkulu následujícím způsobem. Každý stav diagramu aktivit převedeme na proces π-kalkulu s kanály `inevent` a `outevent`, kde `inevent` slouží pro příjem události při vstupu do stavu a `outevent` při opuštění stavu.

Akční stav `S` vykonávající akci `action` bude vyjádřen následujícím procesem:

$$\begin{aligned} \alpha_S &\stackrel{def}{=} (ack).(inevent.(ACTION|ack.\overline{outevent})) \\ ACTION &\stackrel{def}{=} action.\overline{ack}.0 \end{aligned}$$

Zavedeme nový operátor[7] spojující procesy π-kalkulu `P` a `Q` kanálem `m`, který se v procesu `P` připojí na kanál `k` a v procesu `Q` na kanál `l`. Operátor bude definován pomocí substituce:

$$P \frown Q\,(m/k, m/l) \stackrel{def}{=} (m)(P\,[m/k] \mid Q\,[m/l])$$

Nyní můžeme vyjádřit *přechod mezi stavy* v diagramu aktivit (tj. vyjádřit hranu v grafu diagramu aktivit) pomocí výše definovaného operátoru.

$$\alpha_{source} \frown \alpha_{target}\,(m/outevent, m/inevent)$$

Procesy zastupující v π-kalkulu *pseudostavy paralelního větvení a synchronizační závory* (fork and join) stavů obsažených v množině `I` definujeme následovně:

$$\begin{aligned} \alpha_{FORK} &\stackrel{def}{=} inevent.\,\Pi_{i\in I}\,\overline{outevent_i} \\ \alpha_{JOIN} &\stackrel{def}{=} (x)(\Pi_{i\in I}(inevent_i.\overline{x})|x^{|I|}.\overline{outevent}) \end{aligned}$$

V případě implementace *rozhodnutí a sloučení* (decision and merge) v π-kalkulu zavedeme kanál guard, po kterém přijme proces hodnotu použitou pro rozhodnutí. Pokud je `I` množina identifikátorů procesů napojených z jedné strany na rozhodnutí nebo sloučení, definujeme procesy rozhodnutí a sloučení takto:

$$\begin{aligned} \alpha_{DECISION} &\stackrel{def}{=} inevent.guard(c).(\Sigma_{i\in I}[c = c_i]\overline{outevent_i}) \\ \alpha_{MERGE} &\stackrel{def}{=} \Sigma_{i\in I}\,inevent_i.\overline{outevent} \end{aligned}$$

Speciální uzly vyjádříme v π-kalkulu procesy, které manipulují s kanály `inevent` a `outevent`. Proces zastupující iniciální stav vyšle signál po kanálu `begin` a pak odstartuje činnost systému pomocí signálu na kanálu `outevent` (prvnímu nespeciálnímu uzlu). Finální proces diagramu aktivit čeká na signál po kanálu `inevent` (od posledního nespeciálního uzlu) a poté na signál po kanálu `final`, kterým proces celého systému končí.

[7] Použití operátoru je bez újmy na obecnosti řešení, neboť se jedná pouze o zkratku zápisu π-kalkulu dle definice operátoru.

$$\alpha_{initial} \stackrel{def}{=} \overline{begin}.\overline{outevent}.0$$
$$\alpha_{final} \stackrel{def}{=} inevent.final.0$$

Díky výše uvedeným definicím procesů iniciálních a finálních stavů můžeme definovat proces pro *stav s podaktivitami*. Proces reprezentující systém podaktivit (tj. diagram aktivit obsažený ve stavu s podaktivitami) napojíme přes kanály `begin` a `final` na kanály `inevent` a `outevent` v okolí procesu stavu s podaktivitami.

Nyní můžeme *celý systém modelovaný diagramem aktivit* vyjádřit v π-kalkulu jediným procesem. Tento proces vznikne paralelní kompozicí všech procesů uzlů stavů (množina `AS`), pseudostavů (množina `PS`), iniciálních a finálních stavů, přičemž kanály `l` použité pro vnitřní komunikaci zamaskujeme (typicky varianty kanálů `inevent` a `outevent`).

$$\alpha \stackrel{def}{=} (l)(\prod_{as \in AS} \alpha_{as} \mid \prod_{ps \in PS} \alpha_{ps} \mid \alpha_{initial} \mid \alpha_{final})$$

Závěrem je třeba upozornit, že [Dong03] neprezentuje převod všech prvků diagramů aktivit do výrazů π-kalkulu (např. nejsou zahrnuty toky objektů).

4. Důkazy nad π-kalkulem pro UML

4.1. Využití vlastností π-kalkulu

Procesní kalkul π-kalkul má mnoho zajímavých vlastností [Milner92]. Díky vyjádření stavového diagramu či diagramu aktivit v π-kalkulu můžeme využít tvrzení vyplývající z těchto vlastností π-kalkulu pro systémy modelované v UML.

Analogicky s CCS zavádí π-kalkul *relaci slabé bisimulace*[8], jako ekvivalenci dvou procesů `P≈Q` tak, že platí zároveň:

- pokud `P` přejde pod akcí `α` do `P'`, tak existuje `Q'` takové, že `Q` přejde pod akcí `α` do `Q'` a `P'≈Q'`,
- pokud `Q` přejde pod akcí `α` do `Q'`, tak existuje `P'` takové, že `P` přejde pod akcí `α` do `P'` a `P'≈Q'`.

Jinými slovy relace slabé bisimulace říká, že procesy v relaci mají shodné chování. Pokud tuto vlastnost přeneseme na UML diagramy dynamického chování systému, tak máme prostředek, jak zjistit *funkční ekvivalenci chování daných systémů*[9].

4.2. Formální verifikace

Procesy zapsané pomocí π-kalkulu je možné automaticky formálně verifikovat. Například nástroj *The Mobility Workbench* (MWB) umožňuje manipulovat a analyzovat systémy popsané pomocí π-kalkulu [Bjorn94].

Obecně je možné dokázat nad procesy π-kalkulu platnost logických formulí (především temporálních logik[10]). Takto lze vyjádřit a dokázat některé klíčové vlastnosti modelovaných systémů, např. že systém může vždy normálně skončit (dosáhnout konečného stavu) nebo že

[8] Slabá bisimulace narozdíl od silé bisimulace umožňuje procesům v relaci nezávisle přecházet pod τ-akcemi.
[9] využitelné např. v důkazu, že konkrétní návrh chování systému je funkčný shodná s referenčním návrhem
[10] konkrétně MWB používá k vyjádření formulí mu-kalkul

neexistují deadlocky (tj. systém nemůže přejít do stavu, kde nereaguje na žádné akce – tedy na proces bisimulačně ekvivalentní procesu `0`).

5. Závěr

Tato práce stručně shrnuje poznatky z oblasti formálního vyjádření dynamických digramů UML pomocí procesního π-kalkulu. Vyjádření systémů modelovaných ve stavových diagramech a diagramech aktivit a převod do procesů π-kalkulu umožňuje formální analýzu a verifikaci pomocí automatických nástrojů. Mezi hlavní přínosy vyjádření pomocí π-kalkulu patří především možnost formálního důkazu ekvivalence procesů a důkazů temporální logiky.

Předmětem dalšího výzkumu může být např. rozšíření konverze do π-kalkulu na všechny druhy dynamických diagramů UML, zejména na celý diagram aktivit ([Dong03] nezahrnuje některé obtížnější části diagramů aktivit). Výzvou může být také uplatnění π-kalkulu pro samostatné modelování business procesů (a verifikaci business logiky systémů[11]).

Bibliografie

[1] Robin Milner, Joachim Parrow a David J. Walker. „A calculus of Mobile Processes (parts I & II)“. 1–40 & 41–77. *Information and Computation*. 100(1). Elsevier. 1992. 0890-5401.

[2] Robin Milner. „The polyadic π-calculus: a tutorial“. 203–246. *Logic and Algebra of Specification*. F. L. Bauer. W. Brauer. H. Schwichtenberg. Springer-Verlag. 1993.

[3] Vitus S. W. Lam a Julian A. Padget. „Formalization of UML Statechart Diagrams in the π-calculus“. 213–223. *Australian Software Engineering Conference*. IEEE Computer Society. 2001. 0-7695-1254-2.

[4] Yang Dong a Zhang Sheng. „Using π-calculus to Formalize UML Activity Diagram“. 47–54. *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003)*. IEEE Computer Society. 2003. 0-7695-1917-2.

[5] Victor Björn a Faron Moller. „The Mobility Workbench — A Tool for the π-Calculus“. 428–440. *CAV'94: Computer Aided Verification*. Springer-Verlag. 1994.

[11] UML se zaměřuje především na aplikační logiku