VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů

Ing. Bohuslav Křena

METODY ANALÝZY OBJEKTOVĚ ORIENTOVANÝCH PETRIHO SÍTÍ

ANALYSIS METHODS OF OBJECT ORIENTED PETRI NETS

Zkrácená verze Ph.D. Thesis

Obor:	Informační technologie
Školitel:	Prof. RNDr. Milan Češka, CSc.
Oponenti:	Prof. Ing. Štefan Hudák, DrSc. Doc. Ing. Jaroslav Sklenář, CSc.
Datum obhajoby:	20. říjen 2004

Klíčová slova

objektově orientované Petriho sítě, *PNtalk*, formální metody, stavové prostory, problém stavové exploze, paralelizace, architektury se sdílenou pamětí, Java, *JOMP*

Key Words

Object-Oriented Petri Nets, *PNtalk*, Formal Methods, State Spaces, State Space Explosion Problem, Parallel Approach, Shared Memory Architecture, Java, *JOMP*

Místo uložení

Vědecké oddělení, FIT VUT v Brně, Božetěchova 2, 612 66 Brno

ISBN 80-214-2797-3
ISSN 1213-4198

Obsah

1 Současný stav řešené problematiky

Systémy založené na počítačích jsou nyní používány téměř ve všech oblastech lidské činnosti. Nesprávná funkce těchto systémů může mít za následek velké finanční ztráty a při použití v kritických aplikacích může dojít k poškození zdraví, ke ztrátám lidských životů nebo k vážnému poškození životního prostředí. Při vývoji počítačových systémů je tedy více než žádoucí, aby tyto systémy obsahovaly minimum chyb. Zásadní překážkou pro tvorbu *dokonalých* počítačových systémů je jejich složitost.

V praxi se využívá řada metod pro zlepšení kvality vyvíjených počítačových systémů, přičemž jednotlivé metody se používají s různou úspěšností v různých fázích vývoje systémů. Některé metody se snaží zlepšit analýzu a návrh systémů, řada technik se používá pro minimalizaci vzniku chyb při implementaci software a existuje též nepřeberné množství metod a technik pro testování výsledného produktu. Kromě toho existují i metody, které ovlivňují všechny fáze vývoje systému — jedná se například o metody pro řízení vývoje systémů, používání CASE nástrojů či využití formálních technik. A právě formálním technikám je tato práce věnována.

1.1 Možnosti odhalování chyb v systémech

Zřejmě nejstarší a nejrozšířenější metodou pro odhalování chyb v systémech je *testování*, které spočívá v experimentování se samotným systémem nebo jeho částí. Nevýhodou testování je, že u reálných systémů nelze vyzkoušet reakce systému na všechny možné situace, protože jich je příliš mnoho. Další často používanou metodou je *simulace*, která spočívá v experimentování se simulačním modelem systému. Poslední významnou skupinou metod pro odhalování chyb v systémech je *formální analýza a verifikace*. Tyto metody mají na rozdíl od testování a simulace potenciál pro dokazování správnosti systémů.

Vstupem formální analýzy a verifikace je model systému, který může být popsaný např. pomocí automatů, Petriho sítí, procesních algeber nebo univerzálních programovacích jazyků. Dalším vstupem formální verifikace je specifikace vlastnosti, která má být pro daný systém ověřena. Vlastnosti jsou nejčastěji vyjádřeny pomocí některé temporální logiky, přičemž mezi nejpoužívanější patří ***L**inear-time **T**emporal **L**ogic* (LTL), ***C**omputation **T**ree **L**ogic* (CTL) a CTL*, která spojuje vyjadřovací možnosti LTL a CTL[6]. Podle způsobu zjišťování, zda model danou vlastnost splňuje, můžeme metody formální analýzy a verifikace rozdělit do tří tříd — strukturální metody, deduktivní metody (*theorem proving*) a metody založené na procházení stavových prostorů (*model checking*).

Hlavním problémem prakticky všech metod formální analýzy a verifikace je

složitost. U metod založených na generování a procházení stavového prostoru modelů, které jsou předmětem našeho zájmu, se složitost projevuje problémem stavové exploze — velikost stavového prostoru roste exponenciálně s velikostí modelu [20]. Pro omezení tohoto problému se používají dvě základní skupiny technik. Do první skupiny řadíme metody, které využívají úspornější reprezentaci stavových prostorů nebo které části stavového prostoru vůbec negenerují. Metody z druhé skupiny pro generování a procházení stavových prostorů zase využívají výkonnější počítače (s distribuovanou nebo sdílenou pamětí). Metody z obou skupin lze s výhodou kombinovat.

1.2 Související výzkum na FIT

Na Fakultě informačních technologií (FIT) VUT v Brně je výzkumu metod pro vývoj počítačových systémů věnována dlouhodobá pozornost. Základní metody modelování a simulace diskrétních i spojitých systémů jsou rozšiřovány za účelem modelování a simulace složitých, heterogenních, paralelních i objektově orientovaných systémů [18, 7].

Pro modelování, simulaci, formální analýzu a verifikaci paralelních systémů jsou hojně používány Petriho sítě [19], které poskytují grafickou reprezentaci a přesný matematický popis systémů. Petriho sítím se na FIT věnuje výzkumná skupina Petriho sítí, jejíž dřívější úspěšný výzkum černobílých Petriho sítí vyvrcholil návrhem a implementací nástroje PESIM [5]. V poslední dekádě byl výzkum zaměřen na vysokoúrovňové Petriho sítě, zejména pak na rozšíření Petriho sítí o koncepty objektové orientace. Výsledkem výzkumu byl návrh formalizmu objektově orientovaných Petriho sítí[1] a s ním spojeného jazyka a nástroje *PNtalk* [8, 21, 4].

Na modelování a simulaci *OOPN* modelů logicky navazuje jejich formální analýza a verifikace [17, 22, 3], v jejímž rámci jsou zkoumány metody formální analýzy a verifikace zejména softwarových systémů založené na generování a procházení konečných stavových prostorů a nově i na speciální metody pro verifikaci parametrických a nekonečně stavových systémů [2].

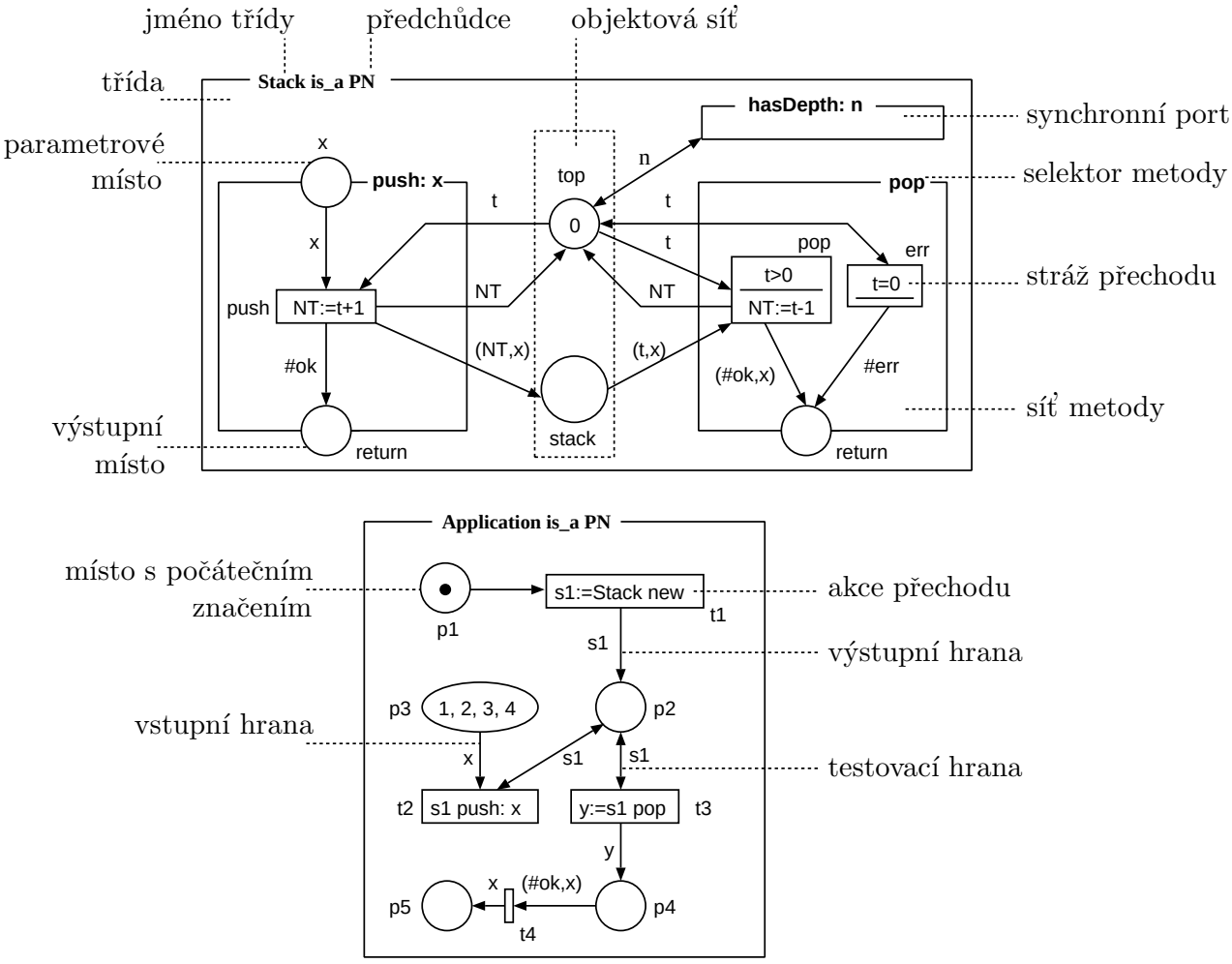
1.3 Objektově orientované Petriho sítě

Objektově orientované Petriho sítě (*OOPNs*) jsou formalizmem, který je vhodný pro modelování, prototypování a formální analýzu a verifikaci paralelních a distribuovaných systémů. *OOPNs* spojují koncepty Petriho sítí (modelování kauzality, nedeterminizmu a paralelizmu v diskrétních systémech)

[1] Objektově orientované Petriho sítě budeme v dalším textu označovat zažitou zkratkou *OOPNs* (**O**bject-**O**riented **P**etri **N**ets), kterou budeme používat vždy, když budeme o objektově orientovaných Petriho sítích hovořit jako o jazyku, o formalizmu nebo o systému. Když však budeme chtít označit určitý model popsaný pomocí *OOPNs*, budeme používat zkratku *OOPN* (**O**bject-**O**riented **P**etri **N**et).

s objektovou orientací (zapouzdření, dědičnost, polymorfizmus). Zavedení tříd a objektů do Petriho sítí umožňuje snadné a přirozené vytváření rozsáhlých a přesto přehledných modelů. Formální analýza a verifikace *OOPN* modelů je možná, protože *OOPNs* jsou formálně definovány, ale je poměrně složitá, protože *OOPNs* jsou bohatým modelovacím formalizmem.

Základní principy *OOPNs* nyní ukážeme na jednoduchém příkladu *OOPN* modelu (obr. 1), který se skládá ze dvou tříd — třída Stack představuje model zásobníku, zatímco třída Application ukazuje jeho použití.



Obrázek 1: Jednoduchý příklad *OOPN* modelu

Principy objektové orientace jsou v *OOPNs* využity pro strukturování — třídy jsou uspořádány relací dědičnosti, pro asynchronní komunikaci slouží metody a pro synchronní komunikaci zase synchronní porty. Prvky vysokoúrovňových Petriho sítí slouží pro popis chování objektů a metod. Přechody jsou s místy spojeny hranami, na nichž se nacházejí inskripční výrazy. Přechody mohou obsahovat stráže, jejichž prostřednictvím lze volat synchronní porty, a akce, z nichž lze zase volat metody. Dynamické chování *OOPNs* je založeno na konceptu událostí.

2 Cíl práce

Cílem této práce je zlepšit možnosti formální analýzy a verifikace softwarových systémů popsaných pomocí *OOPNs*. Tato oblast je ale poměrně rozsáhlá, proto se zaměříme pouze na vybrané metody a techniky. První oblastí, které se budeme věnovat, je typová analýza *OOPNs*. *OOPNs* nejsou silně typované, ale určité informace o typech značek, které se mohou dostat do jednotlivých míst modelu, mohou být užitečné. Prvním cílem této práce je proto navrhnout metody pro automatické zjišťování množin typů značek, které se při evoluci mohou dostat do jednotlivých míst *OOPN* modelů.

Druhý přístup k analýze *OOPNs*, kterým se v této práci budeme zabývat, je založen na generování a prohledávání stavových prostorů. Hlavním problémem těchto metod je problém stavové exploze. Druhým cílem této práce je proto paralelizace generátoru stavových prostorů *OOPNs* na architekturách se sdílenou pamětí, která by měla omezit vliv problému stavové exploze na praktickou použitelnost těchto metod.

3 Zvolené metody zpracování

Při návrhu metod pro automatickou typovou analýzu *OOPNs* budeme vycházet z jejich dynamického chování, přičemž se jeho vhodnou aproximací pokusíme tyto metody urychlit. Zavedeme proto tzv. *symbolickou simulaci*, při které budou běžné značky nahrazeny svými typy a běžné události budou nahrazeny symbolickými událostmi, které budou pracovat se symbolickými značkami (tedy s typy). Dynamický charakter *OOPNs*, který zvyšuje jejich vyjadřovací schopnosti, komplikuje jejich formální analýzu a verifikaci. Navrhneme zde dvě metody, které se s dynamickým vytvářením objektů a metod při typové analýze *OOPNs* vypořádají, každá však jiným způsobem. Základní principy typové analýzy ověříme implementací prototypu typového analyzátoru *OOPNs* v Javě.

Při implementaci generátoru stavových prostorů *OOPNs* budeme pro úsporu práce vycházet z typového analyzátoru *OOPNs*. Jako implementační jazyk proto použijeme také Javu, která se v oblasti vysoce náročných výpočtů teprve prosazuje. Pro vlastní paralelizaci generátoru stavových prostorů *OOPNs* pak použijeme nástroj *JOMP*, který zpřístupňuje pro Javu standard *OpenMP* používaný pro jazyky C a Fortran. Vlastní paralelizace na architekturách se sdílenou pamětí bývá relativně jednoduchá. Náročnější je pak nalezení optimálního rozdělení dat na sdílená a lokální a synchronizace přístupu ke sdíleným datům pro dosažení co nejvyššího paralelního zrychlení. Popíšeme zde výsledný paralelní algoritmus a zhodnotíme dosažené experimentální výsledky.

4 Hlavní výsledky práce

4.1 Automatická typová analýza *OOPNs*

OOPNs nejsou silně typované, tzn. *OOPN* modely neobsahují explicitní definice typů značek, které jsou přípustné v místech či v hranových výrazech. Tato vlastnost *OOPNs*, která byla při návrhu *OOPNs* převzata z jazyka Smalltalk, je výhodná pro rychlý návrh a prototypování, protože při změnách modelu není nutné udržovat konzistenci typových deklarací. V některých případech však mohou být informace o typech užitečné. Cílem typové analýzy je proto určit typy značek, které se mohou při evoluci *OOPN* modelu vyskytnout v jeho jednotlivých místech.

Výsledky typové analýzy mohou být využity v několika oblastech. Typovou analýzu můžeme chápat jako speciální prostředek pro ladění *OOPN* modelů — při modelování máme obvykle určitou představu o tom, které typy značek se mohou dostat do jednotlivých míst modelu. Pokud se výsledky typové analýzy pro daný model liší od našich představ, může to signalizovat chybu v modelu. Informace získané typovou analýzou mohou být také použity jako automaticky generovaná součást dokumentace modelu. V případě, že bychom chtěli *OOPN* modely překládat do jiných formalizmů, které umožňují formální analýzu a verifikaci a které jsou obvykle silně typované, bylo by možné výsledky typové analýzy využít pro zjištění v syntaxi *OOPNs* chybějící informace o typech. Znalost typů také může umožnit další optimalizace simulátoru *OOPNs* a generátoru stavových prostorů *OOPNs*.

4.1.1 Symbolická simulace

Automatická typová analýza *OOPNs* je založena na vhodné aproximaci chování *OOPNs* s cílem snížit její časovou i prostorovou složitost na přijatelnou úroveň. Touto aproximací se však zavádí určitá nepřesnost, takže množiny typů, které dostaneme jako výsledek typové analýzy, mohou být větší, než bychom dostali přesným odvozením z úplného stavového prostoru. Jsme-li si vědomi této nepřesnosti ve výsledcích, jsou takto získané výsledky stále užitečné.

Jádrem použité aproximace je symbolická reprezentace, kdy jednotlivé značky jsou nahrazeny svým typem (např. $7 \mapsto \mathsf{int}$). Místo běžných událostí jsou pak prováděny symbolické události, při kterých nejsou ze vstupních míst odebírány značky a stráže přechodů nejsou vyhodnocovány běžným způsobem (např. přesné vyhodnocení výrazu $\mathsf{int} \geq \mathsf{int}$ nedává smysl), je pouze kontrolováno, zda ve vstupních a testovacích místech jsou příslušné symbolické značky. Je také nutné předefinovat primitivní funkce, které se vyskytují v akcích přechodů, aby pracovaly s typy (např. $\mathsf{int} + \mathsf{int} = \mathrm{int}$). Protože při symbolickém provádění

nemá počet značek vliv na proveditelnost přechodu, není nutné uchovávat informaci o počtu symbolických značek v místě, takže místo multimnožin, které se vyskytují ve značení míst a na hranových výrazech, můžeme pracovat pouze s množinami. Symbolickou simulaci ukončíme v okamžiku, kdy se provedením žádné symbolické události nezmění symbolické značení modelu.

Z praktických důvodů požadujeme, aby typová analýza skončila. Množina základních typů *OOPN* modelu je samozřejmě konečná. Nekonečnost prostoru typů však mohou způsobit symbolické seznamy[2], které mohou být složeny ze základních typů a ze symbolických seznamů. Abychom omezili tento potenciálně nekonečný prostor typů na konečný a tím zaručili konečnost symbolické simulace, zavedeme limit na maximální hloubku zanoření a na maximální délku symbolických seznamů, které budou zpracovány přesně. Když by pak některý symbolický seznam měl překročit stanovené meze, bude nahrazen přibližnou symbolickou hodnotou, která nebude obsahovat informaci o hloubce zanoření ani o pozicích jednotlivých základních v něm obsažených.

4.1.2 Statická typová analýza

Statická i dynamická typová analýza je založena na symbolické simulaci. Obě metody se ale liší v přístupu k dynamickému charakteru *OOPNs* tedy k vytváření objektů a k volání metod. Při statické typové analýze je na počátku pro každou třídu vytvořena jedna objektová síť a pro každou metodu jedna instance její sítě. Počet sítí se pak v průběhu symbolické simulace nemění.

Dynamické chování *OOPNs* je založeno na konceptu událostí, což se pochopitelně odráží i při symbolické simulaci, kde však místo o běžných událostech hovoříme o událostech symbolických. Proveditelnost symbolických událostí závisí na symbolickém značení vstupních a testovacích míst a na typové konzistenci navázání proměnných na hranách, ve stráži a v akci přechodu. Efekt jednotlivých typů symbolických událostí je však odlišný od běžných událostí.

Provedení symbolické události N nemá při statické typové analýze za následek vznik nového objektu dané třídy, ale pouze přiřazení identifikátoru této třídy do příslušné proměnné. Při provedení symbolické události F se pouze přidají příslušné symbolické značky do parametrových míst v síti volané metody. Symbolická událost J pak neruší síť metody, ale v souladu s výstupními hranami přechodu přidá do příslušných míst symbolickou značku z výstupního místa této metody. Při provádění výpočtů při symbolické události A místo klasických výpočtů probíhají pouze odpovídající symbolické výpočty nad typy (např. $\mathsf{int} + \mathsf{int} = \mathrm{int}$).

[2]Seznamy zde chápeme jako jeden z možných pohledů na n-tice.

4.1.3 Dynamická typová analýza

Při návrhu statické typové analýzy jsme kladli důraz na její rychlost, ale při výskytu obecných datových struktur v modelu můžeme získat poněkud nepřesné výsledky. Pro získání přesnějších výsledků typové analýzy při zachování přijatelné rychlosti výpočtu lze použít tzv. dynamickou typovou analýzu, jejíž principy zde nyní stručně popíšeme.

Základním rozdílem mezi dynamickou a statickou variantou typové analýzy je způsob, jakým jsou zpracovány objekty a metody. Zatímco při statické typové analýze existovala právě jedna instance každé sítě, při dynamické typové analýze jsou jednotlivé instance sítí rozlišovány. Ke každému objektu je při jeho vzniku připojena instance objektové sítě, která je označena počátečním symbolickým značením. Pokud se v jednom místě objeví dva objekty stejné třídy a symbolické značení objektové sítě jednoho z nich pokrývá symbolické značení objektové sítě druhého, můžeme tento druhý objekt odstranit. Pokrývající objekt totiž poskytuje z hlediska symbolické simulace minimálně stejné chování jako objekt pokrytý. Při rušení pokrytého objektu je však nutné přesunout všechny jeho rozpracované metody k objektu pokrývajícímu a také nahradit všechny reference na rušený objekt, které se vyskytují v aktuálním symbolickém značení, referencemi na pokrývající objekt.

Tím jsme vyloučili výskyt dvou objektů se stejným symbolickým značením v jednom místě. Pro zajištění konečnosti dynamické typové analýzy nám tedy zbývá vyřešit poslední problém související se zavedením objektů. Již ukončení statické typové analýzy bránily neomezené seznamy, což jsme vyřešili zavedením maximální hloubky zanoření a délky seznamů, které byly zpracovány přesně. Při dynamické typové analýze se mohou vyskytnout neomezené seznamy dynamicky vytvářených symbolických objektů. Zavedeme proto limit na maximální vzdálenost (měřeno počtem referencí) od počátečního objektu tak, že symbolické události, po jejichž provedení by se ve značení objevil příliš vzdálený objekt, nebudou proveditelné. Domníváme se, že když zvolíme tento limit dostatečně velký, nemuselo by mít zavedení tohoto limitu vliv na přesnost získaným výsledků.

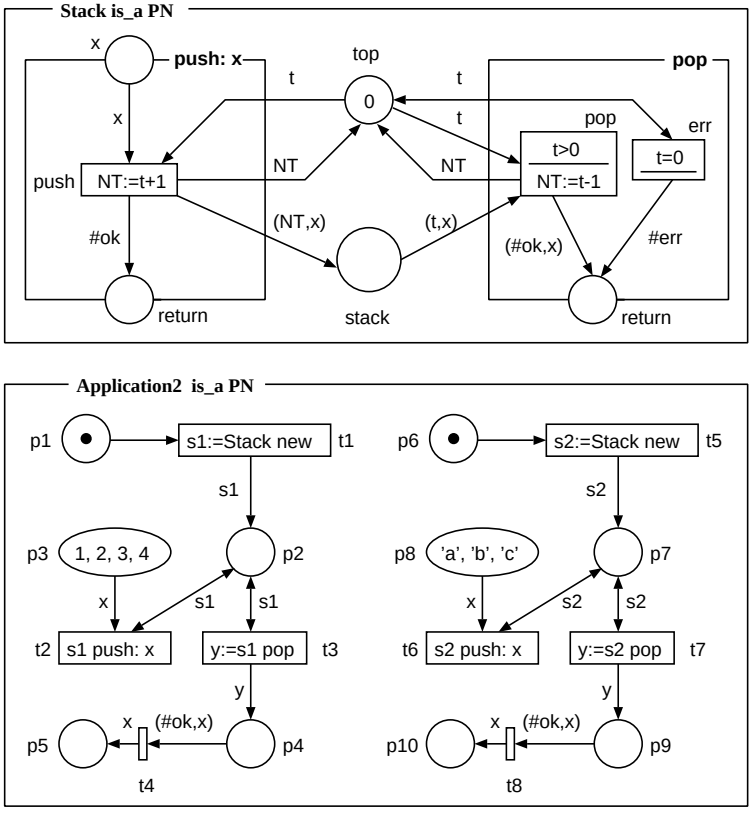
Tímto jsme se tedy vypořádali s objekty v rámci dynamické typové analýzy a zbývá nám vyřešit zpracování metod. Při vyvolání symbolické metody ze symbolicky prováděného přechodu je vytvořena nová instance sítě této metody, která je označena počátečním symbolickým značením a do jejích míst pro parametry jsou umístěny symbolické argumenty. Reference na tuto metodu je uchována v navázání právě prováděného přechodu v proměnné mid. Protože při symbolické simulaci neodebíráme z míst symbolické značky, které se do nich dostaly, lze se domnívat, že výsledek typové analýzy nezáleží na pořadí výskytu jednotlivých symbolických událostí, neboť výskyt jedné symbolické události nemůže zabránit budoucímu provedení ostatních proveditelných

symbolických událostí. Můžeme proto při symbolické simulaci požadovat, aby typová analýza vyvolané metody byla dokončena co nejdříve. Symbolická simulace se proto soustředí právě na tuto novou metodu, případně na další metody z ní vyvolané.

Po dokončení typové analýzy dané metody bude tato metoda ukončena následujícím způsobem. Aktuální symbolické značení míst její sítě bude přidáno ke stávajícím množinám typů značek, které se mohou dostat do jednotlivých míst dané metody, a pro všechny symbolické značky ve výstupním místě se dokončí symbolické provedení přechodu, který metodu vyvolal. Síť metody je pak zrušena a spolu s ní i všechny symbolické objekty, na které tímto krokem přestal existovat odkaz.

Abychom zaručili konečnost dynamické typové analýzy, omezíme potenciální cykly volání metod následujícím pravidlem. Pokud je metoda (i nepřímo) vyvolána sama sebou, nejsou v ní proveditelné ty události, které jsou aktuálně rozpracovány v nadřazených instancích této metody. Protože množina symbolických událostí, které se mohou vyskytnout v dané síti, je konečná, musí být konečná i posloupnost volání metod.

Ukázka výsledků dynamické typové analýzy je na obrázku 2.



a) *OOPN* model se dvěma zásobníky

Místo	Výsledné typy
top	{int}
stack	{(int,int), (int,char)}
push: x	{int, char}
push: return	{symbol}
pop: return	{symbol, (symbol,int), (symbol,char)}
p1	{symbol}
p2	{Stack}
p3	{int}
p4	{symbol, (symbol,int), (symbol,char)}
p5	{int}
p6	{symbol}
p7	{Stack}
p8	{char}
p9	{symbol, (symbol,int), (symbol,char)}
p10	{char}

b) Výsledky analýzy

Obrázek 2: Příklad výsledků dynamické typové analýzy *OOPNs*

4.2 Paralelizace generátoru stavových prostorů *OOPNs*

Základním problémem metod formální analýzy a verifikace založených na generování a procházení stavových prostorů je problém stavové exploze, který brání jejich úspěšnému nasazení i pro rozsáhlejší modely. Pro omezení praktických dopadů problému stavové exploze lze využít řadu optimalizací založených na redukcích stavového prostoru nebo paralelní přístup. Právě využitím paralelního přístupu při generování a procházení stavových prostorů se zabýváme v této práci. Jako modelovací formalizmus byly použity *OOPNs*, ale dále popsaný paralelní algoritmus lze použít i pro další modelovací formalizmy.

Než se budeme věnovat paralelizaci generátoru stavových prostorů *OOPNs*, popíšeme zde jeho základní principy. Stavový prostor je dán množinou stavů, množinou strukturálních přechodů, množinou sémantických přechodů a počátečním stavem. Stav *OOPN* modelu je relativně složitý a odpovídá systému vzájemně propojených objektů, přičemž každý objekt se skládá z právě jedné instance objektové sítě a z několika instancí sítí právě rozpracovaných metod. Stav každé sítě je dán značením všech míst sítě a případným odkazem na rozpracovanou metodu u přechodů sítě. Množina strukturálních přechodů odpovídá množině přechodů a množina sémantických přechodů událostem. Počáteční stav *OOPN* modelu odpovídá jednomu objektu hlavní třídy, který obsahuje pouze objektovou síť označenou počátečním značením a bez odkazů na rozpracované metody u přechodů.

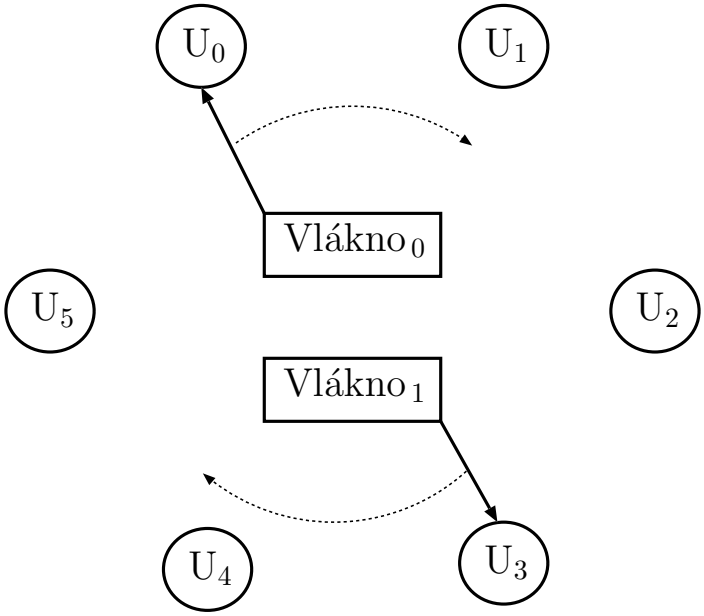
Při generování stavového prostoru je obvykle omezujícím faktorem dostupná operační paměť, proto se téměř vždy do stavového prostoru ukládají pouze unikátní stavy. Pokud je z některého uzlu vygenerován následník, který již ve stavovém prostoru existuje, je tento existující stav považován za následníka a nově vygenerovaný duplicitní stav není nutné znovu ukládat. Při vygenerování každého stavu je však nutné projít dosud vygenerovaný stavový prostor a zjistit, zda nově vygenerovaný stav je unikátní nebo duplicitní. To je časově náročné zvláště pro větší stavové prostory. Hledání stavů ve stavovém prostoru i operace porovnání dvou stavů musí být proto co nejrychlejší. Rychlé vyhledávání ve stavovém prostoru je zajištěno vyhledávací tabulkou (angl. *hash table*), v níž se k jednotlivým stavům přistupuje pomocí vyhledávacího klíče.

Hlavními úkoly při paralelizaci libovolného algoritmu je rozdělení práce mezi jednotlivé procesory a zajištění synchronizovaného přístupu ke sdíleným datům. V našem případě se proto budeme dále zabývat stavovým prostorem, který je realizován vyhledávací tabulkou, seznamem dosud neprozkoumaných stavů a oddílem pro ukládání značení.

4.2.1 Rozdělení vyhledávací tabulky

Vyhledávací tabulku, která slouží pro rychlý přístup ke stavům, rozdělíme na více částí (vzhledem ke struktuře vyhledávací tabulky to není obtížné). Ke každé takové tabulce pak přiřadíme seznam dosud nezpracovaných stavů, které by měly být uloženy právě do této vyhledávací tabulky. Přirozený způsob, kterým jsme začali, je rozdělení vyhledávací tabulky na tolik částí, kolik máme k dispozici procesorů (vláken týmu).

Při tomto přístupu však velice často docházelo k nerovnoměrnému rozdělení práce mezi jednotlivé procesory. Rozdělili jsme proto vyhledávací tabulku na více částí, než je vláken. Jednotlivá vlákna se pak u jednotlivých tabulek střídají, jak ukazuje obrázek 3, na kterém je počáteční přiřazení vláken k datovým uzlům (vnitřní struktura uzlů a vláken je na obrázku 4).



Obrázek 3: Dělení práce mezi dvě vlákna

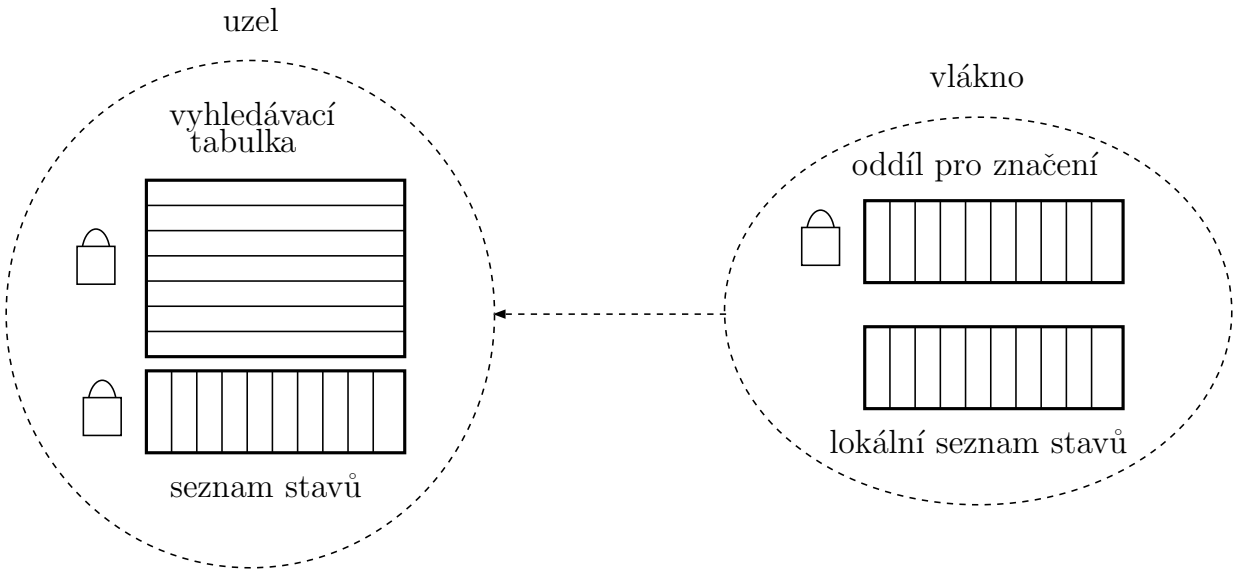
Pokud vlákno zpracuje všechny nezpracované stavy příslušející k danému uzlu, tento uzel opustí a hledá další uzel, který obsahuje nějaké nezpracované stavy. Pokud narazí na uzel přiřazený jinému vláknu, přeskočí ho. Experimentálně jsme určili, že počet datových uzlů je vhodné zvolit jako trojnásobek počtu vláken, které se u jednotlivých vyhledávacích tabulek střídají.

4.2.2 Přístup k nezpracovaným stavům

Na počátku je v seznamu nezpracovaných stavů pouze počáteční stav modelu. Generování stavového prostoru pak spočívá v postupném zpracování dosud nezpracovaných stavů, přičemž zpracováním se rozumí zkontrolování, zda se již stav ve stavovém prostoru nenachází, a pokud ne, vygenerují se všichni

jeho následníci, kteří jsou pak přidáni na konec seznamu dosud nezpracovaných stavů.

Kdyby byl seznam dosud nezpracovaných stavů sdílený všemi vlákny, bylo by sice nejlepší rozdělení práce mezi všechna vlákna, ale docházelo by k častým kolizím. Rozdělili jsme proto tento seznam na několik částí, které pochopitelně korespondují s rozdělením vyhledávací tabulky (viz obrázek 4).



Obrázek 4: Vnitřní datová struktura uzlů a vláken

K takovému částečnému seznamu nezpracovaných stavů pak přistupují všechna vlákna týmu, protože do něj vkládají vygenerované stavy, které patří do příslušné vyhledávací tabulky. Nejčastěji však k němu přistupuje vlákno, které zpracovává daný uzel, protože z něj jednotlivé stavy vyjímá a zpracovává. Abychom omezili konflikty mezi vlákny na minimum, má každé vlákno svůj lokální seznam nezpracovaných stavů, který zpracovává, dokud není prázdný. Pak ho prostě vymění za seznam stavů, do kterého zatím mohla ostatní vlákna přidat další stavy ke zpracování. Pokud ani v tomto seznamu stavů není nic ke zpracování, vlákno uzel opustí.

Zkoumali jsme také dva možné způsoby řazení dosud nezpracovaných stavů. Došli jsme k na první pohled překvapivému závěru, že nezpracované stavy je lepší ukládat do zásobníku (LIFO) než do fronty (FIFO). Při generování rozsáhlých stavových prostorů je stav na začátku fronty ve vyrovnávací paměti s menší pravděpodobností, než je tomu u stavu na vrcholu zásobníku. Použitím zásobníku se tedy zvýší úspěšnost hledání ve vyrovnávací paměti (angl. *cache hit rate*) a tím se urychlí běh programu. Druhou možnou příčinou je promíchání stavů generovaných v různých fázích generování v zásobníku, které může mít za následek lepší rozdělení nezpracovaných stavů mezi procesory a tím ke snížení počtu kolizí.

4.2.3 Oddíl pro značení míst

Poslední datovou strukturou (z obrázku 4), kterou jsme se dosud nezabývali, je oddíl pro ukládání značení míst. Před tím, než se tímto problémem budeme zabývat, musíme si uvědomit, že čtení z tohoto oddílu bude velice časté, zatímco zapsání nové hodnoty do něj bude – kromě počáteční fáze – spíše výjimečné.

Oddíl pro ukládání značení by mohl být společný pro všechna vlákna, což by mohlo zvýšit počet kolizí. Druhou možností je přiřadit každému vláknu lokální oddíl pro ukládání značení. Do tohoto oddílu se vkládají značení při generování stavů. Protože však vlákna pracují i se stavy, které byly vygenerovány jinými vlákny, je nutné, aby všechny lokální oddíly obsahovaly stejné hodnoty. Proto je nutné, aby nově nalezené značení bylo zapsáno do oddílů všech vláken. Tento postup však zvyšuje paměťové nároky. Experimentálně jsme ověřili, že velikost oddílu pro značení je v porovnání s velikostí stavového prostoru zanedbatelná, takže jej lze udržovat na několika místech. Může se však pro některé typy stavových prostorů ukázat, že jako výhodnější bude varianta se společným oddílem pro značení.

4.2.4 Shrnutí dosažených výsledků

Použitím všech výše uvedených optimalizací paralelní verze generátoru jsme oproti optimalizované sekvenční verzi dosáhli paralelního zrychlení 3,3 při použití šestnácti procesorů na systému Sun Fire 15k a zrychlení až 2,3 při použití čtyř procesorů na systému Sun Enterprise 450. I přes relativně nízkou efektivitu lze tyto výsledky považovat za uspokojivé. Kromě dosažení určitého zrychlení se nám podařilo na obou architekturách dosáhnout velice vyváženého rozdělení práce mezi jednotlivá vlákna, což je nezbytná podmínka pro dosažení vysoké efektivity paralelních výpočtů.

Hlavním problém, který brání vyššímu paralelnímu zrychlení generátoru, je příliš vysoká režie spojená se správou paměti – *garbage collecting*. Možnou cestou k odstranění tohoto problému je implementovat uživatelské přidělování paměti, které bylo s úspěchem použito v podobné situaci v [1]. Dalším řešením je výrazně zredukovat počet vytvářených objektů, nicméně vzhledem k objektově orientovaným rysům *OOPNs* je těžké implementovat stavový generátor bez Java objektů a bez *garbage collectingu*. Pokud bychom však chtěli zkoumat stavový prostor modelů popsaných formalizmem s nižší dynamikou, lze předpokládat, že výše navržený algoritmus bude velice dobrý.

5 Závěr

V této práci byly navrženy nové metody a techniky pro formální analýzu *OOPN* modelů. Význam této práce proto spočívá především v rozvinutí výsledků dříve dosažených dr. Janouškem [8] a dr. Vojnarem [22] v oblasti *OOPNs*. Určitý dopad na oblasti mimo *OOPNs* a díky použití Javy a nástroje *JOMP* dokonce i na oblasti mimo formální analýzu a verifikaci může mít část práce věnovaná paralelnímu zpracování stavových prostorů *OOPNs*.

5.1 Dosažené výsledky

Dosažené výsledky lze rozdělit do dvou oblastí. Nejdříve byly popsány základní principy automatické typové analýzy a jako hlavní přínos v této oblasti byly navrženy dva algoritmy pro typovou analýzu — tzv. statická a dynamická typová analýza. Obě metody se snaží vyřešit problémy spojené zejména s dynamikou *OOPNs*, každá však jiným způsobem. Statická typová analýza problémy řeší zjednodušením situace, čímž se sice výpočet urychlí ale za cenu zanesení dalších nepřesností do výsledků. Dynamická typová analýza se naopak snaží *OOPN* objekty a metody zpracovat přesně za cenu složitějšího a pomalejšího algoritmu.

Druhou oblastí, ve které byly v rámci této práce dosaženy zajímavé výsledky, je oblast paralelního zpracování stavových prostorů *OOPNs* na architekturách se sdílenou pamětí. Byl navržen a implementován algoritmus pro paralelní generování stavových prostorů zahrnující detekci uváznutí. Tento algoritmus obsahuje unikátní mechanizmus pro synchronizaci přístupu k vyhledávací tabulce i velice účinný mechanizmus pro rozdělování práce mezi jednotlivé procesory. Zde navržené principy a techniky proto mohou být užitečné i pro paralelizaci dalších podobných problémů.

5.2 Možnosti dalšího výzkumu

OOPNs jsou bohatý formalizmus, a proto je implementace nástrojů pro práci s nimi značně časově náročná. V rámci bakalářských a diplomových prací lze očekávat úplné dokončení automatického typového analyzátoru *OOPNs*. Dále se plánuje propojení vytvořeného prototypu generátoru stavových prostorů *OOPNs*, který zatím nepodporuje všechny vlastnosti *OOPNs*, s nově vytvářeným otevřeným procesorem a simulátorem *OOPNs* [9].

V paralelizaci generátoru stavových prostorů byly dosaženy dobré výsledky, nicméně hlavním problémem, který brání dosažení vyššího paralelního zrychlení, stále zůstává *garbage collecting*. Lze se pokusit ještě více zredukovat paměťové operace nutné pro generování a ukládání stavů *OOPNs*, nicméně

vzhledem k dynamickému charakteru *OOPNs* nebude možné *garbage collecting* zcela eliminovat. Zajímavým směrem výzkumu by také mohla být aplikace navrženého paralelního algoritmu pro formalizmy s menší dynamikou chování a s jednoduššími stavy (např. P/T Petriho sítě).

Aktuální verze generátoru stavových prostorů *OOPNs* umožňuje pouze detekci uváznutí. Dalším logicky navazujícím krokem je tedy rozšíření generátoru o paralelní dotazování nad stavovými prostory *OOPN*, které je však poměrně náročné vzhledem k dynamice *OOPNs* plynoucí z jejich objektové orientace.

5.3 Související publikace

Výčet autorových publikací věnujících se dílčím výsledkům této práce zde uvádíme v chronologickém pořadí. V práci byly využity výsledky dosažené již v rámci inženýrského studia [10, 11]. Problému efektivního vyhodnocování izomorfizmu grafů, které je důležité při ukládání vygenerovaných stavů do stavového prostoru, se věnuje [13]. Motivaci pro typovou analýzu a základní metody pro její implementaci lze nalézt v [16]. Zmapování metod založených na generování a procházení stavových prostorů a metod pro omezení problému stavové exploze se zhodnocením jejich použitelnosti v kontextu *OOPNs* je v [12]. *OOPN* model protokolu *ABP* [14] lze využít jako případovou studii pro ověřování vytvářených nástrojů formální analýzy a verifikace. Integrace nástrojů pro typovou analýzu a pro generování a procházení stavových prostorů *OOPNs* usnadňující implementaci identifikací částí společných oběma nástrojům byla popsána v [17]. Zkušenosti získané z implementace typového analyzátoru *OOPNs* jsou shrnuty v [15].

Literatura

[1] Allmaier, S. C., Horton, G. Parallel Shared-Memory State-Space Exploration in Stochastic Modeling. In Bilardi, G., et al., editoři, *Solving Irregularly Structured Problems in Parallel, 4th International Symposium, IRREGULAR'97*, volume 1253 of *Lecture Notes in Computer Science*, Paderborn, Germany, 1997. Springer. s. 207–218. ISBN 3-540-63138-0.

[2] Bouajjani, A., Habermehl, P., Vojnar, T. Abstract Regular Model Checking. In *Computer Aided Verification - CAV'2004*, volume 3114 of *Lecture Notes in Computer Science*, Berlin, Germany, 2004. Springer Verlag. s. 372–386.

[3] Češka, M., Haša, L., Vojnar, T. Partial-Order Reduction in Model Checking of Object-Oriented Petri Nets. In *Computer Aided Systems Theory*

– *EUROCAST 2003*, volume 2809 of *Lecture Notes in Computer Science*, Berlin, Germany, 2003. Springer-Verlag. s. 265–278. ISBN 3-540-20221-8.

[4] Češka, M., Janoušek, V., Vojnar, T. PNtalk – A Computerized Tool for Object-Oriented Petri Nets Modelling. In Pichler, F., Moreno-Díaz, R., editoři, *Computer Aided Systems Theory – EUROCAST'97*, volume 1333 of *Lecture Notes in Computer Science*, Las Palmas de Gran Canaria, Canary Islands, Spain, 1997. Springer-Verlag. s. 591–610. ISBN 3-540-63811-3.

[5] Češka, M., Skácel, M. Petri Net Tool PESIM. In *5th International Workshop on Petri Nets and Performance Models*, Toulouse, France, 1993.

[6] Clarke, E. M., Grumberg, O., Peled, D. A. *Model Checking*. The MIT Press, Cambridge, Massachusetts, United States of America, 1999. 314 s. ISBN 0-262-03270-8.

[7] Hrubý, M., Rábová, Z. Modelling of Real-world Objects using the HELEF Language. *ASU Newsletter: a publication of the Association of SIMULA Users.*, 38(1):47–57, 2002. ISSN 1102-593X.

[8] Janoušek, V. *Modelování objektů Petriho sítěmi*. Disertační práce, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, Brno, Česká republika, 1998. 137 s.

[9] Kočí, R. The Open Architecture of the PNtalk System. In *Proceedings of the International Conference and Competition – Student EEICT'2003*, Brno, Česká republika, 2003. s. 358–362. ISBN 80-214-2401-X.

[10] Křena, B. Podpora pro analýzu stavových prostorů objektově orientovaných Petriho sítí. In Baštinec, J., Diblík, J., editoři, *Sborník prací studenů a doktorandů*, Brno, Česká republika, květen 2000. Akademické nakladatelsví CERM, s.r.o. s. 192–194. ISBN 80-7204-155-X.

[11] Křena, B. *Podpora pro analýzu stavových prostorů objektově orientovaných Petriho sítí*. Diplomová práce, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, Brno, Česká republika, červen 2000. 59 s.

[12] Křena, B. First Approach to Model Checking in Object-Oriented Petri Nets. In Štefan, J., editor, *Proceedings of XIIIrd International Autumn Colloquium on Advanced Simulation of Systems, ASIS'2001*, Velké Losiny, Česká Republika, září 2001. MARQ Ostrava. s. 105–110. ISBN 80-85988-61-5.

[13] Křena, B. The Graph Isomorphism Problem. In Arnošt, V., editor, *Proceedings of 7th Conference Student FEI 2001*, volume 2, Brno, Česká republika, duben 2001. Vysoké učení technické v Brně. s. 343–347. ISBN 80-214-1860-5.

[14] Křena, B. A Case Study: Modelling Alternating Bit Protocol by PNtalk. In Štefan, J., editor, *Proceedings of 36th International Conference on Modelling and Simulation of Systems, MOSIS'02*, volume I, Rožnov pod Radhoštěm, Česká republika, duben 2002. MARQ Ostrava. s. 65–72. ISBN 80-85988-71-2.

[15] Křena, B. Object-Oriented Petri Nets and their Application and Type Analysis. *Information Technologies and Control*, 1(1):27–31, 2003.

[16] Křena, B., Vojnar, T. Type Analysis in Object-Oriented Petri Nets. In Zendulka, J., editor, *Proceedings of 4th International Conference on Information System Modelling, ISM'01*, Hradec nad Moravicí, Česká Republika, květen 2001. MARQ Ostrava. s. 173–180. ISBN 80-85988-51-8.

[17] Křena, B., Vojnar, T., Češka, M. Integrated Type Analyzer and State Space Generator of Object-Oriented Petri Nets. In *Brazilian Petri Net Meeting*, Natal, Brazil, září 2002. 6 s.

[18] Peringer, P. *Hierarchické modelování na bázi komunikující objektů.* Disertační práce, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, Brno, Česká republika, 1996. 82 s.

[19] Petri, C. A. *Kommunikation mit Automaten.* Disertační práce, Institut für Instrumentelle Mathematik, University Bonn, Germany, 1962. Available as Schriften des IIM Nr. 2. (In German).

[20] Valmari, A. The State Explosion Problem. In Reisig, W., Rozenberg, G., editoři, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, s. 429–528. Springer-Verlag, 1998.

[21] Vojnar, T. *Systém PNtalk.* Technická zpráva, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, Brno, Česká republika, 1997.

[22] Vojnar, T. *Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets.* Disertační práce, Ústav informatiky a výpočetní techniky, Fakulta elektrotechniky a informatiky, Vysoké učení technické v Brně, Brno, Česká republika, 2001. 148 s.

Curriculum Vitae

Základní údaje

Jméno a příjmení:	Bohuslav Křena
Narozen:	25. 6. 1977, Brno
Vzdělání:	Ing., 2000, FEI VUT v Brně

Zahraniční stáže

- 16. 10. 2002 - 19. 12. 2002, Central Laboratory for Parallel Processing, Bulharská akademie věd, Sofie, Bulharsko, projekt BIS-21.
- 21. 5. 2003 - 9. 7. 2003, Edinburgh Parallel Computing Centre, The University of Edinburgh, Edinburgh, Skotsko, program TRACS.
- 31. 5. 2004 - 31. 7. 2004, Sofware Testing and Analysis Laboratory, Università degli Studi di Milano - Bicocca, Milano, Itálie, projekt SegraVis.

Vybrané projekty

- *Modelování, verifikace a prototypování distribuovaných aplikací s využitím Petriho sítí*, GAČR, kód GA102/00/1017, 2000-2002.
- *Prostředí pro vývoj, modelování a aplikaci heterogenních systémů*, GAČR, kód GA102/01/1485, 2001-2004.
- *Metody formální analýzy a verifikace v objektově orientovaných Petriho sítích*, FRVŠ MŠMT, kód FR1948/2002/G1, 2002.
- *Bulgarian Information Society Center of Excellence for Education, Science and Technology in 21 Century*, BIS-21, EU, kód ICA1-2000-70016, 2002.
- *Access to Research Infrastructure Action of the Improving Human Potential Programme*, TRACS, EU, kód HPRI-CT-1999-00026, 2003.
- *Automatizované metody a nástroje pro vývoj spolehlivých paralelních a distribuovaných systémů*, GAČR, kód GA102/04/0780, 2004-2006.
- *SegraVis – Syntactic and Semantic Integration of Visual Modelling Techniques*, EU, kód HPRN-CT-2002-00275, 2004-2005.

Abstract

Object-oriented Petri nets (*OOPNs*) have been developed (in [8, 22]) to support modelling, investigating, and prototyping concurrent object-oriented software systems. *OOPNs* join together Petri net concepts (causality, nondeterminism, and concurrency) with object orientation concepts (encapsulation, inheritance, and polymorphism). Objects and classes allow *OOPNs* to be used for creating large and well-arranged models which is a significant benefit for their practical usage. Formal analysis and verification of *OOPN* models is possible due their exact mathematical definition. It is, however, quite complex because *OOPNs* are a rich formalism. In this work, we concentrate on two analysis methods in the context of *OOPNs*.

Type Analysis

The first investigated method is type analysis. *OOPNs* — unlike many of the common dialects of high-level Petri nets — are not (syntactically) strongly typed which means that modellers do not have to explicitly declare the types of *OOPN* places, the types of the variables used in *OOPN* inscriptions, and so on. The goal of the type analysis is to automatically compute the types of tokens that can get to the particular places of an *OOPN* model. Weak typing of *PNtalk* is useful in the area of prototyping, however, as mentioned below, there are situations in which it may be useful to know at least something about the types associated with the particular *OOPN* places.

Type analysis can be understood as a special means of debugging *OOPN* models: modellers usually have some intuition about what types of tokens should get into particular *OOPN* places and if the results of a type analysis are different, there may be a fault in the model. The information derived from type analysis can also be used as a part of the documentation of a model. Moreover, the results obtained from type analysis can further be useful when translating *OOPNs* into models described by some other — usually strongly typed — modelling languages. Finally, the results which may be obtained from type analysis may be used to optimize the internal representation of *OOPNs* and thus to increase the efficiency of running *OOPN* models and generating their state spaces.

In order to decrease time and space requirements of type analysis and to ensure its termination, we suitably approximate the behaviour of *OOPNs*. The price which we pay for this is the fact that the sets of the types of tokens about which we are informed that they can get into the particular places can be bigger than in reality. However, it is a safe approximation and the obtained results can still be quite useful.

We have proposed two approaches to type analysis in *OOPNs* — the so-called *static* and *dynamic type analysis*. Both methods work with symbolically represented markings and events which do not contain concrete data values but the corresponding types only. Accordingly, we do not perform classical trivial computations in guards of transitions and ports or in actions of transitions. Instead, we only derive the types of the involved variables. Next, the multiplicities of elements in initial markings and arc expressions are ignored. Moreover, we do not remove types symbolically representing some tokens once they get into a place. The main difference between static and dynamic type analysis consists in working with object and net instances: the dynamic one distinguishes (in a restricted way) different object and net instances while the static one does not. Thus, the dynamic type analysis is more exact but more complicated and slower than the static one.

Parallel State Space Generator

The second approach to analysis of *OOPNs* we deal with is based on state space generation and exploration. The main problem of such methods is the so-called *state space explosion problem* — the number of states grows exponentially with the size of the model. Thus, it is difficult or practically impossible to apply these methods directly to large systems. We can identify two main approaches for dealing with the problem. The first class of these methods consists in sophisticated generating and exploring state spaces while the second one exploits more powerful computers. In this thesis, we have decided to adapt the parallel reasoning for *OOPNs*. We have developed a parallel state space generator on architectures with shared memory using the Java programming language and the *JOMP* tool. Let us note that the methods we have proposed here are usable even outside the domain of *OOPNs*.

In the parallel algorithm, we exploit specific optimization techniques in order to achieve good work sharing among threads with as few synchronization as possible. They include synchronized access to a divided hash table, virtual threading, indirect storage of markings, etc. Using all these optimizations, we have reached parallel speedups up to 3.3 using 16 processors on the Sun Fire 15k server and speedups up to 2.3 using 4 processors on the Sun Enterprise 450 server. Moreover, the work distribution among particular threads is fine.

The main problem that prevents us to achieve an even better parallel speedup is memory management system (especially *garbage collecting*) due to the memory operations being performed sequentially. A possible approach to this problem is to implement an user-specific memory management system like in [1], to reduce the number of dynamically created objects, or to use the proposed algorithm for less dynamic formalisms.