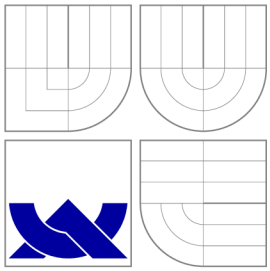
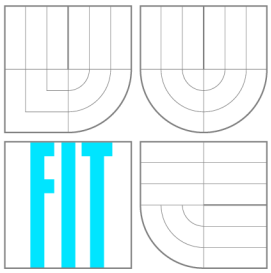


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET PETRIHO SÍTÍ PRO ŘÍDICÍ SYSTÉMY S PROCESOREM ATMEL

INTERPRET OF PETRI NETS FOR CONTROL SYSTEMS WITH ATMEL PROCESSOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL MINÁŘ
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Práce se zabývá interpretací vnořených petriho sítí popsaných jazykem PNML na procesorech *Atmel*. Zahrnuje popis cílové architektury, jež je značně omezená, jak co se týče operační a úložní kapacity, tak i výkonu. A tedy značně ovlinila návrh a implentaci samotného interpretu. Ten je hlavní náplní této práce a bude tedy detailně popsán z obou zmíněných pohledů. Protože při všech krocích byl kladen důraz na možnost verifikace a simulace samotného interpretu, byl pro implementaci použit jazyk *smalltalk* na platformě *squeak*. Ten nám umožnil testovat interpret na PC a následně jej přeložit pro cílovou architekturu v nezměněné podobě. Motivace k tomuto řešení a podrobný popis převodu jsou rovněž předmětem této práce.

Abstract

Thesis focuses on interpretation of nested petri nets described in PNML language on *Atmel* processors. It introduces this limited — from memory capacity and perfomance point of views — targeted architecture, since it greatly affected both design and implementation. The interpreter is thouroughly described from all aspects of its design. One of most important concerns in the whole process was the ability to test and verify achieved state of functionality quickly and possibly without *Atmel* processor. That's why the implentation took place on a *squeak* platform, that allowed to translate whole interpreter for targeted platform. Motivation behind this and overall process of translation is also a subject of this work.

Klíčová slova

Vysokoúrovňové Petriho síťě, referencované síťě, interpretace, Squeak, věstavěné systémy, Arduino

Keywords

High-level Petri nets, reference nets, interpretation, Squeak, embedded systems, Arduino

Citace

Michal Minář: Interpret Petriho sítí pro řídicí systémy
s procesorem Atmel, diplomová práce, Brno, FIT VUT v Brně, 2013

Interpret Petriho sítí pro řídicí systémy s procesorem Atmel

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana docenta Vladimíra Janouška

.....
Michal Minář
22. května 2013

Poděkování

Děkuji docentu Vladimíru Janouškovi za poskytnutou kostru programu a jeho koncepty, na kterých jsem stavěl a dále je rozšiřoval. Dále za všechny jeho podnětné rady a přípomínky, které mne vedli správným směrem.

Obsah

Kapitola 1

Úvod

Petriho sítě se postupem času ukázaly být mocným modelovacím nástrojem pro obecné, jednoduché i paralelní algoritmy. Navzdory své jednoduchosti a názornosti mají vysokou expresivnost i výpočetní sílu. Postupně byly vytvořeny různé varianty přidávající další výrazové prostředky. Byly do nich vkládány principy z různých programovacích paradigmat. Například spojením objektového paradigma s Petriho sítěmi umožnilo nahlížet na síť jako objekt s rozhraním, který se sám může odkazovat na jiné objekty a s nimi komunikovat. Tyto sítě jsou nazývány jako vysoko-úrovňové, kam také patří barvené Petriho sítě, hierarchické a další.

Se vzrůstající modelovací silou Petriho sítí bylo jejich využití neustále univerzálnější. Od algoritmů a protokolů, přes modely celých aplikací, systémů, až po modelování rozsáhlých platforem na různých úrovních detailů, což je umožňeno právě heirarchickým zanořováním. Příkladem aplikace popsané formalismem *referencovaných Petriho sítí* (viz 1.1.1) je například aplikace *Renew*, o které se také zmíníme dále 1.1.3.

Příkladem platformy využívající *referencovaných sítí* (dále jen *RefNets*) jako specifikačního formalismu je *Mulan*, což je implementace multi-agentního systému. *RefNets* sítě zde modelují jak chování jednotlivých agentů, protokoly, které používají, tak i celé jejich platformy a infrastrukturu. Díky těmto sítím lze specifikovat chování na tak detailní úrovni, že implementace v jiném programovacím jazyku postrádá význam, pokud tuto síť dokážeme interpretovat. Nástrojů k jejich interpretaci neustále přibývá. Je jich však zatím nedostatek pro mikroprocesory, jež jsou často cílovou platformou agentních sítí. Agenti jsou totiž často situování v reálném prostředí jako jednoduché senzory nebo roboti, kde použití obecných desktopových procesorů nepřipadá v úvahu.

Předmětem této práce je právě implementace takového interpretu, který umožní na mikroprocesorech *Atmel*, případně dalších, interpretovat agentní modely specifikované pomocí *RefNets*. Popisem jeho implementace se zabývá kapitola 5. Jak bude dále uvedeno, je zde třeba brát v úvahu jistá omezení této architektury a volit určité kompromisy v návrhu.

1.1 Teorietický základ

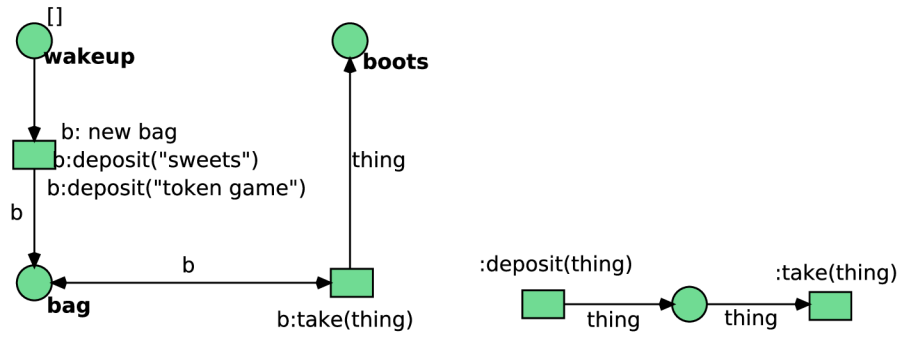
Protože se tato práce zabývá množinou různorodých technologií, s kterými čtenář nemusí být obeznámen, uvedeme si je zde ve zkrácené formě. Pro detailní teoretické podklady je možno se obrátit na jmenované zdroje, z kterých autor čerpal.

1.1.1 Referencované sítě

Jednou z variant Petriho sítí jsou *barvené Petriho sítě* (neboli *Coloured Petri Nets*, dále jen *CPN*). Jedním z nejvíce používaných dialektů (podle [23]) je dialekt zavedený Kurtem Jensenem z university Aarhus v Dánsku. Podrobnosti o těchto sítích lze nalézt v jeho třídílné monografii [17]. Zde pouze shrneme základní rozšíření:

- Definice množin barev. Barvy jsou analogií k datovým typům v programovacích jazycích.
- Každá značka je spojena s určitou definovanou barvou. Je tedy analogií k proměnné konkrétního datového typu v programovacím jazyku.
- Zaveden inskripční jazyk pro výrazy nad těmito značkami. Obsahuje základní operace nad podporovanými barvami (datovými typy).
- Barvy značek jsou manipulovány při průchodu přechodem vyhodnocením těchto výrazů.
- Přechody definují stráže, což jsou výrazy omezující jejich proveditelnost.

Referencované sítě jsou rozšířením *CPN* o objetové paradigma. Je to formalismus zahrnující koncept sítí jako objektů značek podle Rüdigera Volka [22], implementuje synchronní kanály podle [5] a umožňuje instanciovat šablony sítí. Definice množiny barev *CPN* je zde tedy rozšířena o datový typ šablony sítě a značky mohou reprezentovat jejich instance. Na



(a) Síť `santa` (b) Síť `bag`

Obrázek 1.1: Příklad *referencované sítě*

obrázku 1.1 vídíme příklad takové sítě, který byl namodelován s pomocí *Renew* 1.1.3. Síť `santa` vytváří novou instanci sítě `bag` a v tom samém přechodu volá její *uplink* `:deposit` poprvé s parametrem `"sweets"` a podruhé s `"token game"`. Toto volání je vlastní komunikací přes synchronní kanál, jehož vyústění v přechodu se nazývá `downlink` resp. `uplink` ve volající resp. volané síti.

`downlink` se zapisuje `netexpr:channelname(expr, expr, ...)` kde `netexpr` je výraz vyhodnocující se v referenci na síť.

`uplink` zápisem `:channelname(expr, expr, ...)` definuje název kanálu a jeho parametry.

Protože je kanál synchronní, jsou oba přechody (`downlink` i `uplink`) provedeny naráz. Parametry těchto kanálů mohou být vstupní a výstupní.

Podrobnější popis fungování kanálů a jejich interpretaci lze nalézt v [18].

1.1.2 PNML

Neboli *Petri Net Markup Language* je textový formát na bási *XML* určený pro zápis, úschovu a přenos Petriho sítí. Mezi jeho přednosti patří nezávislost na jakýchkoliv nástrojích a platformách. Dále maximální univerzálnost a obecnost, což umožňuje podporu libovolným druhům Petriho sítí s libovolnými rozšířeními. *PNML* chápe tyto sítě jako značené grafy, kde všechny doplňkové informace jsou uchovány jako popisky přiřazené síti, jejím místům nebo přechodům či propojením.

Zavádí *PNTD* (*Petri net type definition*), které určují správné popisky konkrétním typům sítě. Přiřazením určitého typu síti se její popis stává jednoznačným.

1.1.3 Renew

Je zkratkou pro *Reference Net Workshop*. Je to simulátor vysokoúrovňových Petriho sítí psaný v *Javě*. Je to zároveň integrované vývojové prostředí pro tyto typy sítí. Umožňuje tedy modelovat, vizualizovat a simulovat mnoho druhů Petriho sítí včetně *RefNets* (viz výše 1.1.1). Navíc umožňuje jejich export do *PNML*. Velmi zajímavý je také jeho model, který je plně popsán formalismem *RefNets*. To je podrobně zdokumentováno publikací [4].

Díky *Javě* je to nástroj multiplatformní, podporovaný všude tam, kde je podporovaná samotná *Java*. Zároveň to však znamená, že není příliš vhodný pro embedded zařízení.

My jej budeme používat jako výchozí bod pro naše experimenty s cílovou platformou a jako modelovací a simulační nástroj pro jejich verifikaci.

1.1.4 Squeak

Squeak je moderní, otevřená a plně vybavená implementace programovacího jazyka *Smalltalk* a zároveň jeho vývojové prostředí. Je sama napsána ve *Smalltalku*. Má vlastní správu paměti i souborový systém. Aplikace v něm psané jsou distribuovány jako *image* soubory. Ty obsahují všechny zdrojové kódy squeaku, celého uživatelnského rozhraní i poslední uložený stav celého grafického prostředí. Avšak díky těmto vlastnostem je *Squeak* a jeho aplikace bezproblémově přenositelný. Výčet jeho vlastností a návod k použití lze nalézt v [3].

Samotný interpret *Smalltalku* (lépe řečeno virtuální stroj) je ze *Smalltalku* převáděn do jazyka *C* a následně kompilován. To platí i pro jeho rozšíření a ovladače hardwaru. Translátor do jazyka *C* je rovněž psán ve *Smalltalku*. Autoři se řídili heslem „dělej vše ve *Smalltalku*, aby každé vylepšení činilo vše menším, rychlejším, a lepším“ [16]. Zajímavostí je, že po zhruba 3 měsících vývoje byl *Squeak* schopen simulovat svůj vlastní interpreter a tak se stal zcela samostatným.

To, že *Squeak* umožňuje své zdrojové kódy překládat přímo do *C* využijeme pro naše účely. Více o tom bude v sekci 2.1.

Smalltalk

Je to interpretovaný, čistě objektový, programovací jazyk. Byl inspirován jazykem Simula. Vznikl v 70. letech minulého století. My se budeme však zabývat především jeho podmno-

žinou nazývanou *Slang*.

Slang

Jak již bylo zmíněno výše 1.1.4, virtuální stroj *Squeaku* a jeho rozšíření, psaná ve *Smalltalku*, jsou překládány do jazyka *C* pro následnou kompilaci do jazyku symbolických instrukcí procesoru. Protože by však překlad čistě *Smalltalk*ovského kódu vyžadoval velmi náročnou a pokročilou analýzu. Využívá se pro tento kód pouze podmnožina *Smalltalku*. Ta se nazývá *Slang*.

Tento jazyk krátce představíme a uvedeme význam jeho konstrukcí z pohledu translátoru do *C* (viz níže 2.1.4).

1.2 Cílové platformy

Kdybychom se omezili pouze na jednu konkrétní platformu, jako je například *Arduino*, velmi bychom si ulehčili práci. Implementačním jazykem by se stalo *C* či *C++* a zdrojový kód by mohl využít všech nabízených možností platformy. V tom případě by však přenositelnost kódu byla velmi slabá a úsilí věnované vývoji by našlo uplatnění pouze na malé škále zařízení kompatibilních s *Arduinem* a ve speciálních případech.

Tato práce si však klade za cíl vytvořit co nejobecnější simulátor schopný běhu na mnoha platformách s větší vahou kladenou na věstavěné systémy. To nás omezuje v mnoha hlediscích a především také ve výběru implementačního jazyka. Samozřejmě se zde nabízí jazyk *C* s možností kompilace pro takřka libovolné zařízení. Z hlediska vývoje a hlavně snadnosti rozšiřitelnosti a modifikací však tato volba zcela selhává. My se nechceme o jmenované a velmi příjemné vlastnosti vyšších programovacích jazyků ochudit, avšak stejně tak je pro nás důležitá přenositelnost jazyka *C*. Naštěstí není nutné se vzdávat ani jedné z těchto předností. Mnoho vysoce-úrovňových jazyků je dnes doprovázeno překladači do symbolických instrukcí nebo dokonce do jazyka *C*. Jedním z nich je i *Smalltalk* jmenovaný výše (1.1.4) a jeho rozšířená, multiplatformní implementace *Squeak*. S využitím možnosti překladu ze *Squeak*u do *C* máme k disposici přívětivý, čistě objektově orientovaný jazyk a především kompletní vývojové prostředí a prostředky pro sdílení kódu, a k tomu bezkonkurenční přenositelnost.

Arduino je tedy pouze zástupcem věstavěných systémů, kde simulátor může být použit.

1.2.1 Arduino

Je to otevřená platforma založená na jednoduché desce s mikrokontrolérem. Je to zároveň multiplatformní vývojové prostředí pro software běžící na tomto vystavěném systému [1].

Díky tomu, že hardware i software platformy jsou otevřené, může jejich zdrojové kódy a schémata kdokoliv studovat, měnit a dál distribuovat pod danou licencí [1]. Z dalších výhod platformy jmenujme:

- Přívětivou cenu hardware.
- Širokou bázi volně stažitelných a snadno použitelných knihoven.
- Aktivní komunitu uživatelů a vývojářů.

[1] Plány desek jsou pod Creative Common License, zatímco software pod LGPL 2.1

- Cíleno na neprogramátory a nepříliš technicky zdatné lidi. Díky tomu je i pro zkušené vývojáře potřeba jen velmi krátká doba ke studiu.
- Dostupnost volně šířené kvalitní literatury a dokumentace.

Hardware

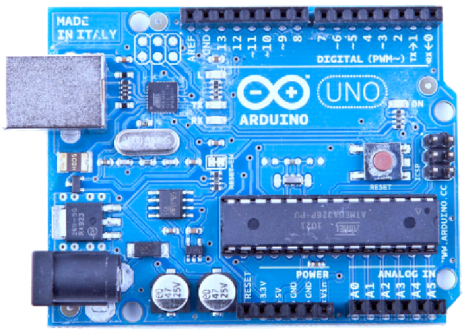
Tým vývojářů platformy vyrobil již řadu desek s různými označeními postavených na 8-bitovém RISCovém mikrokontroléru *Atmel*. Zde budeme uvažovat desku *Arduino Uno* s mikrokontrolérem *ATmega328*. Jeho základní parametry shrnuje tabulka 1.1.

Parametr	Hodnota	Parametr	Hodnota
Flash	32 kB	ADC kanálů	8
Počet pinů	32	ADC rozlišení	10 bitů
Max. Operační frekvence	20 MHz	ADC rychlost	15 kb/s
CPU	8bit. AVR	Teplotní čidlo	1
Max. I/O pinů	24	SRAM	2kB
SPI	2	EEPROM	1024
TWI	1	Čítačů/Časovačů	3
UART	1	PWM kanálů	6
32 kHz RTC	ano		

Tabulka 1.1: Základní parametry mikrokontroléru *ATmega328*

Mikrokontrolér poskytuje pouze 2 KB paměti SRAM, což není mnoho pro uložení šablon *RefNet* sítí s jejich instancemi a všemi hodnotami. Ovšem pro malé sítě toto není problém. Větší sítě však vyžadují připojení externí paměti.

Arduino Uno Obrázek 1.2 ukazuje desku *Arduino Uno* z pohledu shora. Na ní nalezneme 14 digitálních I/O pinů, 6 analogových vstupních a 6 výstupních. Deska může být napájena jak externím adaptérem, tak přes *USB* port.



Obrázek 1.2: Deska *Arduino Uno*

Rozhraní s okolním světem

Interpretované sítě mají k disposizi digitální piny pro čtení i zápis. Operace nad nimi jsou provedeny synchronně při spuštění přechodu. Dále nesmí chybět komunikace přes sériovou linku. Více o této problematice pojednává sekce **??**.

Tyto prostředky jsou v prostředí *Squeak* pouze simulovány. Vlastní simulátor proto využívá volání vysokoúrovňových univerzálních funkcí poskytnutých konkrétní implementací pro interakci s okolím.

Protože nám specifikace referencovaných sítí, jazyk *PNML* a *Arduino* platforma dávají velkou volnost a skýtají netušený potenciál, bude možné v budoucnu přidávat další komunikační protokoly a schopnosti ovládat různá periferní zařízení.

1.2.2 x86

Pro ověření správnosti implementace je výhodné mít možnost překladu a spuštění simulátoru na osobním počítači. Díky snaze o co největší přenositelnost kódu toto není problém. A lze tedy překládat i simulovat na těchto architekturách. Podorora je zahrnuta i pro 64bitové procesory a systémy (x86-64).

1.2.3 *Squeak*

Lze taktéž považovat za samostatnou platformu. Zde však neklademe příliš velký důraz na efektivitu běhu ani na prostorovou náročnost. *Squeak* je využit především pro vývoj a testování hlavní části simulátoru – *plug-in*u. Simulátor v tomto prostředí není příliš efektivní, protože emuluje spoustu datových struktur a operací, které jsou *Squeak*u vlastní a jsou v něm řešeny optimálnějšími metodami. Dalším kritickým místem je komunikační rozhraní s ostatními procesy v systému a zařízeními připojenými přes periferie. Oproti programu simulátoru, běžícímu přímo jako proces operačního systému, mají aplikace psané ve *Squeak*u omezené možnosti interakce.

Nebudeme se proto v tomto textu implementací simulátoru ve *Squeak*u zabývat. Omezíme se pouze na proces překladu do *C* a více se zaměříme na detaily hlavních programů výše zmíněných platforem.

Kapitola 2

Proces kompilace

Vývoj a ladění jakéhokoliv netriviálního programu pro embedded architekturu je náročné. Jak na návrh, který musí být robustní, tak na samotnou implementaci. Pokud jsou nalezeny chyby modelu nebo je třeba přidat do takové aplikace novou funkcionalitu, neobejde se to často bez drastického a časově náročného zásahu do existujícího kódu. Obvykle je v těchto případech nutné začít od začátku.

Dalším omezením, s tím souvisejícím, je nutnost použití nízkoúrovňového jazyka. Cílová architektura sice podporuje kompilaci vysokoúrovňových programovacích jazyků jako je *C++*. Z nich lze však využít často jen zlomek poskytovaných vlastností. Navíc tímto krokem často ztrácíme kontrolu nad správou paměti, která je na těchto architekturách kritická. Proto se často musíme smířit s programováním v jazyce *C*. V současné době pro embedded systémy stále ještě platí, že assembler, *C* a velmi limitované *C++* jsou jedinné, rozumně použitelné jazyky [2].

Na naši aplikaci interpretu jsou však kladeny nároky, které přímo odrazují od implementace pro vestavěné systémy. Interpreter by měl být snadno rozšiřitelný o nové funkce, stejně jako *Renew*, jehož výstup interpretujeme a formalismus *PNML*, který zpracováváme. Chceme mít možnost interpreter testovat, verifikovat a ladit v přívětivých podmínkách. Bohužel tyto požadavky nelze splnit na samotné cílové architektuře pouhým spuštěním zkompilovaného programu.

Proto bylo přistoupeno k vývoji interpretu ve *Smalltalk*u, na jeho volně šiřitelné implementaci *Squeak*. Ta (jak bylo zmíněno výše 1.1.4), umožňuje překlad kódu *Smalltalk*u do *C*. Takže většina vývoje, testování a ladění může probíhat v přívětivém integrovaném vývojovém prostředí *Squek*u se *Slang*em (viz 2.1.4) jako programovacím jazykem. Při správném návrhu volání funkcí a alokaci objektů lze udržet správu paměti zcela v naší režiji a vyhnout se tak nekontrolovatelnému zahlcení paměti nebo její fragmentaci při využití dynamické alokace.

Níže jsou popsány podrobnosti tohoto procesu.

Konvence v tomto textu: Pro označení náležitosti metody k třídě se budeme držet zavedené konvence ClassName » methodName.

Direktivy pro překladač do jazyka *C* budeme zapisovat následovně:

```
<var: #varName type: 'int const *'>
```

namísto zaslání zprávy objektu **self**.

2.1 Překlad *Smalltalk*u do *C*

Squeak umožňuje překlad kódu takzvaných *plug-in*ů virtuálního stroje do jazyka *C*. Tyto *plug-in*y mohou být využity ke zrychlení některých náročných operací, jejichž interpretace ve *Squeak*u by trvala neúnosně dlouho. Toto však není ten nejpodstatnější důvod k využití *plug-in*ů. Těmi nejdůležitějšími jsou tyto:

přístup k hardwaru: „vývojáři *Squeak*u nemohli předpovídat každou možnou potřebu pro primitivu systému. Někdy je požadavek přístupu k nízkoúrovňovým ovladačům či k lokální funkcionalitě operačního systému esenciální pro provedení určitých operací.“ [15]

přístup ke knihovnám: Mnoho dobře udržovaných a široce přenositelných knihoven poskytuje funkcionalitu potřebnou ve *Squeak*u. V tomto případě by její přepis do *Smalltalk*u byl zbytečnou prací, když ji lze jednoduše zpřístupnit pomocí rozšíření.

rychlost: Snad ani není třeba jmenovat. V kritických aplikacích a u náročných výpočtů by režije interpretace každého zaslání zprávy v nativním *Squeak*u byla neúnosná.

My tohoto prostředku využijeme k vytvoření *plug-in*ů — *C* modulů, které budou následně součástí našeho programu simulátoru.

2.1.1 Softwarové požadavky

Pro proces popsaný níže je kromě virtuálního stroje *Squeak*u a image potřeba i následující rozšíření:

1. `FFI-Pools` [9] ve verzi `FFI-Pools-eem.3`. Závislost pro balíček `VMMaker`.
2. `SharedPool-Speech` [7] ve verzi `SharedPool-Speech-dtl.2`. Závislost pro balíček `VMMaker`.
3. `VMMaker` [10] ve verzi `VMMaker-dtl.319`. Je překladač jazyka *slang* do *C*. Podporuje všechny platformy, kde *Squeak* běží. Má grafické rozhraní, lze jej však použít i přímo z *workspace*.
4. `SlangBrowser` [6] ve verzi `SlangBrowser-dtl.12`. Není naprosto nezbytný. Výrazně však usnadní vývoj. Jedná se interaktivní prohlížeč zdrojových kódu v *C* pro interpreter a *plug-in*y.
5. `Toothpick` [8] ve verzi `PaulDeBruicker.10`. Je konfigurovatelný logovací mechanismus pro *Smalltalk*.

Jednotlivé závislosti jsou uvedeny v pořadí, v jakém by měly být instalovány. Bohužel ne všechna zmíněná rozšíření v aktuálních verzích jsou k nalezení v repositářích *Squeak*u. Proto doporučuji následovat uvedené reference a stáhnout aktuální verze. Ty lze jednoduše nainstalovat například podle sekce „6.9 The File List Browser“ v knize Squeak By Example [3].

Použité verze *Squeak*u a jeho image uvádí tabulka 2.1.

Název software	Verze
Squeak virtual machine	`4.10.2.2614` (unix)
Squeak image	`Squeak4.4#12335`

Tabulka 2.1: Verze použitého *Squeak*u

2.1.2 Anatomie rozšíření

Vygenerovaný a zkompilovaný *plug-in* je knihovna rutin v jazyce stroje s minimálně jednou bezparametrovou (primitivní) funkcí. Každá vracející 32-bitové číslo jako výsledek. Dále obsahuje jednu speciální funkci setInterpreter s jedním parametrem představujícím ukazatel na *proxy* interpretu. Tato funkce je volána jako první funkce *plug-in*u se zmíněnou *proxy*, obsahující ukazatele na různé interní datové struktury virtuálního stroje a podmnožinu jeho interních subrutin. Přes ní probíhá obousměrná komunikace mezi *plug-in*em a virtuálním strojem.

Každý objekt *Squeak*u je reprezentován jako 32-bitový integer, který budeme dále nazývat jako *oop*. Před voláním primitivní funkce jsou *oop*y na zásobník uloženy jako její parametry. Ten je přístupný z volané primitivy skrz *proxy*. Primitiva by měla odstranit ze zásobníku všechny parametry a na jejich místo vložit jeden *oop* jako výsledek.

Detailní rozbor rozšíření je popsán Andrew C. Greenbergem v [15].

2.1.3 Rozhraní *plug-in*u

My *plug-in* modul budeme používat z naší aplikace simulátoru *PNVM*. Abychom jej mohli použít, je nutné se seznámit s jeho rozhraním. Je potřeba si uvědomit, že rozhraní je navrženo pro interakci s virtuálním strojem *Squeak*u a ne pro obecné použití v libovolné aplikaci. Tomu odpovídají níže uvedené struktury a funkce, které by při použití v našem interpretu způsobili značnou režiji během vzájemné interakce modulů mezi sebou. Díky textovému post-procesingu je však možné tyto struktury obejít. Na něj se však zaměříme později 2.3.

Naším rozhraním je objekt *proxy*, které umožňuje přistupovat, k nám nepotřebnému, virtuálnímu stroji. Emuluje zásobník pro volání funkcí, slouží k dynamické typové kontrole, přetypování a přístupu k atributům objektů. Slouží také jako notifikátor chyb.

proxy jako zásobník

Jak již bylo řečeno, jedná se o zásobník uchovávající parametry volaných funkcí a sloužící k vracení jejich výsledků. K tomuto účelu slouží několik metod:

- InterpreterProxy » push:
  uloží hodnotu na zásobník. Má několik přetížených variant s dodatečnými kontrolami vkládané hodnoty, např.: pushBool:, pushFloat:, pushInteger:.
- InterpreterProxy » pop:
  odstraní a vrátí hodnotu z vrcholu zásobníku. Stejně jako push: má i tato metoda několik přetížených variant.
- InterpreterProxy » pop: n thenPush: value
  odstraní n hodnot z vrcholu zásobníku a vloží na něj value.

- InterpreterProxy » stackValue: offset
  vrátí hodnotu z pozice offset ze zásobníku.

proxy jako notifikátor chyb

Rozšíření virtuálního stroje mohou přes *proxy* hlásit chyby o selhání vykonávání operace. Slouží k tomu příznak successFlag, která je nastavována zprávou

```
InterpreterProxy » success: aBoolean
```

Přečíst ji lze zasláním zpráv successful nebo failed.

Toto hlášení chyb je využíváno napříč celým modulem. Tzn. i statickými funkcemi. Díky tomu návratové hodnoty metod mohou mít datový typ odpovídající této hodnotě. To, zda funkce selhala nebo ne, lze zkontrolovat ověřením tohoto příznaku.

proxy jako rozhraní k datovým typům

Protože datové struktury jsou ve *smalltalk*u a *C* jinak uloženy a jinak se přistupuje k jejich položkám, je nutno mít k dispozici jednotné rozhraní, aby nebylo nutné psát specifický kód pro obě varianty vykonávání. Tímto rozhraním je opět objekt *proxy*.

*Plug-in*y tak mohou přistopit k libovolnému prvku libovolného objektu a volat jeho metody. Díky typové kontrole, kterou *proxy* také poskytuje, je možno ověřit, zda opravdu přistupujeme k prvku očekávaného typu.

My se však spokojíme pouze s přístupem k prvkům indexovatelných objektů jako jsou: ByteArray, String, Array. Přístup k prvkům instancí těchto objektů s pomocí zpráv arr at: index a arr at: index put: value je sice překladačem do jazyka *C* přeložen správně. Je tu však problém indexace. V *C* jsou veškerá číselně indexovatelná pole indexována od 0. Zatímco *Smalltalk* indexuje od 1. Pokud chceme přistupovat k prvkům sekvencí v obou variantách vykonávání stejně (se stejnými hodnotami indexů), musíme si pro tyto sekvence vytvořit obal.

Proxy nám takovýto obal vytvoří voláním její metody:

```
wrapper := interpreterProxy firstIndexableField: arr.
```

na libovolnou sekvenci. Ve *Smalltalk*u je tento wrapper instancí CArrayAccessor, která zprávy pro přístup k prvku jí zaslané přeposílá původnímu objektu s indexem navýšeným o 1. V *C* je toto volání přeloženo na pouhé přiřazení ukazatele ukazateli. Dále nám tento obal umožní ve *Smalltalk*u používat omezené množství aritmetických operací nad ukazateli, tak jako v *C*.

2.1.4 *Slang*

Vzhledem k ne-příliš velké rozšířenosti samotného jazyka *Smalltalk*, natož jeho podmnožině *Slang*, je vhodné si zde shrnout jeho základní výrazové prostředky. K nim budou uvedeny odpovídající vygenerované bloky kódu v *C*.

Výrazy — literály, jako jsou čísla, znaky, symboly a řetězce, jsou překládány do odpovídajících literálů v *C*. Pole nejsou povoleny. Přiřazení jsou přeloženy 1:1.

Zprávy objektům jsou přeloženy na *C*-volání funkce. Níže jsou uvedeny zprávy unární, binární a zprávy s klíči.

Slang

```
anObject methodCall.
"binary message"
anObject methodWith: argument.
"keywords message"
anObject methodWith: arg1 and: arg2.
```

C

```
methodCall(anObject);
// binary message
methodWith(anObject, argument);
// keywords message
methodWithand(anObject, arg1, arg2);
```

Zprávy s klíči vytvoří název funkce konkatenací jména metody.

Speciální zprávy Vyjímku z těchto pravidel tvoří zprávy zaslané objektu `self` nebo `interpreterProxy`. Zpráva objektu `self` je přeložena jako volání funkce bez samotného objektu `self`, protože z pohledu *plug-in*u zastupuje tento ukazatel překládaný modul. Zatímco `interpreterProxy` je ukazatel na speciální objekt s ukazateli na funkce virtuálního stroje, které lze v *C* jednoduše zavolat. Tabulka níže ukazuje jak.

Slang

```
self methodWith: arg.
interpreterProxy methodWith: arg.
```

C

```
methodWith(arg);
interpreterProxy->methodWith(arg);
```

Naštěstí smalltalkovské zprávy odpovídající základním *C* operátorům jako &, |, and:, +, apod. překládány do očekávaných *C* výrazů. Zpráva foo at: exp, což je *smalltalk*ovský přístup k prvku indexovaného objektu je přeložena jako: foo[exp]. Podobně je tomu u přiřazení prvku do indexovaného objektu — foo at: exp1 put: exp2.

Řídicí struktury *Slang* poskytuje také základní kontrolní struktury jako jsou:

```
[stmtList] whileTrue: [stmtList2]
exp1 to: exp2 do: [stmtList]
exp1 ifTrue: [stmtList] ifFalse: [stmtList]
```

Které jsou přímo mapovány na struktury v *C*, které bychom očekávali.

Zápis přímého kódu pro oba režimy vykonávání V některých případech je vhodné nebo dokonce nutné zapsat příkaz nebo výraz pouze pro *Smalltalk* resp. *C*, který nebude vykonán v *C* resp. *Smalltalk*u. Příkladem takové potřeby je například deklarace globálních proměnných modulu, které jsou ve *Smalltalk*u reprezentovány instančními proměnnými, zatímco v *C* modulu se musejí definovat jako statické proměnné. Ty se uvádějí v třídní metodě declareCVarsIn: codeGenerator *plug-in*u.

Squeak poskytuje pár funkcí pro tyto účely:

- Object » cCode: 'memcpy(dest, src, cnt)'.
  Bere jako argument řetězec s úsekem kódu v jazyce *C*, který má být vygenerován na současné pozici v kódu. Při interpretaci *Squeak*em je tento příkaz ignorován.
- Object » cCode: 'strlen(strptr)' inSmalltalk: [strptr size].
  Je variantou předchozího, kdy navíc zadáváme blok *smalltalk*ovského kódu, který bude proveden pouze při interpretaci *Squeak*em.

Listing 2.1: Příklad deklarace globálních proměnných modulu

```
declareCVarsIn: cg
    cg addHeaderFile: '"pnvm_types.h"'.
    cg addHeaderFile: '"pnvm_utils.h"'.
    cg var: 'netTemplateCode'
            declareC: 'char netTemplateCode[NetTemplateCodeSize]'.
    cg var: 'netTemplateTable'
            declareC: 'PNVMTemplate * netTemplateTable[MaxNetTemplates]'.
    cg var: 'calendarHead' type: 'PNVMEvent *'.
    cg var: 'calendarTail' type: 'PNVMEvent *'.
```

Příkladem jejich využití může být metoda třídy PNVMObjectMemory ve výpisu 2.1. Kde `MaxNetTemplates` je konstanta definovaná jako proměnná třídy `PNVMConstants` a definovaná v třídní funkci:

```
PNVMConstants » initialize
```

Zde nejedná se o jazyk *Slang*, neboť tato metoda je volána pouze překladačem (nejedná se o exportovaný kód).

2.1.5 Typová kontrola

Narozdíl od *C* je *Smalltalk* dynamicky typovaný. Překladač do *C* informace o typu proto není schopen rozpoznat a musíme ji explicitně dodat. Implicitně jsou veškeré proměnné a argumenty deklarovány jako 32 bitový celočíselný typ sqInt. Všechny ostatní typy je nutné specifikovat následujícími direktivami definovanými třídou CCodeGenerator:

- **self** var: #variableName type: 'char const *'.
  Specifikuje typ proměnné nebo argumentu metody. Toto je preferovaná metoda, která nám však neumožní specifikovat pole konstantní délky.
- **self** var: #variableName declareC: 'int variableName[20]'.
  Detailní specifikace typu. Zde máme plnou kontrolu nad zápisem. Musíme však uvést název proměnné dvakrát.
- **self** returnTypeC: 'char *'.
  Specifikuje typ návratové hodnoty.

V kódu, kdy naše implementovaná metoda náleží třídě dědící z InterpreterPlugin, se tyto direktivy zasílají objektu **self**. My však využijeme zjednodušeného zápisu ve tvaru:

```
<var: #varName type: 'int const *'>
```

pro všechny popsané direktivy.

Platí, že všechny tyto direktivy jsou do *C* kódu zapsány pouze jako deklarace typů a tedy negenerují žádnou dynamickou typovou kontrolu a režiji navíc.

Dalšími užitečnými direktivami pro překlad jsou zprávy přidané třídě Object balíčkem `VMMaker`.

- Object » doNotGenerate zamezí překlad zprávy do *C*.

- Object » export: aBoolean přidá funkci k rozhraní modulu. V generovaném kódu se to projeví odstraněním resp. přidáním klíčového slova **static** pro hodnotu parametru **true** resp. **false**.
- Object » inline: aBoolean povolí či zakáže inlinování funkce.
- Object » cCoerce: value to: cType přetypuje hodnotu na daný typ. Například:

  ```
  value := (self cCoerce: number to: 'unsigned') >> shiftBy.
  ```

  Je přeložen na:

  ```
  1 value = ((unsigned) number) >> shiftBy.
  ```

- Object » cPreprocessorDirective: str vloží na aktuální pozici ve generovaném kódu direktivu preprocesoru. To lze s úspěchem použíť pro deklaraci maker zpřehledňujících, případně zeštíhlujících kód.
- Object » proprocessorExpression: str je obdobou předchozího. Řetězec je navíc uvozen znakem '#'.

Lze je nalézt v kategorii `*VMMaker-translation support`. Pokud není uvedeno jinak, použité direktivy ovlivní pouze generovaný kód a nijak nezmění interpretaci kódu *Squeak*em.

2.1.6 Primitivní a statické funkce

Jak již bylo řečeno výše, primitivní funkce jsou rozhraním modulu a komunikuje se s nimi prostřednictvím objektu *proxy*. Manipulace s ní však představuje pro modul nezanedbatelnou zátěž, obzvlášť pro embedded aplikaci. Pro ne-primitivní funkce (dále je budeme nazývat jako statické) již však není potřeba. Jejich rozhraní je čistě *smalltalk*ovské s přidanými typovými direktivami. Jejich přeložením do *C* kódu vzniknou statické funkce se stejnými parametry jako jejich *smalltalk*ovská předloha.

To, zda funkce bude označena jako primitivní nebo ne, rozlišuje direktiva

```
<export: aBoolean>
```

Je-li aBoolean = **true**, bude funkce primitivní, jinak statická.

2.1.7 Strukturování kódu

Mezi největší výhody *Smalltalk*u patří beze sporu jeho čistá objektová orientace. „Vše je objekt“ [3], kde běh aplikace je interakcí objektů mezi sebou. Kde úpravou tříd, z kterých jsou tyto objekty vytvářeny, měníme jejich chování.

V případě *plug-in*ů virtuálního stroje je samotný *plug-in* objektem, který je po přeložení reprezentován knihovnou. Tento objekt může komunikovat s jinými zasíláním zpráv. Každý *plug-in* dědí z třídy InterpreterPlugin, kde metody instance určují chování *plug-in*u a jsou předmětem generování kódu.

Modularita

V našem případě však *plug-in* nekomunikuje s žádnými objekty. Manipuluje pouze s primitivními datovými typy a ukazateli na ně. Pokud bychom chtěli využít zasílání zpráv, museli bychom vytvořit více modulů, tj. více *plug-in*ů. Ty by mezi sebou komunikovaly skrz

objekt InterpreterProxy. To by ovšem představovalo značnou režiji z důvodu využívání dvou oddělených zásobníků.

První by byl využíván pro současný kontext procesoru a uchovával by návratové adresy do kódu při volání primitivních a statických funkcí. Dále by sloužil k alokaci místa pro lokální proměnné.

Druhý by byl součástí InterpreterProxy a uchovával by parametry primitivních funkcí a jejich návratové adresy. Přístup k argumentům by však byl v tomto případě krkolomný a kód matoucí. Navíc v *C* je tento způsob předávání argumentů velmi neefektivní v porovnání s optimalizovaným přístupem k lokálnímu zásobníku.

V situaci, kdy máme dva zásobníky, musíme se zabývat otázkou alokace jejich místa. Protože však se chceme vyhnout dynamické alokaci paměti, museli bychom omezit maximální velikost zásobníku. Jeho velikost, tak aby vyhovovala pro všechny případy, se však velmi špatně odhaduje. V ideálním případě by zásobník byl pouze jeden a tak bychom se vyhnuly problémům s jejich možnou kolizí nebo zaplněním.

Využití více *plug-in*ů s *proxy* jako prostředníkem proto nepřipadá v úvahu.

Nicméně nic nám nebrání více *plug-in*ů vytvořit a získat tak výhodu modularity. Je možno takto rozdělit program do více nezávislých částí s jasně daným rozhraním a skrytou implementací. Je však třeba brát v úvahu, že objekt **self** třídy reprezentující *plug-in* v generovaném kódu neexistuje. V generovaném kódu chybí jakékoliv rozdělení prostoru jmen. Nemáme proto k disposici žádný prostředek, jak specifikovat, který plugin má danou zprávu zpracovat. Je tedy důležité volit unikátní názvy zpráv pro ne-statické funkce. Zde nám naštěstí práci usnadní linker zkompilovaných modulů, který vícenásobnou definici funkce odhalí.

Použité prostředky

Co se týče strukturování kódu, máme tedy následující možnosti:

- Modularizace vytvořením stromu potomků z bázové třídy `InterpreterPlugin`.
- Nevětvená hierarchická dědičnost tříd, z nichž bázovou třídou je `InterpreterPlugin`.
- Organizace metod do kategorií — je pouze sdružení metod v rámci jedné třídy pod název, který dává představu o tom, čím se metody zabývají. Tato kategorizace se nijak neprojeví ve vygenerovaném modulu. Slouží pouze po přehlednost v rámci vývojového prostředí *Squeak*.

Vzhledem k pozdějšímu odhalení možnosti modularizace bez nutnosti využití *proxy* objektu jako prostředníka, nebylo v návrhu a pozdější implementaci s tímto prostředkem počítáno. Implementace tedy využívá pouze poslední dva zmíněné body. Pro naše účely jsou však tyto prostředky naprosto dostatečné.

2.1.8 Shrnutí poskytovaných výrazových prostředků

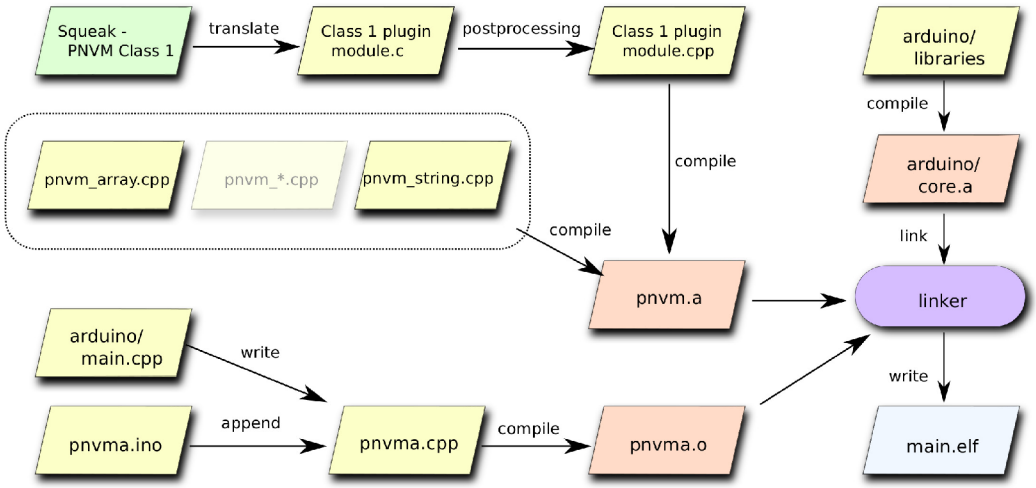
Jazyk *Slang* 2.1.4 nám umožňuje *smalltalk*ovský zápis kódu *C* v prostředí *Squeak*, kde jej lze spouštět a ladit bez nutnosti předkladu. Poskytuje nám veškeré možnosti jazyka *C*. Na druhou stranu se nedá říci, že je zápis jednodušší, než by byl v čistém *C*. Vzhledem k občasné potřebě zápisu různého kódu pro interpretovaný *Smalltalk* a překládané *C*, je místy zápis

dvojnásobně delší. Zde navíc musíme deklarovat argumenty funkcí (viz 2.1.5), lokálních i globálních proměnných zvlášť pro generátor *C*. Přesto však výhody tohoto přístupu pro snadné úpravy a testovaní jasně převažují.

2.2 Kompilace pro *Arduino*

Zatím byl popsán převod kódu ze *Squeak*u do *C* modulu s určitým rozhraním, které je společné jak pro *Smalltalk* tak i hlavní modul simulátoru. Ten budeme nadále nazývat *PNVM* (*Petri Net Virtual Machine*). Aplikaci pro *Arduino* nazvěme *PNVMA* a její hlavní modul jako `pnvma`.

Obrázek 2.1 názorně ukazuje všechny moduly, z kterých se výsledný binární soubor skládá. Těmi jsou:



Obrázek 2.1: Proces kompilace

`pnvma.o` jenž je výsledkem kompilace modulu `pnvma.cpp`

`pnvm.a` je statická knihovna s moduly implementujícími datové typy a různé doplňkové funkce. Zde je přítomen i zkompilovaný, vygenerovaný modul *plug-in*u.

`arduino.a` je statická knihovna se základními moduly platformy *Arduino* šířenými s touto platformou. Poskytuje abstraktní vrstvu nad hardwarem.

Modul `pnvma.cpp` vznikne konkatenací modulu `pnvma.ino` a `arduino/main.cpp`. To je běžný postup pro projekty této platformy. `arduino/pnvma.cpp` obsahuje pouze jednoduchou smyčku volající v patřičných okamžicích funkce modulu `main.ino`. Jehož rozhraním je typicky jen:

- funkce **void** setup()
  Je volána hned restartu hardware a slouží k jeho inicializaci.
- funkce **void** loop()
  Definuje chování aplikace. Je volána ve smyčce spolu s funkcí pro zpracování událostí (jako je příchod bytu po sériové lince).

2.3 Post-procesing *plug-in* modulu

Protože je generátor kódu *Smalltalk* do *C* primárně určen ke kompilaci rozšíření virtuálního stroje, je vygenerovaný soubor prosycen deklaracemi metod, proměnných, příkazy pro preprocesor a dalšími náležitostmi potřebnými ke správnému chodu a slinkování s kódem pro virtuální stroj. Většina z těchto přídavků však pro naše účely není relevantní. My naopak potřebujeme modulu přidat hlavičkové soubory, které nám umožní bezproblémové provázání s moduly pro *Arduino*. Post-procesing tedy pouze zpřehlední vygenerovaný modul odstraněním zbytečností. Nesnaží se příliš zasahovat do samotného kódu. Snaha o jeho správnost je mířena do prostředí *Squeak*.

Ve vygenerovaném kódu je třeba provést tyto kroky:

1. odstranění hlavičkou souborů
2. odstranění maker pro preprocesor
3. odstranění statických proměnných a automaticky vegenerovaných funkcí
4. odstranění ladících proměnných tříd
5. odstranění typových kontrol
6. přepis deklarací typů
7. přepis zasílání zpráv datovým typům na volání funkce
8. export primitivních funkcí do hlavičkového souboru
9. přepis logovacích příkazů
10. „zkrášlení“ kódu

Na vybrané kroky se zaměříme podrovněji níže. Velká část post-procesingu se věnuje odstranění nadbytečných konstrukcí. Co se týče automaticky vygenerovanýčh statických proměnných a funkcí, jako jsou např:

```
static char ___buildInfo[];
static const char *moduleName;
EXPORT(const char*) getModuleName(void);
```

je tato operace zcela oprávněná z hlediska kompilace pro embedded architektury neoplývající nadbytkem paměti. Navíc v rámci simulátoru by tyto informace nenašli uplatnění. Makra sice nepřispívají k velikosti zkompilovaného modulu, ale v tomto případě komplikují kód, který se tak stává méně čitelným. Proto makra typu `#pragma export on` jsou rovněž odstraněny.

Generovaný kód dále používá náhradní jména pro základní datové typy. `sqInt` pro **int**, `usqInt` pro **unsigned**. Ty jsou přepsány zpět na typy, které reprezentují.

V neposlední řadě hlavičkové soubory s rozhraním virtuálního stroje a nastavením pro konkrétní platformu jsou také nadbytečné.

Zkrášlení kódu je provedeno utilitou `indent`, která pouze unifikuje vzhled vygenerovaného kódu tak, aby byl čitelnější. Tento krok nijak neovlivní výslednou funkci modulu.

2.3.1 Odstranění typových kontrol

Typová kontrola je užitečná běh simulátoru v prostředí*Squeak*, kde tímto myslíme dynamickou typovou kontrolu s použitím zpráv Object » assert: a vnořenými Object » isKindOf:. To je ovšem přeloženo do výsledného *plug-in*u v *C* jako volání funkce *proxy* pro kontrolu typu. My se však v kompilovaném programu spoléháme pouze na typovou kontrolu statickou. Proto příkazy assert(...) s výskytem interpreterProxy jsou post-procesorem zakomentovány.

2.3.2 Přepis zasílání zpráv datovým typům

Jak bude zmíněno níže (3.1), využíváme datových abstrakcí pro práci s objekty dynamicky alokovanými a objekty, které s nimi manipulují. Ty jsou ve *Smalltalk*u reprezentovány samostatnými třídami, které ale nejsou ve výsledku nijak generovány spolu s kódem *plug-in*u. V generovaném kódu na ně nalezneme pouze reference:

Slang

```
token := PNVMToken pnvmTokenNew.
```

C

```
1 token = pnvmTokenNew(PNVMToken);
```

Je mnoho způsobů, jak tuto vlastnost generátoru ošetřit. Funkce s prvním argumentem tohoto typu by mohla být přepsána na volání statické funkce třídy `PNVMToken`, případně prostoru jmen:

```
1 token := PNVMToken::pnvmTokenNew().
```

Zde však bylo zvoleno prostější řešení. První argument je zcela odstraněn:

```
1 token := pnvmTokenNew().
```

Funkce tříd už jsou sami o sobě prefixovány jménem datového typu, kterému náleží, a proto není nutné tuto informaci ve vygenerovaném kódu nadále duplikovat. `pnvmTokenNew` je tedy globální funkcí pracující nad typem `PNVMToken`. Takto jsou ošetřena všechna zaslání zpráv třídám s názvem `PNVM[a-zA-Z]+`.

2.3.3 Export primitivních funkcí

Primitivní funkce (viz 5.2 níže) tvořící rozhraní modulu je nutno deklarovat v hlavičkovém souboru pro možnost deklarace funkcí pro ostatní moduly, z nichž je program linkován. Tyto funkce jsou uvozeny makrem `EXPORT(returnType)`, jenž nám velmi ulehčí hledání těchto deklarací v generovaném kódu. Všechny takovéto deklarace pak stačí pouze zapsat jak do generovaného souboru, tak i do odpovídajícího hlavičkového souboru.

2.3.4 Přepis logovacích příkazů

Logovací zprávy jsou neocenitelnou pomůckou při ladění složitějších programů. Ani *PNVM* simulátor se bez nich neodejde. Ve výsledné, zkompilované aplikaci chceme mít logování pod kontrolou podobně jako je tomu ve *Squeak*u. Všechny příkazy přítomné v generovaném *Slang* kódu chceme mít rovněž přítomné v kompilované aplikaci. Bohužel použité logovací mechanismy obou prostředí jsou natolik odlišné, že jednoduché zkopírování logovacího příkazu nestačí.

Zprávy navíc mohou mít také argumenty, které generátor do *C* nedokáže zpracovat a modul tak bez textového procesingu není kompilovatelný.

Abychom mohli mapovat logovací příkazy ze *Squeak*u do *C*, je nutné se blíže seznámit s použitými mechanismy.

Logování ve *Squeak*u

Ve *Squeak*u byl zvolen mechanismus *Toothpick* pro logování zpráv poskytovaný stejnojmenným balíčkem. Nebudeme zde zacházet do detailů, jak pracovat s tímto frameworkem, ani jak jej konfigurovat. Podrobné informace lze nalézt na [8].

Logovací událost se vytvoří interpretací kódu:

```
LoggingEvent
    category: #categoryName
    level: #info
    message: 'message string with argument: ',obj.
```

Kategorie události i název jsou udávány symbolickými jmény předem deklarovanými v nastavení frameworku. Na jejich základě lze rozhodovat, zda bude zpráva vypsána, kam a v jakém formátu.

Všimněme si, že argumenty zprávy jsou konkatovány operátorem (,). Je to posloupnost objektů, z nichž první je typu `String`. Zbytek posloupnosti je na tento typ objektu převeden postupnou aplikací (,), jejímž výsledkem je spojení dvou řetězců do jednoho. Výsledný řetězec je předán zprávě LoggingEvent » category:level:message:.

Logování v *C*

Logovací zprávy v embedded aplikaci pro *Arduino* by byly přítěží. Příliš by navýšily velikost kódu a zcela zahltily sériovou linku, pokud by byly použity. O výrazném zpomalení interpretu netřeba hovořit. Logování má proto smysl pouze pro aplikaci `pnvmx86`, kde není třeba se omezovat v nastavení logovacího frameworku a formát zpráv upravit tak, aby odpovídal své *Squeak*ovské předloze. Logovací framework byl zvolen tak, aby zvládl vše, co jeho protějšek *Toothpick* a byl přitom snadno konfigurovatelný.

Stal se jím framework *log4cxx* pro *C++*. Opět zde nebudeme zabýhat do detailního popisu použití ani konfigurace. Ten, a další podrobnosti, lze nalézt na online zdroji [14].

Příkaz k logování zprávy se zapisuje následovně:

```
LOG4CXX_INFO(logger, "message string with argument: " << obj)
```

`LOG4CXX_INFO` je makro, jehož část za `'_'` udává důležitost zprávy a odpovídá druhému argumentu logovací zprávy *Squeak*u. `logger` je ukazatel na instanci třídy log4cxx::Logger určující výstupní formát, parametry výstupu a minimální prioritu zprávy.

Přepis

Přepis volání logovací zprávy s prvními dvěma argumenty je primitivní. První argument určí `logger`, který má vytvořenou instanci pro všechny kategorie použité ve *Squeak*u. Tedy například kategorie `pnvmMemory` se přepíše na memlog a předá se jako první argument makru `LOG4CXX_<LEVEL>`, kde část `<LEVEL>` je nahrazena druhým argumentem zprávy s velkými písmeny. Poslední argument vyžaduje náročnější analýzu. Vygenerovaný *C* kód převede infixovou notaci aplikace (,) operátoru na prefixovou a argumenty uzavře do závorek s čárkou jako oddělovačem. Takže výše (2.3.4) uvedený příkaz se přepíše na:

```
categorylevelmessage(LoggingEvent, categoryName, info,
  ,("message string with argument: ", obj));
```

Textový post-procesor tedy musí prefixovou notaci převézt zpět na infixovou a operátor (,) nahradit operátorem («). Tím je zpracování logovací zprávy dokončeno.

2.3.5 Interface k *plug-inu*

Jak již bylo zmíněno výše (2.1.6), má vygenerovaný modul rozhraní společné pro *Smalltalk* i *C*. To je složeno z funkcí zvaných „primitivy“. Ty lze volat s pomocí *proxy* objektu, kterému jsou předány parametry pro volanou primitivu do jeho zásobníku. Volaná funkce je po skončení odstraní a zanechá v ní návratovou hodnotu. Podrobnosti byly uvedeny výše (viz 2.1.3).

My tento přístup zcela obejdeme. Definici

```
struct VirtualMachine* interpreterProxy;
```

smažeme, exportujeme primitivní funkce do hlavičkového souboru a ten se tak stává úplným rozhraním vygenerovaného *plug-in*u. Podrobnému popisu rozhraní se věnuje sekce 5.2.

Kapitola 3

Výrazové prostředky

Cílem práce je možnost interpretovat referencované sítě na vestavěných systémech. Jedním z hlavním důvodů je snadnost popisu požadované funkce v porovnání s psaním zdrojových kódu v *C/C++*. Z tohoto pohledu hrají výrazové prostředky, kterými program popisujeme, tu nejdůležitější roli. Kdyby se expresivnost interpretovaného jazyka s popisem sítě blížila assembleru, zřejmě by nemělo smysl do implementace vůbec pouštět.

Jelikož si tato práce neklade za cíl pokrýt a implementovat všechny vlastnosti *RefNets*, omezíme se pouze na základní datové typy a operace nad nimi. Tato kapitola zahrne výčet implementovaných datových typů, jejich vlastnosti, možné operace a jazyk *PNAL* sloužící k popisu referencovaných sítí.

3.1 Datové typy

Těmi jsou myšleny uživatelské datové typy, které uživatel může vytvářet, mazat, referencovat, a provádět nad nimi definované operace. Rozlišujeme u nich několik vlastností:

referencovatelnost zda je možno se na objekt daného typu odkazovat z více míst.

modifikovatelnost zda je možno hodnotu proměnné měnit. Všichni referenti daného objektu se budou po jeho modifikaci odkazovat na upravenou hodnotu.

indexovatelnost zda objekty daného typu mohou obsahovat objekty jiné. Takové nazvěme indexovatelné, neboť jednotlivé položky lze získat pomocí indexu.

Simulátor *PNVM* podporuje několik základních typů:

Integer Reprezentuje celá čísla se znaménkem. Objekty tohoto typu nejsou modifikovatelné, referencovatelné ani indexovatelné.

String Reprezentuje řetězce znaků s hodnotami v rozmezí $\langle 0 - 255 \rangle$. Je obdobou datového typu `std::string` v jazyce *C++*. Typ je referencovatelný, modifikovatelný a indexovatelný.

Tuple Reprezentuje n-tici prvků. Může obsahovat libovolný z ostatních datových typů. Je referencovatelný a indexovatelný. Využijeme zde syntaktický zápis hodnot používaný v *Renew* (viz 1.1.3), je jím sekvence *token*ů uzavřená do hranatých závorek: `[]`.

Array Reprezentuje pole prvků, které lze libovolně rozšiřovat, zmenšovat a modifikovat. Položkami jsou, stejně jako v případě *Tuple*, objekty ostatních uživatelských datových typů. Referencovatelný, modifikovatelný a indexovatelný. Opět se budeme držet zápisu hodnot tohoto typu podle *Renew*. Položky jsou uzavřené složenými závorkami: {}.

NetInst Reprezentuje instance sítě. Síť se skládá z N míst obsahujících dvojice $(num, value)$, kde první položka udává výskyt hodnoty *value* v místě; *value* je hodnota libovolné uživatelského datového typu. N je určeno kódem šablony, z které je instance vytvořena. Je referencovatelný, modifikovatelný a indexovatelný.

Tabulka 3.1 obsahuje limity těchto datových typů v závislosti na cílové platformě.

Datový typ	Arch	Min	Max	Max položek
Integer	8bit	-2^{14}	$2^{14}-1$	
Integer	32bit	-2^{30}	$2^{14}-1$	
Integer	64bit	-2^{62}	$2^{62}-1$	
String	8bit			2^{10}
String	32bit			2^{32}
String	64bit			2^{64}
Tuple	8bit			2^{10}
Tuple	32bit			2^{32}
Tuple	64bit			2^{32}
Array	8bit			2^{10}
Array	32bit			2^{32}
Array	64bit			2^{32}
NetInst	8bit			2^{10}
NetInst	32bit			2^{32}
NetInst	64bit			2^{32}

Tabulka 3.1: Přehled omezení uživatelských datových typů v závislosti na platformě

Objekt jakéhokoliv z výše uvedených uživatelských datových typů budeme nadále označovat jako *token*. Budeme-li později hovořit o hodnotě typu *Bool*, bude tím myšlen *token* typu *Integer* s hodnotou 1 pro `true` a 0 pro `false`.

Konverze na *Bool* Ve výrazech nad *tokeny* se často vyhodnocuje, zda je *token* „pravdivý“ nebo není. Tato informace může být použita jak v samotných výrazech, tak k vyhodnocení splnitelnosti přechodu. Je tedy důležité umět tuto informaci odvodit pro *token* libovolného typu a hodnoty.

„Pravdivost“ [1] *tokenu* lze určit podle tabulky níže. Splňuje-li hodnota alespoň jednu z podmínek pro její odpovídající typ, pak je „pravdivá“. Jinak je „nepravdivá“. Na hodnotu *tokenu* se v podmínkách odkazujeme symbolem *value*.

3.1.1 Operátory

Chceme-li s nad *tokeny* provádět operace, potřebujeme k tomu operátory. Ty lze rozdělit podle arity, návratového typu nebo podle typu operandů. Souhrn všech operátorů lze nalézt v tabulkách 3.4 3.5 a 3.6.

[1] to, zda *token* odpovídá po konverzi na *Bool* hodnotě 1

Typ	Podmínka
NetInst	true
Integer	$value \neq 0$
String	$value \neq$ ""
Tuple	$value \neq []$
Array	$value \neq \{\}$

Tabulka 3.2: Určení pravdivosti hodnoty *value tokenu* pro uživatelské datové typy

Není nutno uvádět prioritu operátorů, protože ta je dána závorkováním výrazů. Jejich zápis je obdobou zápisu programu v jazyku *lisp*. O výrazech však budeme pojednávat později (3.3.4).

Aritmetické operátory

Operují nad *tokeny* typu *Integer*. Jsou jimi: `+`, `-`, `*`, `/`, `%` mají aritu 2 a vracejí opět *Integer*. Jejich sémantika je shodná s operátory v jazyku *C*.

Logické operátory

Pracují nad *tokeny* libovolného typu. K vyhodnocování pradivosti platí pravidla uvedená v tabulce výše 3.2. Podporovanými binárními operátory jsou `&`, `|`,`^` s opět shodnou sémantikou jako v jazyce *C*. Dále je podporován unární operátor `!` negace. Návratovými hodnotami však pro binární operátory nejsou *Bool* proměnné, nýbrž samotné operandy:

Operátor	*op*1 pravda	*op*1 nepravda
&	*op*2	*op*1
|	*op*1	*op*2
^	¬*op*2	*op*2
!	0	1

Tabulka 3.3: Návratové hodnoty logických operátorů

Do této skupiny lze zařadit i testovací, 2-znakové operátory `ii`, `is`, `it`, `ia`, `in` a `iv`. Ty testují, zda je operand daného typu. Testovaný typ je určen druhým znakem: *Integer* pro `i`, *String* pro `s`, *Tuple* pro `t`, *Array* pro `a` a *NetInst* pro `n`. Poslední operátor `v` testuje, zda je proměnná vázaná.

Porovnávací operátory

Jsou opět definovány pro všechny datové typy s vyjímkou typu *NetInst* [2].

Výčet logických operátorů zahrnuje: `<`, `>`, `=`. Jejich arita je 2. Podmínkou však je, že oba operandy musejí být stejného typu. Jinak je celý výraz považován za nevalidní. Návratová hodnota je *Bool*. Pro datový typ *Integer* je význam stejný jako v jazyce *C*.

Pro ostatní, indexovatelné datové typy platí následující pravidla:

[2] na obecných platformách *x86* je výsledkem porovnání adres obou operandů, v prostředí *Squeak* je to porovnání identifikátorů bloků

$$a = b \iff size_a = size_b \quad \wedge \quad a_i = b_i, \quad 0 \leq i < size_a \tag{3.1}$$

$$\begin{aligned} a < b \iff & \exists n : 0 \leq n < min(size_a, size_b) \implies \\ & \Big\{ \quad \big(\forall i : 0 \leq i \leq n \Rightarrow a_i = b_i\big) \\ & \wedge \quad \big(a_{n+1} < b_{n+1} \vee (n+1 = size_a \wedge size_b > size_a)\big) \quad \Big\} \end{aligned} \tag{3.2}$$

$$\begin{aligned} a > b \iff & \exists n : 0 \leq n < min(size_a, size_b) \implies \\ & \Big\{ \quad \big(\forall i : 0 \leq i \leq n \Rightarrow a_i = b_i\big) \\ & \wedge \quad \big(a_{n+1} > b_{n+1} \vee (n+1 = size_a \wedge size_a > size_b)\big) \Big\} \end{aligned} \tag{3.3}$$

Kde a_i značí i-tý [3] prvek operandu a a $size_a$ počet prvků v operandu a.

Operátory nad kolekcemi

Kolekcí je myšlen jakýkoliv objekt indexovatelného typu. Jsou jimi nulární operátory `ns`, `na`, resp. `nt` sloužící k vytvoření nového objektu typu *String*, *Array*, resp. *Tuple*.

Dále sem patří unární operátory `h`, `t`, z nichž první vrací první prvek kolekce a druhý zbytek sekvence jako nový objekt stejného typu s velikostí o 1 menší. Podmínkou úspěšného provedení je neprázdnost kolekce.

Binárními operátory nad kolekcemi jsou:

`#` — vyber prvek. Prvním operandem je kolekce, druhým je *Integer* s indexem do ní. Výsledkem je prvek na pozici dané indexem.

`a` — přidej prvek. Prvním operandem je kolekce, druhým je libovolný token, který je zařazen na konec kolekce. Zde není vytvořen nový objekt, nýbrž kolekce je modifikována. Výsledkem je tedy modifikovaný první argument.

`,` — spoj řetězce. Typ obou operandů je *String*. Výsledkem je nový objekt typu *String* obsahující konkatenovaný obsah obou operandů.

Vstup/výstupní operátory

Ty pracují s piny zařízení nebo umožňují serializovat *token* a posílat na výstup. Následuje jejich výčet:

`p` — přečti hodnotu pinu. Operandem je *Integer* udávající index pinu číslovaného od nuly. Výstupem je *Integer* s hodnotami 1 pro nastavený pin, jinak 0.

`o` — zapiš hodnotu pinu. První operand je *Integer* udávající index pinu. Druhým je *token*, který je konvertován na *Bool*, jenž je zapsán jako jeho hodnota. Návratovou hodnotou je druhý operand.

`s` — pošli hodnotu na výstup. Tento operátor poskytuje alternativu k `:output(x)` přechodu sítě platformy. Hodnota operandu je serializována a okamžitě poslána na výstup [4]. Serializaci se věnuje sekce 3.5.

[3] položky kolekcí jsou indexovány od 0

[4] po sériové lince v případě *Arduina* nebo na standardní výstup v případě `pnvmx86`

Ostatní operátory

Název této skupiny je mírně zavádějící, neboť zde přítomné operátory svou důležitostí převyšují operátory výše uvedené a pro běh jsou zcela nezbytné. Umožňují nám měnit databázi šablon sítí, instanciovat je a vázat volné proměnné. Popíšeme je v pořadí od nejnižší arity:

nulární

`d` — vypiš současnou instanci. Umožňuje serializovat instanci, jejíž přechod je právě prováděn. Výsledkem je *token* typu *String*.

unární

`l` — nahraj šablonu sítě. Přijímá operand typu *String* s kódem šablony v jazyce *PNAL* (viz 3.2 níže). Výsledkem je *Bool* udávající, zda šablona byla úspěšně nahrána. V případě, že kód šablony není validní nebo databáze šablon je plná, je výšledkem 0.

`i` — nahraj instanci sítě. Operandem je *String* se serializovanou instancí sítě, jejíž šablona je přítomna v lokální databázi šablon sítí. Výsledkem je *token NetInst*. Je-li přechod úspěšný a instance je vložena do místa, bude v následujícím kroku simulátoru interpretována.

`c` — vytvoř instanci sítě. Operandem je jméno šablony, která má být instanciována. Pokud šablona sítě s daným jménem neexistuje, celý přechod selže. Pro výsledek platí to stejné, co v případě `i` operátoru výše.

`u` — smaž šablonu sítě. Operandem je jméno šablony, která má být odstraněna z databáze. Výsledkem je 1 v případě úspěšného odstranění, jinak 0.

binární

`:` — přiřaď hodnotu proměnné. Proměnná musí být volná. V opačném případě přechod selže. Prvním operandem je index proměnné, do které se má přiřadit operand druhý, libovolného typu. Po úspěšném přiřazení se volná proměnná stane vázanou. Návratovou hodnotou je druhý operand.

Operátor	Význam
`ns`	vytvoř prázdný řetězec
`na`	vytvoř prázdné pole
`nt`	vytvoř prázdnou n-tici
`d`	serializuj aktuální instanci

Tabulka 3.4: Přehled nulárních operátorů

Operátor	Význam
`!`	logická negace
`iv`	test, zda parametr obsahuje validní hodnotu
`ii`	test, zda parametr je integer
`is`	test, zda parametr je řetězec
`it`	test, zda parametr je n-tice
`ia`	test, zda parametr je pole
`in`	test, zda parametr je instance sítě
`h`	head — první položka seznamu nebo n-tice
`t`	tail — ostatní položky seznamu nebo n-tice
`p`	vstup z pinu daného výrazem
`s`	odešli hodnotu
`l`	nahraj šablonu sítě
`i`	nahraj instanci sítě
`c`	vytvoř instanci sítě
`u`	smaž šablonu

Tabulka 3.5: Přehled unárních operátorů

3.2 *PNAL*

3.2.1 Syntaxe jazyka

Pro simulátor by bylo zbytečně složité parsovat jazyk *PNML* 1.1.2, který je příliš obecný. Výsledný parser simulátoru napsaný ve *Slang*u či v čistém *C* by byl neúnosně kompexní a rozsáhlý. Tento parser je vhodnější napsat v přívětivějším programovacím jazyce a převést ho do zjednodušeného formátu sloužícího k přenosu popisu *RefNets* mezi *PC* a simulátorem *PNVM*. Na tento jazyk se budeme nadále odkazovat zkratkou *PNAL*.

Dosud žádný parser z *PNML* nebyl vytvořen. Dosud však syntaxe a sémantika tohoto jazyka není ustálena, proto vytvářet automatizovaný translátor nemá prozatím smysl. Zápis sítě do *PNAL* je tedy prováděn ručně. Je však natolik jednoduchý, že to není žádný problém i pro velké sítě. Jazyk si zde podrobně popíšeme, neboť nám to umožní ilustrovat, jak simulátor funguje a jaké funkce podporuje.

Tento jazyk zdaleka není finální a umožňuje pouze základní popis jednoduchých sítí.

3.3 Syntaxe *PNAL*

3.3.1 Lexikální struktura

Jazyk má několik vyhrazených znaků, které uvozují začátek, či konec popisu nějakého prvku nebo slouží k oddělení symbolů. Těmi jsou:

jednoduché závorky '(' a ')' slouží k ohraničení konkrétního prvku. Za otevírací závorkou následuje ihned znak značící typ elementu.

znak čárky ',' slouží jako oddělovač jmen, symbolů a parametrů.

znak dvojitých uvozovek '"' slouží pro ohraničení začátku i konce řetězcové konstanty. Podobně jako v jazyku *C* lze přítomnost tohoto znaku uvnitř řetězce zachovat

Operátor	Význam
`+`	součet dvou číselných hodnot
`-`	rozdíl dvou číselných hodnot
`*`	součin dvou číselných hodnot
`/`	podíl dvou číselných hodnot
`%`	zbytek po dělení
`&`	logický `and`
`|`	logický `or`
`~`	exkluzivní disjunkce
`=`	test rovnosti
`<`	test „menší než“
`>`	test „větší než“
`#`	vybere z pole prvek určený indexem
`a`	přidání prvku na konec pole
`,`	konkatenace řetězců
`o`	zápis hodnoty na výstup
`:`	přiřazení hodnoty

Tabulka 3.6: Přehled binárních operátorů

vložením zpětného lomítka ('\\'). Narozdíl od jazyka *C* nelze několik řetězcových konstant oddělených pouze sekvencí bílých znaků považovat za jeden řetězec:

```
("toto je " "validni zapis retezce v C")
```

Z lexikálního hlediska je potřeba zmínit to, že bílé znaky jsou ignorovány, pokud se nevyskytují v řetězci. Slouží především zapisovateli sítě pro zpřehlednění zápisu. Bílé znaky obsažené v řetězcích jsou ponechány.

Komentáře Nejsou dosud podporovány, což je podstatný nedostatek, nicméně z implementačního hlediska není problém tuto funkcionalitu doplnit.

Názvy Všechny znaky typů prvků sítě jsou případně-citlivé [5].

Veškeré názvy, ať už sítě, míst nebo proměnných přechodů, musejí odpovídat následujícímu regulárnímu výrazu:

Listing 3.1: Regulární výraz pro názvy ve specifikaci sítě

```
1 [a-zA-Z][a-zA-Z0-9]*
```

3.3.2 Typy prvků sítě

Před samotnou gramatikou si ještě uvedeme symboly pro označení typu elementu sítě, které se nacházejí vždy ihned za otevírací závorkou. Jsou jimi:

`N` prvek značící definici šablony sítě, jedná se o kořenový prvek celé sítě, který se v kódu šablony může vyzkytnout pouze jednou.

[5] je rozlišováno mezi jejich velikostí

`T` přechod sítě spolu s jeho celou definicí. Tento typ přechodu není *uplink*em, ale může obsahovat libovolný počet *downlink*ů.

`U` *uplink* značí přechod, který je *uplink*em. Je zde využito vlastnosti *RefNets* (viz 1.1.1), že přechod může mít maximálně jeden *uplink*. Přepisujeme-li síť z modelu *RefNets* do *PNAL*, bude každý přechod s *uplink*em značen ve výsledném kódu právě jedním elementem `U`.

`I` je inicializační přechod, který nesmí mít žadné podmíněné příkazy. Tento element se musí vyskytovat právě jednou v kódu sítě. Je spuštěn právě jednou, ihned po vytvoření instance.

Výše uvedené elementy jsou buď kořenovým elementem kódu šablony a nebo jeho přímými potomky a tedy nemohou se vyskytovat vnořeny uvnitř jeden druhého.

Následují elementy přechodů, které mohou mít za rodičovský element pouze jeden z typů přechodu (`T`, `U`, `I`).

`P` odstranění tokenů z místa. Slouží zároveň jako podmínka vykonání přechodu. Pokud místo neobsahuje požadovaný počet tokenů, přechod nemůže být proveden.

`G` strážní funkce. Obsahuje výraz, který se musí vyhodnotit jako pravdivý, aby mohl být přechod proveden. Je zde použita konverze *token*u, jako výsledku výrazu, na hodnotu *Bool*. Ta byla podrobně popsána výše 3.1.

`A` vykonání akce přechodu, která je specifikována výrazem. Na rozdíl od elementu `G` není výsledek výrazu brán v potaz pro hodnocení úspěšnosti provedení přechodu.

`D` *downlink* kanálu. Zahájení synchronní komunikace s *uplink*em jiné sítě. Jeho proměnné se dělí na parametry a lokální proměnné. Je předmětem zpětného navracení tak, jako kdyby všechny elementy volaného *uplink* přechodu byly zkopírovány a vloženy na místo tohoto elementu.

`O` vložení tokenů do místa.

`Y` vložení tokenů do místa se zpožděním. Zpoždění je specifikováno kladným celým číslem reprezentující milisekundy reálného času.

Všechny dosud popsané elementy jsou uzly kódu sítě a mohou tedy obsahovat elementy jiné jako své potomky. Nyní nám zbývá popsat poslední — listové — elementy kódu, jenž se mohou vyskytovat v hierarchii elementů pouze jako přímí či nepřímí potomci prvků přechodů.

`I` integer — prvek obsahuje celo-číselnou hodnotu v rozsahu $-2^{15} \leq i < 2^{15}$ v desítkové soustavě.

`S` identifikátor symbolu šablony sítě. Symbol je indexován kladným, celým číslem v rozsahu $0 \leq i < 2^{16}$. Index musí být menší, než počet pvků v kolekci symbolů sítě.

`V` identifikátor proměnné. Platí pro něj stejné omezení rozsahu jako u symbolu `S`. Index musí být menší, než počet proměnných v hlavičce přechodu, v nemž je identifikátor uveden. Mezi proměnné počítáme v případě *uplink*u i jeho parametry.

Pro *uplink* u s $P \in \aleph^0$ parametry a $V \in \aleph^0$ lokálními proměnnými, indexuje index i, $0 \leq i < P$ parametry a index j, $P \leq j < P + V$ lokální proměnné.

3.3.3 Gramatika

Následuje popis pravidel gramatiky jazyka *PNAL* se sémantickým popisem jednotlivých prvků. Nonterminální symboly se nacházejí v závorkách `<>`. Znak `':'` značí přepis nonterminálního symbolu na jeden z retězců terminálních a nonterminálních symbolů oddělených znakem `'|'` a ukončených středníkem `';'`. Znak ϵ značí prázdný řetězec. Počátečním nonterminálním symbolem gramatiky nechť je `<net>`.

Kořenový element

```
<net>
    : (N <name> (<symbols>) (<names>) <uplinks> <init> <transitions>)
    ;
```

Po znaku `N`, značícího kořenový prvek popisu sítě, následuje jméno sítě. Dále seznam symbolů, po něm seznam jmen míst. Následně specifikace *uplink* kanálů. Po nich právě jeden inicializační přechod a po něm několik obyčejných.

Seznam symbolů by v ideálním případě vůbec nemusel být specifikován a veškeré konstanty by mohly být obsaženy přímo v kódu sítě. To by však znamenalo, že duplicitní symboly by zabíraly zbytečně úseky objektové paměti navíc. Inteligentní parser šablon by samozřejmě dokázal toto ošetřit a vybudovat tabulku symbolů během zpracování zápisu šablony. Nicméně pro jednoduchost implementace byl zvolen zápis šablon co nejvíce podobný uložení dat v objektové paměti, případně v paměti kódu šablon.

Názvy a symboly

```
<names>
    : <_names> | ϵ
    ;
<_names>
    : <name>, <_names> | <name>
    ;
```

`<name>` je jméno odpovídající regulárnímu výrazu výše (3.1). Ty jsou v seznamu jmen odděleny vyhrazeným znakem `','`.

```
<symbols>   : <items> ;

<symbol>
            : <string> | <number> | <tuple> | <array>
            ;

<tuple>     : [ <items> ] ;
<array>     : { <items> } ;

<items>     : <_items> | ϵ ;
<_items>    : <symbol>, <_items> | <symbol> ;
```

`<string>` je řetězec znaků ohraničený uvozovkami. Může obsahovat libovolné tisknutelné znaky kromě znaku „konce řádku“ [6]. Podrobnosti k zápisu řetězcových konstant byly již zmíněny výše (3.3.1).

`<integer>` je celé číslo v dekadickém zápisu odpovídající regulárnímu výrazu:

[6] *Line Feed* — ASCII znak 10 v dekadickém zápisu

Listing 3.3: Prvky přechodů

```
<init>
    : (I <nofailcode>)
    ;
<transitions>
    : <transition> <transitions>
    | ϵ
    ;
<transition>
    : (T <name> (<names>) <code>)
    ;
```

Listing 3.2: Regulární výraz pro celá čísla

```
[−+](0|[1−9][0−9]*)
```

`<tuple>` je n-tice obsahující n různých symbolů. Jedná se o serializovanou podobu objektu typu *Tuple*. Podobně `<array>` je textový zápis objektu *Array*. Serializace bude detailněji popsána níže 3.5.

Přechody sítě

```
<uplinks>
    : <uplink> <uplinks> | <uplinks>
    ;
<uplink>
    : (U <name> (<names>) (<names>) <code>)
    ;
```

Uplink přechodů může být libovolný počet — od 0 po n. Kažný z nich obsahuje název, seznam názvů parametrů a seznam názvů lokálních proměnných. Nakonec podmínky a akce přechodu představované symbolem `<code>`.

Výpis 3.3 obsahuje pravidla pro nonterminály inicializačního a obyčejných přechodů.

Inicializační přechod nepotřebuje název a nemá žádné argumenty ani lokální proměnné. Obsahuje jen kód, který nemůže selhat — nemá žádné podmínky. Unicializační přechod musí být právě jeden.

Obyčejných přechodů může být stejně jako *uplink* přechodů neomezeně. Mají název a seznam lokálních proměnných.

Elementy kódu přechodů

jsou uvedeny ve výpisu 3.4. Jejich pořadí významně ovlivňuje efektivitu provádění přechodu. Protože tak, jak jsou za sebou uvedeny, jsou při simulaci prováděny. Proto by všechny podmíněné příkazy měly předcházet nepodmíněným. Parser pořadí nijak nekontroluje, a simulátor kód pouze slepě vyhodnocuje, takže efektivita provádění je čistě v rukou autora popisu sítě. Rozdělení příkazů na `<conditionals>` a `<nofailcode>` je z hlediska parsování důležité především pro inicializační přechod, kde se podmíněné příkazy nesmí vyskytnout.

Kód `P` odstraní z místa určeným nonterminálem `<placeid>` počet (`<amount>`) *token*ů (`<value>`).

Listing 3.4: Pravidla pro bloky kódu přechodů

```
<code>
    : <conditionals> <nofailcode> ;

<conditionals>
    : (P <placeid>, <amount>, <value>)   <conditionals>
    | (G <expression>)                   <conditionals>
    | (D <name>, <variableid>, <values>) <conditionals>
    | ϵ
    ;

<nofailcode>
    : (A <expression>)                          <nofailcode>
    | (O <placeid>, <amount>, <value>)          <nofailcode>
    | (Y <placeid>, <amount>, <value>, <time>) <nofailcode>
    | ϵ
    ;
```

Kód `D` vyhledá v šabloně instance sítě, určené indexem do pole proměnných reprezentované symbolem `<variableid>`, *uplink* s názvem `<name>` a předá mu seznam parametrů `<values>` ve stejném pořadí, jak jsou v tomto elementu uvedeny.

Kód `O` umístí do místa `<placeid>` `<amount>` tokenů s hodnotou `<value>`.

Kód `Y` je obdobou předchozího s tím rozdílem, že poslední nonterminál `<time>` určuje časový interval, za jaký má k vložení dojít. Jeho hodnota musí být v rozsahu 32-bitového **unsigned** integeru.

Proměnné a hodnoty symbolů

uvádí výpis 3.5. `<posinteger>` je celé číslo v rozsahu $\langle 0, 2^{16} - 1 \rangle$.

Nonterminály končící na `id` představují index do odpovídajícího pole názvů. V prvním případě (řádek 12) je to index do pole názvů míst, ve druhém do pole symbolů a v posledním do pole názvů proměnných přechodu.

Je nutno podotknout, že symboly sítě, příkazy přechodů ani výrazy nemohou obsahovat vnořenou instanci sítě jako hodnotu. Nicméně může být přítomna v řetězcové konstantě jako symbolu sítě a z něj může být nahrána pomocí operátoru `i` (viz 3.1.1). To samé platí pro vnořený kód šablony sítě.

3.3.4 Výrazy

Tvoří tělo elementů stráží a akcí. Jedná se o aplikaci operátorů s aritou 0 až 2 v prefixové notaci. Zápis je podobný programu v jazyku *lisp*. Pravidla přepisu výrazů jsou uvedeny ve výpisu 3.6. Výpis neuvádí přepis nonterminálů unárních a binárních operátorů, ty jsou analogií k pravidlům s nonterminálem `<nularyop>` na levé straně. Čtenář si je může vyvodit z tabulek operátorů 3.5 a 3.6.

Sémantika jednotlivých operátorů je popsána výše (viz 3.1.1).

Zkrácené vyhodnocování

Pro zrychlení vyhodnocování výrazů je využit princip zkráceného vyhodnocování operandů. To je mechanismus, kdy u operátorů s aritou > 1 je vyhodnocena pouze část operandů

Listing 3.5: Pravidla přepisu proměnných a hodnot symbolů

```
<amount>    : <posinteger> ;
<variable>  : V <variableid> ;

<value>
    : I <integer>
    | <variable>
    | S <symbolid> ;

<values>    : <_values> | ϵ ;
<_values>   : <value>, <_values> | <value> ;

<placeid>   : <posinteger> ;
<symbolid>  : <posinteger> ;
<variableid>: <posinteger> ;


<variables>  : <_variables> | ϵ ;
<_variables> : <variableid>, <_variables>
             | <variableid> ;
```

Listing 3.6: Pravidla výrazů

```
<expression>
    : (<binaryop> <expression> <expression>)
    | (<unaryop>  <expression>)
    | (<nularyop>)
    | (<value>)
    ;

<nularyop>
    : ns | na | nt | d ;
```

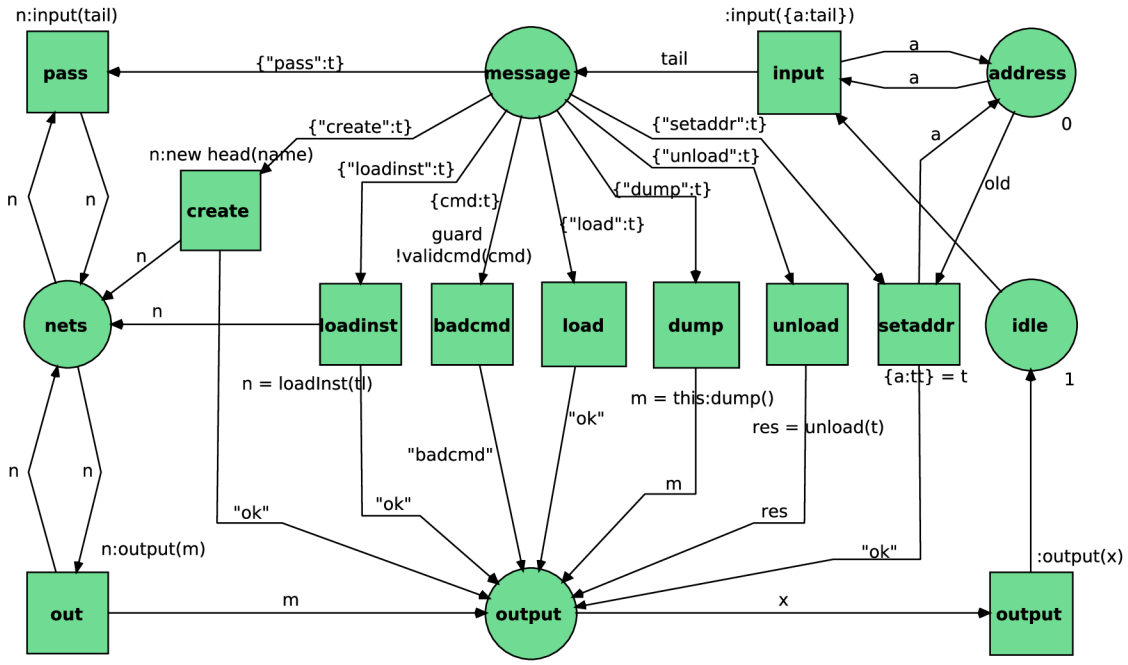
a v případě, že vyhodnocené operandy postačují na určení výsledku, je tento vrácen, a zbytek operandů zůstává nevyhodnocen.

Náš referenční simulátor *Renew* tento princip neimplementuje z důvodu možnosti konkurentního zpracování operandů, kdy zkrácené vyhodnocování implikuje určení pořadí vyhodnocování jednotlivým operandům. My však nepočítáme s využitím více vláknového výpočtu a tudíž je pro nás tento princip výhodný k navýšení efektivity programu.

V současné době je tento mechanismus využit pro logické operátory `&` a `|`.

3.4 Příklad sítě v *PNAL*

Pro ujasnění pravidel gramatiky si zde uvedeme příklad popisu sítě generovaného gramatikou tohoto jazyka. Příklad bude odpovídat modelu sítě vytvořeného programem *Renew* (viz 1.1.3), znázorněnného na obrázku 3.1.



Obrázek 3.1: Příklad popisované *RefNet* sítě — `platform`

Je to jednoduchá platforma umožňující běh sítí, které referencuje a sama vytváří. Se svým okolím komunikuje pomocí jednoduchých textových zpráv synchronními kanály `:input` a `:output`. Má v sobě uloženu adresu, podle které ověřuje, zda přijímaná zpráva jí patří — v našem případě je to 0. Adresa může mít libovolnou hodnotu.

Po předání příkazu síti prostřednictvím kanálu `:input` se z místa *idle* odstraní *token* 1, značící připravenost sítě na zpracování nového příkazu. Ten je vrácen zpět zároveň s odstraněním výstupní zprávy — výsledku provedení příkazu — a jeho předání parametrem *output* volajícímu. Volajícím může být jiná instance stejné platformy, případně zcela jiné šablony, a nebo přímo simulační program *PNVM*. Dokud *token* 1 není vrácen do místa *idle*, síť žádnou další zprávu nepřijme. Tím je však kladen podstatný požadavek na sítě spravované v místě *nets*. Konkrétně každá spravovaná síť musí po akceptování zprávy jejím *uplink*em `:input` a jejím zpracování, odpovědět právě jedním *token*em. Pokud by totiž

Listing 3.7: Výsledek převodu modelu do jazyka *PNAL*

```
(N platform
  (0,
    "setaddr", "pass", "load", "create", "dump", "loadinst", "unload",
    "ok", "failed", "badcmd")
  (address, nets, idle, message, output)
```

libovolná zpráva předaná spravované síti zůstala nezodpovězená, platforma by zamrzla ve stavu *idle*.

Přechod *badcmd* odstraní zprávu z místa *message* v případě, že prvním prvkem není jeden z podporovaných příkazů. Odpovědí je pak "badcmd".

Je to zároveň zjednodušená podoba výchozí šablony sítě, která je nahrána jako první do databáze šablon aplikace `pnvmx86`.

3.4.1 Příklady komunikace se sítí `platform`

- [0, "pass", 20] předá číslo 20 jedné z referencovaných sítí v místě *nets*. Tzn. že jedné z těchto sítí předá synchronním kanálem `:input` *token* 20. Downlink `n:input` s tímto parametrem je volán dokud přechod pro libovolnou instanci z místa *nets* nebude úspěšně proveden.
- [0, "load", net_template] načte do databáze šablonu sítě pro její pozdější instanciaci. net_template je serializovný objekt typu *String*.
- [0, "create", "net_name"] vytvoří novou instanci sítě net_name a vloží ji do místa *nets*.
- [0, "setaddr", "address"] změní adresu platformy na řetězcovou konstantu "address".
- [0, "unload", "net_name"] odstraní z databáze šablonu s názvem "net_name".

3.4.2 Výsledný zápis sítě

Převedením tohoto modelu do jazyku *PNAL* získáme řetězec, jehož první část je zobrazena ve výpisu 3.7. Obsahuje definici sítě až po nonterminál `<uplinks>`. Do popisu byly přidány mezery a odřádkování pro přehlednost. Jak však již bylo zmíněno výše (3.3.1), přidané mezery jsou akceptovány parserem našeho simulátoru a jedná se tedy o validní kód.

Všimněme si, že veškeré řetězce z modelu na obrázku 3.1 jsou uvedeny mezi symboly sítě začínajícími na řádku 2. Prvním symbolem je číslo 0 a představuje výchozí adresu instance. Ta je inicializačním přechodem umístěna do místa *address*. Hodnota výchozí adresy by stejně tak mohla být uvedena přímo v kódu inicializačního přechodu neboť se jedná o *Integer*. Umístěním její hodnoty mezi symboly ji však činíme přístupnější k editaci.

Na řádku 5 nalezneme výčet názvů všech míst sítě. V kódu přechodů se na symboly a názvy míst neodkazujeme jejich jmény, nýbrž indexy do tohoto výčtu, indexovanými od nuly.

Listing 3.8: Pokračování kódu sítě `platform` s definicí *uplink* přechodů

```
(U input (msg) (addr, command)
   (P 2, 1, I1)
   (P 0, 1, V1)
   (G (= (h(V0)) (V1)))
   (A (: (V2) (t (V0))))
   (O 0, 1, V1)
   (O 3, 1, V2))
(U output (x) ()
   (P 4, 1, V0)
   (O 2, 1, I1))
```

Uplink přechody

Jak je vidět na modelu sítě 3.1, přechod *input* se stejnojmenným *uplink*em odstraňuje z místa *address token a* a z místa *idle* libovolnou přítomnou hodnotu. Token *a* je porovnán s prvním prvkem přijatého parameteru kanálu a pokud se shodují, je zbytek parametru vložen do místa *message* a adresa *a* je vrácena zpět.

V popisu *PNAL* je toto zapsáno následujícím způsobem:

1. odstranění 1 *token*u z místa 2 (*idle*) a jeho porovnání s hodnotou 1. Řádek 2.
2. odstranění 1 tokenu z místa 0 (*address*) a jeho vložení do proměnné číslo 1, v tomto případě do lokální proměnné *addr*. Je to druhá podmínka provedení přechodu. Řádek 3.
3. vyhodnocení výrazu `(= (h (V0)) (V1))`, což je prefixový zápis porovnání výrazu `h(V0)` s `V1`. První položka proměnné jedna, což je parametr *msg* kanálu, je povnána s proměnnou 1 (*addr*) obsahující načtenou adresu z předchozícho příkazu. Po ověření pravdivosti výrazu nás stráž pouští k dalším krokům, které už nemohou selhat. Je to tedy poslední podmínka provedení tohoto přechodu. Řádek 4.
4. provedení akce popsané výrazem `(: (V2) (t (V0)))`. To je přířazení položek kolekce, počínaje druhou položkou dále, do lokální proměnné *command*. Řádek 5.
5. vložení jednoho *token*u s hodnotou proměnné 1 (*addr*) do místa s indexem 0 (*address*). Vrátí adresu zpět. Řádek 6.
6. vložení jednoho tokenu s hodnotou proměnné 2 (*command*), tedy zbytku přijaté zprávy, do místa *message*. Řádek 7.

Uplink output už nebudeme podrobně rozepisovat. důležité je si uvědomit, že podmíněný příkaz `P` může být použit jak k navázání proměnné na libovolný *token* s dostatečným počtem výskytů v daném místě, tak k ověření, zda místo obsahuje konkrétní hodnotu v dostatečném množství. Zároveň je daný počet tokenů z místa odebrán a to do doby, než přechod

- selže — v tom případě je akce odebrání invertována
- uspěje — pak je tato změna permanentní

Listing 3.9: Pokračování kódu sítě `platform` s definicí *uplink* přechodů

```
(T badcmd (msg, cmd)
   (P 3, 1, V0)
   (G (: (V1) (h (V0))))
   (G (! (| (| (| (= (V1) (S1))
                  (= (V1) (S2)))
               (| (= (V1) (S3))
                  (= (V1) (S4))))
            (| (| (= (V1) (S5))
                  (= (V1) (S6)))
               (| (= (V1) (S7))
                  (= (V1) (S8)))))))
   (O 4, 1, S11))
(T pass (net, cmd, msg)
   (P 1, 1, V0)
   (P 3, 1, V1)
   (G (= (h (V1)) (S2)))
   (G (| (: (V2) (h (t (V1)))) (I1)))
   (D input, V0, V2)
   (O 1, 1, V0))
```

V případě příkazu na řádku 9 se jedná o navázání proměnné x na *token* z místa *output*. Kdyby však parametr x nebyl volný, ale vázaný [7], jednalo by se o ověření rovnosti (použitím operátoru (=)).

Inicializační přechod

```
(I (O 0, 1, S0)
   (O 2, 1, I1))
```

Musí být právě jeden a musí předcházet obecné přechody. Navíc nesmí obsahovat podmíněné příkazy a především žádné *downlink* volání. Výše uvedený přechod vkládá do místa *address* symbol 0 (adresu 0) a do místa *idle token* 1.

Obyčejné přechody

Letmo si ukážeme zápis pouze dvou obyčejných přechodů odpovídajících modelu 3.1 na výpisu 3.9. Na přechodu není nic zvláštního, kromě využití proměnné 1 pro uložení výsledku aplikace operátoru `h` na kolekci `msg` pro pozdější použití ve druhé stráži. Zápis by mohl být nadále zkrácen tak, že by se přiřazení odložilo až do prvního operandu prvního porovnání ve stráži dva. Jak je vidno, možností zápisu je velké množství i s takto skromnou množinou výrazových prostředků.

Přechod *pass* (18) je zajímavý vústěním kanálu `input` volané sítě. Prvním operandem je instance sítě *net*, na níž je přechod *input* proveden. V případě, že by *downlink* selhal, dostaneme se zpětným navracením až k prvnímu podmíněnému příkazu vybírajícímu z množiny sítí. Dokud se *downlink* úspěšně neprovede nebo dokud se nevyčerpají všechny možnosti výběru síte z místa *nets*, bude se volání stále opakovat. Při úspechu je instance volané sítě navrácena zpět do místa *nets*.

[7] to je možné, v případě, že je tak *downlink* volán

Další přechody již nebudeme rozepisovat a ponecháme toto cvičení na čtenáři. Důležité je si uvědomit, že tento formát je přímo popisem, dle kterého lze celou síť simulovat. Není třeba žádných dodatečných úprav nebo komprese. To je plně využito současnou implementací interpretu.

3.5 Serializace

Pro potřeby komunikace s okolím simulátor *PNVM* poskytuje prostředky, jak objekty uživatelských datových typů serializovat do textové podoby a z této formy opět nahrát původní objekty. Podpora serializace je pro všechny uživatelské datové typy.

Zde je několik obecných pravidel pro tuto konverzi:

- Všechny znaky textové podoby jsou v rozsahu `ASCII` (sedmi-bitová hodnota) a jsou tisknutelné.
- Všechna čísla a indexy jsou v dekadickém zápisu.

Následuje postup konverze pro jednotlivé datové typy.

String Jeho texová podoba je shodná se zápisem v jazyce *PNAL* pro popis šablony sítě (viz 3.3.1). Konverze na textovou podobu probíhá tak, že obsah hodnoty se uzavře z obou stran dvojitými uvozokami (znak '"' — 0x22 v `ASCII`) a před všechny výskyty těchto uvozovek a zpětných lomítek uvnitř serializované hodnoty se umístí zpětné lomítko (znak '\\' — 0x22 v `ASCII`).

Konverze zpět ze serializované podoby je inverzní operace k popsané a nebudeme ji dále rozvádět.

Integer Má shodnou textovou reprezentaci jako token `<integer>`, který je načten lexikálním parserem. Je značen regulárním výrazem 3.2.

Array* a *Tuple Zde se opět můžeme odkázat na syntaxi jazyka *PNAL* 3.3.3. Všechny položky hodnoty jsou serializovány tímto stejným konverzním procesem a odděleny čárkami. Výsledná sekvence je pak ohrazena závorkami [] resp. {} pro hodnotu typu *Tuple* resp. *Array*.

3.5.1 NetInst

Serializace instance sítě je náročnější a gramatika jazyka *PNAL* ji nezná. Uvedeme si proto gramatiku novou pro popis textové formy instance sítě. Počátečním nonterminálem nechť je `<inst>`, dále mějme nonterminály

- `<place>` reprezentující místo sítě
- `<places>` reprezentující seznam míst
- `<plcitem>` pro položku místa
- `<plcitems>` pro seznam položek místa
- `<token>` pro hodnotu místa

Zbytek nonterminálů v následujícím výpisu pravidel je pak převzat z gramatiky jazyka *PNAL*.

```
<inst>
    : (n <name> <places>)
    ;
<place>
    : (p <name> <plcitems>)
    ;
<plcitem>
    : (t <posinteger>, <token>)
    ;

<places>   : <place>   <places>    | ϵ ;
<plcitems> : <plcitem> <plcitems>  | ϵ ;

<token>
    : <symbol>
    | <inst>
    ;
```

Kořenový element obsahuje název šablony sítě k níž instance náleží a seznam míst. Ten může místa obsahovat v libovolném pořadí, díky tomu, že textový připis místa je označeno jménem. To musí být přítomno názvech míst odpovídající šablony sítě.

Všimněme si, že položky míst mohou obsahovat kromě symbolů šablony i vnořené instance. Kořenový element se tedy v tomto případě může libovolně rekurzivně zanořovat.

3.5.2 Současná omezení

V současné době nejsou podporovány ne-`ASCII` znaky. Dále chybí možnost serializace netisknutelných znaků — i znaků pro odřádkování. To jsou však drobné implementační nedostatky, které lze snadno odstranit.

Identické objekty Zásadnějším problémem je však rozlišení identických objektů v serializované podobě. Z hlediska simulace je velmi důležité rozlišení mezi ekvivalentními a identickými objekty. Ekvivalentní objekty mohou a nemusí být identické. Identické jsou v objektové paměti uloženy na stejné adrese. Pokud objekt modifikujeme, všichni jeho referenti budou ukazovat na modifikovanou podobu.

Aktuální verze simulátoru serializuje všechny reference na objekty jako vnořené serializované objekty. To má za následek, že kolekce nebo instance sítě s několika referencemi na identický objekt bude po serializaci a opětovném nahrání obsahovat reference na separátní, neidentické objekty.

Toto je podstatný nedostatek. Řešení však nebude příliš náročné. Během procesu serializace je pouze potřeba si uchovávat reference na již serializované objekty a na ně se ve zbytku serializovného textu odkazovat. Podobně, jak je tomu v definici šablony sítě u symbolů.

Kapitola 4

Objektová paměť

Správa paměti je kritická v aplikacích pro vestavěné systémy. V případě použití dynamické alokace paměti dochází k dvěma nepříjemným jevům. K fragmentaci a možnosti kolize zásobníku s heapem obsahujícím dynamicky alokované bloky. K fragmentaci dochází při alokacích bloků o různé velikosti a jejich uvolňováním v různém pořadí. Kdy celková velikost dostupné paměti je sice dostatečně velká pro blok, který je třeba alokovat, ale volná paměť je rozprostřena mezi bloky zabrané tak, že nelze dostatečně velký souvislý blok paměti najít.

Problém fragmentace lze snadno vyřešit tak, že veškeré alokace budou požadovat bloky paměti konstantní velikosti na adresách zarovnaných na tuto velikost alokačního bloku. V tom případě alokace vždy uspěje, dokud není využit poslední volný alokační blok. Jakmile je jeden z bloků uvolněn, může být znovu alokován pro jiný účel. Tohoto přístupu využívá *PNVM* simulátor. Tato kapitola uvede detaily implentace.

4.1 Blok

Před tím, než se seznámíme s typy alokačních bloků, je potřeba si shrnout informace o objektech, které je potřeba dynamicky alokovat. *PNVM* potřebuje uchovávat několik typů datových objektů, těmi jsou:

- Šablony sítí. Ty obsahují jména míst a přechodů, hodnoty symbolů, kód přechodů a jména jejich proměnných/parametrů.
- Instance sítí. Obsahující objekty míst.
- Místa instancí sítí. Jsou to páry hodnot tokenů, případně odkazy na ně, s počty výskytů v daném místě.
- Řetězce pro objekty typu *String*.
- n-tice pro objekty typu *Tuple*.
- Pole pro objekty typu *Array*.
- Události reprezentující zpožděné vložení *token*u do místa konkrétní instance sítě.

Celá čísla pro objekty typu *Integer* není nutno uchovávat v samostatných blocích paměti. Využijeme zde vlastnosti, že datový typ integer je v jazycích *C*, *C++* přibližně stejně veliký, jako ukazatel do paměti, v závislosti na architektuře. Pokud zvolíme ze základních datových typů ten, který má s ukazatelem shodnou velikost, můžeme ve stejných blocích paměti uchovávat stejný počet hodnot typu *Integer* jako ukazatelů na dynamicky alokované hodnoty. V uvedených jazicích je takovým typem `intptr_t` resp. `uintptr_t` pro znaménkovou resp. bez-znaménkovou variantu. Ty jsou definované v hlavičkovém souboru `stdint.h`.

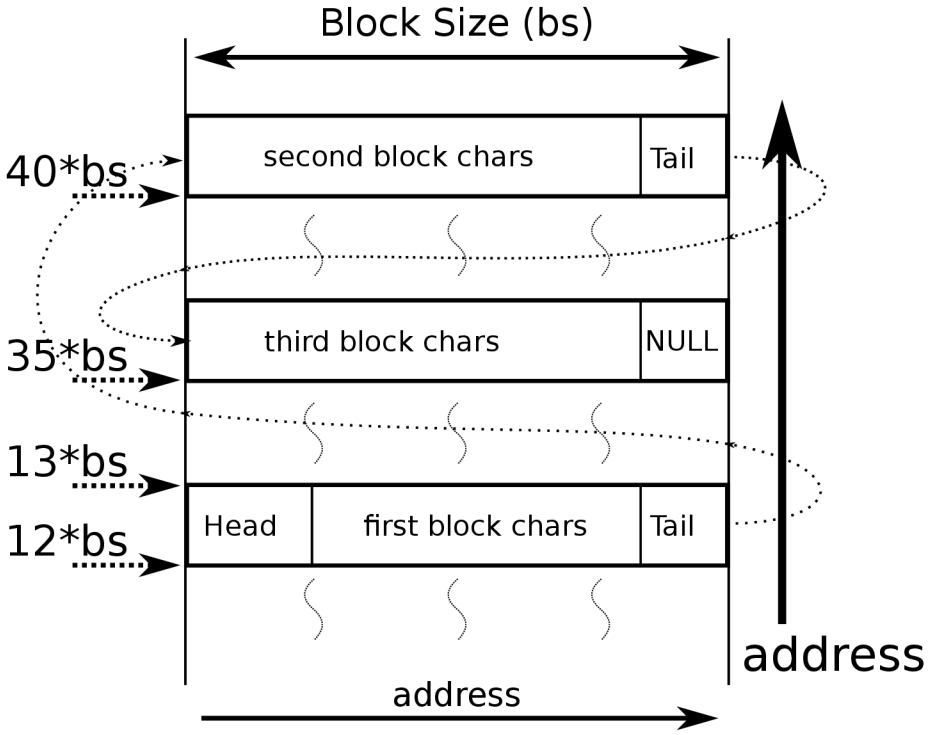
Jak je vidno, objekty žádného z výše zmíněných typů nelze uchovávat v blocích konstantní velikosti až na hodnoty typu *Integer* a události. Ostatní datové typy obsahují teoreticky neomezený počet referencí na jiné objekty. Je tu však možnost tyto reference distribuovat do 1 a více bloků stejné velikosti podle potřeby. Tyto bloky samozřejmě nebudou zabírat souvislý úsek paměti, nýbrž budou náhodně rozprostřeny po celé haldě podle aktuálního stavu jejího zaplnění v čase alokace.

Když se tyto bloky budou navzájem referencovat, není potřeba k uchování hodnoty daného typu více než jeden ukazatel — a to ukazatel na první blok.

4.1.1 Struktura bloku

Protože kritické na vestavěných systémech není většinou rychlost provádění, ale spíše paměťová náročnost, spokojíme se pouze s jednosměrným odkazováním na následující blok. Obousměrný seznam by umožnil daleko efektivnější provádění algoritmů nad hodnotami uloženými v mnoha nesouvislých blocích, ovšem přidáním odkazu na předešlý blok se velikost užitečného místa výrazně zredukuje.

Obrázek 4.1 znázorňuje hodnotu typu *String* rozprostřenou ve třech jednosměrně referencovaných blocích. První blok — označme jej *header* — je uložen na nejnižší adrese [1],



Obrázek 4.1: Příklad rozložení hodnoty typu *String* v několika blocích

jenž je násobkem alokačního bloku. Pro přístup k hodnotě nám tedy stačí ukazatel na něj. Ke zbytku dat uložených v dalších dvou blocích se lze dostat přes ukazatele `tail` na konci bloku. Protože k prvnímu bloku je nejsnadnější přístup, je výhodné, aby obsahoval všechna metadata objektu, která jsou důležitá, ať už pro správu paměti, typovou kontrolu nebo přístup k prvkům. Metadata jsou uložena na začátku každého *header* bloku ve struktuře, kterou nazvěme *Head*.

Metadata *header* bloku

Existuje několik obecných atributů, které chceme v této stuktuře uchovávat:

signatura slouží pro určení typu hodnoty. Je to konstanta výčtového typu. Díky ní lze obecnou hodnotu vždy správně přetypovat na odpovídající typ. Tato informace je přítomna v každém *header* bloku libovolného typu hned na začátku.

počítadlo referencí obsahuje počet referencí na tento objekt. Je to údaj sloužící správě paměti k určení, zda je možno objekt smazat.

počet položek říká, kolik položek je ve všech blocích hodnoty uloženo, případně referencováno. Díky tomuto údaji přístupnému přímo v hlavičce není nutno procházet všechny bloky pro zjištění délky. Výhoda uchování této informace je také v tom, že část *Tail* posledního bloku lze také využít jako úložný prostor.

Toto jsou informace společné většině datových typů. Ty datové typy, které danou informaci nepotřebují ji pro úsporu místa neobsahují. Některé datové typy vyžadují přítomnost dalších informací, které si uvedeme v dalším textu.

4.1.2 Koncept *header* a *tail* bloku

Datové typy obsahující proměnlivý počet odkazů nebo přímo hodnot jiných typů — v sekci datové typy (viz 3.1 jsme je nazvali „indexovatelné“) — jsou uloženy ve dvou typech bloků. Obecný typ bloku *header* byl již popsán. Zbývá popsat vlastnosti druhého typu bloků, který je definován pouze pro indexovatelné typy.

Pro všechny datové typy je struktura bloku *tail* shodná. Obsahuje pouze hodnoty položek a odkaz na další blok. V případě, že blok je posledním v řadě bloků hodnoty, je ukazatele na další prvek využito jako úložného prostoru pro další položky. Obrázky 4.2 zachycují využití bloku položkami pro koncový (označme jej jako *last*) nekoncový (označme jej *not_last*) blok.

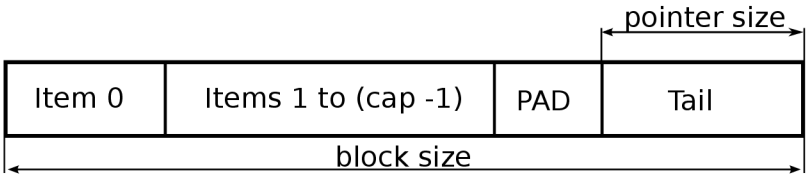
4.1.3 Výpočty nad bloky

Mějme množinu $B = H \bigcup T$ bloků skládajících se z množiny bloků *head* (H) a *tail* (T), kde $H \bigcap T = \emptyset$. Dále mějme množinu I datových typů položek bloků a zobrazení $items : B \to I$ přiřazující každému typu bloku typ jeho položek.

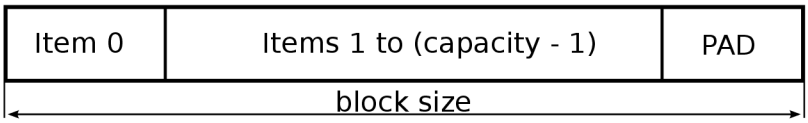
K určení počtu bloků (bc) pro uchování $k \in \aleph^0$ položek v typu bloku $b \in B$ pak lze využít následující vzorec:

$$bc(b,k) = \begin{cases} 1 & \text{když} \quad \neg indexable(b) \vee k < lc(b) \\ 2 + \max\left(0, \frac{k - nlc(b) - lc(tail(b))}{nlc(tail(b))}\right) & \text{jinak} \end{cases} \tag{4.1}$$

[1] uložení bloků je v paměti je zcela náhodné, v tomto případě je první blok hodnoty na adrese `12bs`



(a) Bloky $b_i, 0 \leq i < n-1$ (*not_last*)



(b) Poslední blok b_{n-1} (*last*)

Obrázek 4.2: Poslední blok

Kde:

$indexable : B \to 0, 1$ je zobrazení z množiny typů bloků na množinu $\{0, 1\}$, kdy 1 značí, že blok je indexovatelný.

$lc : B \to \aleph^0$ je zobrazení mapující typ bloku na jeho maximální kapacitu s využitím prostoru *Tail* jako užitečného místa.

$nlc : B \to \aleph^0$ je zobrazení mapující typ bloku na jeho maximální kapacitu jako bloku *last* — tedy bez využití prostoru *Tail* jako úložného místa.

$tail : B \to T$ přiřazuje typu bloku odpovídající blok *tail*.

Díky tomu, že známe velikost alokačního bloku ($bs \in \aleph^+$), velikosti datových typů položek pro každý blok a metadat bloků *Head*, je snadné určit zobrazení lc a nlc jako:

$$lc(b) = \begin{cases} \frac{bs - \text{sizeof}(head(b))}{\text{sizeof}(item(b))} & \text{pro } b \in H \\ \frac{bs}{\text{sizeof}(item(b))} & \text{pro } b \in T \end{cases} \tag{4.2}$$

$$nlc(b) = \begin{cases} \frac{bs - \text{sizeof}(head(b)) - ps}{\text{sizeof}(item(b))} & \text{pro } b \in H \\ \frac{bs - ps}{\text{sizeof}(item(b))} & \text{pro } b \in T \end{cases} \tag{4.3}$$

Kde ps je velikost ukazatele na další blok — velikost části *Tail*. A $head : H \to M$ zobrazuje *header* blok na datový typ jeho metadat — část *Header*.

Pro získání celkové kapacity $tc(b, n)$ pro n bloků, kde $b \in H$ lze využít následující výpočet:

$$tc(b, n) = \begin{cases} 0 & \text{když } n < 1 \\ lc(b) & \text{když } n = 1 \\ nlc(b) + (n-2) * (nlc(tail(b)) + lc(tail(b)) & \text{když } n > 1 \end{cases} \tag{4.4}$$

4.1.4 Typy bloků

Výše (4.1) byl popsán přehled datových typů, které je nutné uchovávat v objektové paměti. Zde si popíšeme konkrétní abstrakce těchto typů s jejich atributy.

Všechny níže uvedené třídy jsou potomky jedné z bázových tříd `PNVMHeaderBlock` nebo `PNVMTailBlock` v závislosti na tom, zda se jedná o abstrakci bloku *header* nebo *tail*. Tyto bázové třídy umožňují uniformě manipulovat se samotnými bloky. V prostředí *Squeak* je informace o typu vždy implicitně přítomna a při zaslání zprávy libovolnému bloku se vždy provede patřičná metoda. To však neplatí pro *C++* implementaci neboť kvůli úspoře paměti tyto třídy nejsou virtuálními. Proto v případě potřeby zaslat objektu zprávu, jenž je definována pouze pro určitý datový typ, je třeba objekt explicitně přetypovat.

C++ implementace však pro často používané funkce definuje specializace pro tyto obecné třídy, které se dle signatury objektu rozhodnou, jak objekt přetypovat a zavolají na něj odpovídající přetíženou funkci.

Šablona sítě

Je reprezentována třídou `PNVMTemplate`. Udržují spoustu různorodých informací, především ale kód přechodů, které zabírají podstatnou část operační paměti. Kdybychom jej chtěli uchovávat v dynamicky alokovaných blocích, znamenalo by to navýšení potřebné paměti k jeho uložení o spoustu ukazatelů navíc. Mnohem závažnějším nedostatkem tohoto přístupu by však měl dopad na efektivnost. Kód přechodů je procházen a vyhodnocován simulátorem *PNVM*, který se do kódu odkazuje ukazatelem a předpokládá, že určité části kódu mají danou velikost nebo jsou uvozeny konkrétními znaky. Díky tomu se může v souvisle uloženém kódu rychle pohybovat a přeskakovat části, které nejsou v daném kontextu podstatné.

Kód přechodů šablon je tedy uložen v separátním, souvislém, staticky alokovaném bloku paměti. Nazvěme jej `netTemplateCode`. Instance šablon se do něj odkazují ofsetem.

Obsahuje z obecných atributů pouze signaturu. Dále uchovává:

počet míst je kladné číslo určující počet míst, které budou alokovány každé instanci této šablony.

offset do kódu šablon — ofset relativní vůči začátku `netTemplateCode`. Ukazuje na první, otevírací závorku kořenového elementu — tedy těsně před znak `N`.

délku kódu — Objektová paměť si neudržuje informaci o využití paměti kódu. Ta je vypočítána z atributů ofsetu a délky kódu všech šablon sítí v databázi, až když je tato informace potřeba. To je obvykle při nahrávání šablony nové nebo naopak při odstraňování staré.

transakce je ofset do `netTemplateCode` na začátek definice prvního obecného přechodu. Simulátor přistupuje ke kódu těchto přechodů nejčastěji a proto je z hlediska efektivity důležitý co nejrychlejší přístup na tuto pozici v kódu.

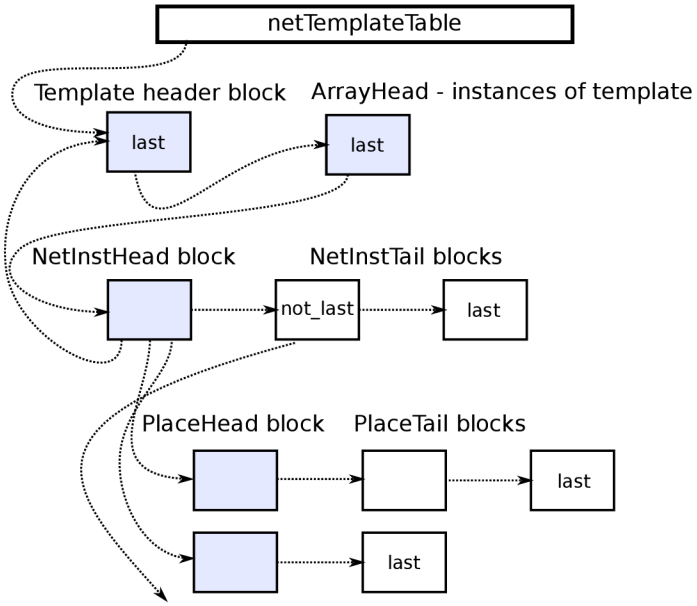
název šablony je uložen jako reference na objekt typu *String*. To představuje řadu výhod. Především možnost rychlého porovnání při vyhledávání šablony podle jména. Neboť jména, ať už dynamicky načtená nebo vytvořená, jsou opět objekty tohoto typu. Dále je možné se na jméno odkazovat přímo a není proto třeba názvy duplikovat. V neposlední řadě nám oddělený úložný prostor pro jméno umožňuje mít všechny atributy šablony uložené pouze v jednom bloku.

názvy míst jsou objekty typu *String* uložené v n-tici ze stejných důvodů jako název šablony.

symboly šablony jsou uchovány stejně jako názvy míst. Díky tomu je lze referencovat a tím je šetřeno objektovou pamětí.

instance je pole referencí na instance šablony. Na obrázku 4.3 si lze všimnout, že se jedná o ukazatel na objekt typu `PNVMArrayHead`.

Díky tomu, že máme vytvořeny abstrakce datových typů pro kolekce, je možno je využít pro úschovu položek, které by jinak musely být součástí instance šablony. Takže pro uchování hodnoty typu šablony slouží právě jeden blok typu `PNVMTemplate`.



Obrázek 4.3: Ukázka referencování objektů v simulátoru *PNVM*

Instance sítě

Je reprezentována třídami `PNVMNetInstHead` a `PNVMNetInstTail` pro *header* resp. *tail* bloky. Jedná se o uživatelský datový typ *NetInst*. Obsahuje jako položky ukazatele na místa (konkrétně na `PNVMPlaceHead` objekt). Z obecných atributů obsahuje všechny a navíc ve své hlavičce drží ukazatel na instanci šablony (`PNVMTemplate`), z níž vznikla.

Místo

Je reprezentováno třídami `PNVMPlaceHead` a `PNVMPlaceTail`. Položkami jsou objekty typu `PNVMPlaceItem`, které v sobě uchovávají hodnoty tokenů s číselnou hodnotou počtu výskytů.

Místo není uživatelským datovým typem, náleží pouze objektu *NetInst*, proto u něj není potřeba počítadla referencí.

Pole a n-tice

Z implementačního hlediska mezi nimi není rozdílu, jsou to oba uživatelské datové typy, jejichž položkami je `PNVMToken`. Co se týče vnitřního uspořádání, je jedinnou odlišností hodnota signatury. Rozlišení na pole a n-tice je tak pouze zužitkováno operátory jazyka *PNAL*, které mohou pro tyto typy mít jinou sémantiku.

Pole je reprezentováno třídami `PNVMArrayHead` a `PNVMArrayTail`, zatímco n-tice třídami `PNVMTupleHead` a `PNVMTupleTail`.

Oba jsou hojně využívány samotnou implementací simulátoru *PNVM* jako pomocné datové typy. n-tice obecně tam, kde informace má pevně danou kvantitu, zatímco pole tam, kde množství položek není předem znám.

Řetězec

Je reprezentován třídami `PNVMStringHead` a `PNVMStringTail`. Implementuje uživatelský datový typ *String*. Z implementačního hlediska je velmi podobný poli a n-tici neboť krom všech obecných atributů (viz 4.1.1) nemá žádný další. Jeho položkami jsou znaky. V prostředí *Squeak* je znak reprezentován třídou `Character`, zatímco v *C* je to **char**.

Událost

Je reprezentována třídou `PNVMEvent`. Z obecných atributů obsahuje pouze signaturu. Obsahuje hodnotu *token*u, kladné číslo, o které bude počet výskytů navýšen. Dále referenci na instanci, která *token* přijme a index místa, do kterého bude vložen.

Jelikož je kalendář implementovaný nad obousměrně vázaným seznamem, obsahuje třída `PNVMEvent` ještě ukazatele na předchozí a následující událost. Více podrobností o kalendáři bude uvedeno níže (4.4.2).

Je nutné podotknout, že v současná implementace nijak nevyužívá výhod obousměrně vázáného seznamu a tedy postačující by byla implementace pouze s ukazately na následníka, tak je to u *priority queue* běžné [19].

Poznámky k implementaci

Abstrakce datových typů musí být v současné době implementovány zvlášť pro prostředí *Squeak*, tak i pro *C++* implementaci. Je pravděpodobné, že v budoucnu se veškerý kód přesune do *Squeak*u, neboť udržovat více implementací stejného nebo podobného kódu je velkou překážkou pro efektivní práci.

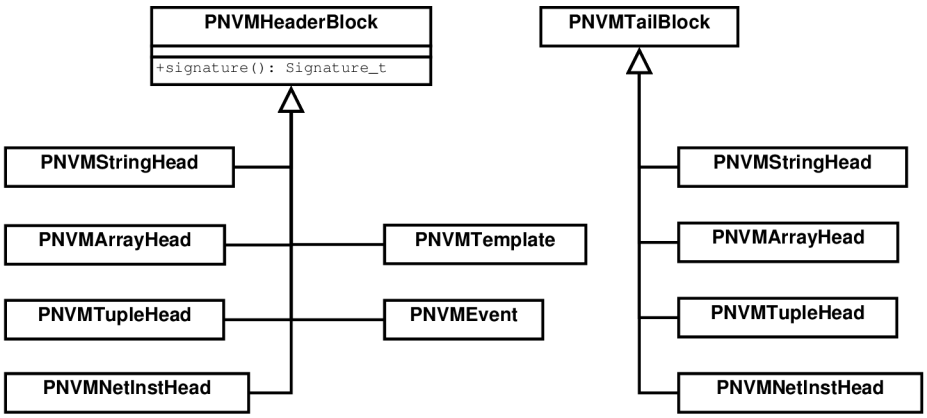
Na druhou stranu není tohoto kódu mnoho. Snahou při návrhu a implementaci bylo co nejvíce kódu generovat a tak dosáhnout co nejmenší redundance. Výsledkem je kompromis, který přispívá k čitelnosti a efektivitě kódu obou implementací. Na 4.4 nalezneme model tříd, který je platný pro prostředí *Squeak*. *C++* implementace je daleko košatější z důvodu využití šablon a principů metaprogramování. Metaprogramování je využito z důvodu přesunutí výpočtů nad typy z *run-time* do *compile-time* a tím je přispěno k lepší efektivitě programu [20]. Nebudeme ji tu však rozepisovat, neboť její model je v principu ekvivaletní.

4.2 Abstrakce pro práci s datovými typy

Aby bylo možné se na objekty odkazovat jedním typem ukazatele a dále uniformně přistupovat k jejich položkám tak, aby bylo možné psát generický kód pro objekty různých typů naráz, byly vytvořeny manipulátory těchto typů. Jsou jimi třídy `PNVMToken` a `PNVMIterator`.

4.2.1 Token

Je obecný ukazatel, který umožňuje uchovávat referenci na libovolný objekt třídy `PNVMHeaderBlock`. Zároveň však slouží k uchování objektu libovolného uživatelského datového typu, tedy i *In-*



Obrázek 4.4: Třídní model datových abstrakcí v prostředí *Squeak*

tegeru.

Jak již bylo uvedeno výše (4.1), jsou celočíselné hodnoty uchovávány v datovém typu shodné velikosti jako ukazatele. Takže na různých platformách bude mít tato abstrakce různé parametry — integer bude mít řádově odlišné rozsahy hodnot — a návrhář sítě s tím musí počítat. Obor hodnot celých čísel již shrnula tabulka 3.1.

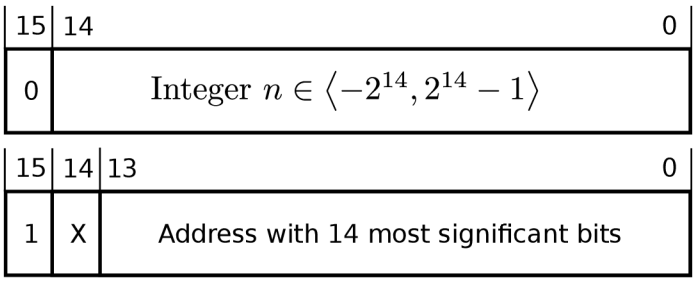
Token kromě samotné hodnoty, která může být buď celým číslem a nebo adresou, nese i informaci o jakou hodnotu se jedná. Tato informace je hodnotou z výčtu:

`TokenTypeInteger` je příznak s hodnotou 0. A říká, že že hodnota je typu *Integer*.

`TokenTypeNetRef` je příznak s hodnotou 2, který říká, že se jedná o instanci sítě.

`TokenTypePointer` je příznak s hodnotou 3, říkající, že hodnota je typu `PNVMHeaderBlock`. Pro bližší určení typu je nutné se přes hodnotu tokenu odkázat na hlavičku *header* bloku a z ní vyčíst signaturu.

Všimněme si, že k uchování této informace (označme ji *ttype*) jsou potřeba nanejvýš dva bity. Toho je využito k úspoře paměti způsobem, že *ttype* zabírá užitečný prostor samotné hodnoty, konkrétně dva nejvíce významné bity. Takže celková velikost *token*u zabírá stejný počet bytů jako jeden ukazatel. Obrázek 4.5 znázorňuje rozložení bitů v tokenu pro všechny hodnoty *ttype* pro platformy s velikostí ukazatele 2 — tedy např. *Arduino*



Obrázek 4.5: Rozložení hodnot v objektu *token* na platfomě *Arduino*

Přístup k hodnotám *token*u řeší funkce

```
// getters
```

```
TokenType_t pnvmTokenType(PNVMToken const &token);
IntegerValue_t pnvmTokenInteger(PNVMToken const &token);
void * pnvmTokenNetRef(PNVMToken const &token);
void * pnvmTokenPointer(PNVMToken const &token);
// setters
PNVMToken & pnvmTokenSetType(PNVMToken &token, TokenType_t type);
PNVMToken & pnvmTokenSetInteger(PNVMToken &token, IntegerValue val);
PNVMToken & pnvmTokenSetNetRef(PNVMToken &token, PNVMHeaderBlock *addr);
PNVMToken & pnvmTokenSetPointer(PNVMToken &token, PNVMHeaderBlock *addr);
```

které mají v prostředí *Squeak* své protějšky ve formě metod třídy `PNVMToken`.

4.2.2 Iterátor

Je neocenitelnou pomůckou pro generický přístup k položkám indexovatelných datových typů. Namísto psaní specifického kódu pro každý typ bloku zvlášť, lze s využitím iterátorů psát jednotný kód. S iterátorem nemusíme brát vůbec v potaz, že se hodnota skládá z více bloků, důležitá je pro nás pouze aktuální pozice od začátku sekvence a znalost, zda jsme u jejího konce sekvence, případně jak daleko od něj se nacházíme.

Squeak naneštěstí nezná tradiční koncept iterátoru, tak je navržen a využíván např. v standardní knihovně *stl* pro *C++*. Implementace plnohodnotného iterátoru podle jejího vzoru ve *Squeak*u by byla příliš složitá, navíc většina operátorů nad iterátory k přístupu a manipulaci s nimi tak, jak jsou používány v *C++* by nešli ve *Squeak*u ani napsat. Proto byla implementována abstrakce iterátoru zcela nová s rozhraním ekvivaletním pro obě prostředí.

Z pohledu *stl* knihovny se jedná o dopředný (forward) iterátor, který však umožňuje emulovat některé vlastnosti iterátorů s náhodným přístupem. Vlastnosti hodnot rozprostřených v jednosměrně linkovaných blokích jej předurčují pouze k použití po způsobu dopředných iterátorů, nicméně náhodný přístup do kolekce je natolik užitečný, že pro tento iterátor byla doplněna podpora skoku na libovolnou pozici v kolekci. Je však třeba brát v úvahu, že náhodné průchody kolekcí nebudou efektivní.

Jeho atributy jsou tvořeny ukazatelem na blok *header*, ukazatelem na současný blok [2] a indexem [3] do kolekce reprezentované hodnotou.

Rozhraní iterátoru

Výpisy 4.1 a 4.2 ukazují nejpodstatnější část rozhraní iterátoru *C++* implementace, které je shodné se *smalltalk*ovským protějškem.

Lze si na první pohled všimnout, že rozhraní je generické díky využití šablon v *C++*. To představuje výraznou úsporu v objemu psaného kódu, především je tak odstraněna redundance. A navíc máme k disposici statickou typovou kontrolu.

Na prvním výpisu si lze všimnout, že iterátor poskytuje mnoho funkcí k zjištění relativní pozice iterátoru vůči významnému bodu a testovací funkce, zda se právě na takovém bodu nachází. To jde proti tvrzení, že algoritmy nemusí vědět, že je kolekce rozdělena do bloků. Tyto funkce jsou důležité pro možnost optimalizace algorimů, a je třeba také brát v potaz, že některé algoritmy přidávající nebo odstraňující bloky potřebují tyto informace pro přesuny položek mezi sousedícími bloky.

[2] tím může být blok *header* i *tail*

[3] počítaným od nuly

Listing 4.1: Rozhraní třídy `PNVMIterator` pro *C++* implementaci

```
// position introspection
template <typename T>
    bool pnvmIterAtStart(PNVMIterator<T> const & i);
template <typename T>
    bool pnvmIterAtEnd(PNVMIterator<T> const & i);
template <typename T>
    bool pnvmIterAtBlockStart(PNVMIterator<T> const & i);
template <typename T>
    bool pnvmIterAtBlockEnd(PNVMIterator<T> const & i);
template <typename T>
    bool pnvmIterAtLastBlock(PNVMIterator<T> const & i);
template <typename T>
    int pnvmIterBlockIndex(PNVMIterator<T> const & i);

// attribute access
template <typename T>
    void * pnvmIterCurrentBlock(PNVMIterator<T> const & i);
template <typename T>
    T * pnvmIterHeaderBlock(PNVMIterator<T> const & i);
template <typename T>
    int pnvmIterIndex(PNVMIterator<T> const & i);
```

Listing 4.2: Rozhraní třídy `PNVMIterator` pro *C++* implementaci

```
// movement
template <typename T>
    typename T::item_type_t & pnvmIterNext(PNVMIterator<T> & i);
template <typename T>
    PNVMIterator<T> & pnvmIterNextBlock(PNVMIterator<T> & i);
template <typename T>
    PNVMIterator<T> & pnvmIterSeekEnd(PNVMIterator<T> & i);
template <typename T>
    PNVMIterator<T> & pnvmIterSeekStart(PNVMIterator<T> & i);
template <typename T>
    PNVMIterator<T> & pnvmIterSeek(PNVMIterator<T> & i, int index);

// item access
template <typename T>
    typename T::item_type_t & pnvmIterSetItem(
            PNVMIterator<T> & i, typename T::item_type_t & value);
template <typename T>
    typename T::item_type_t & pnvmIterValue(PNVMIterator<T> & i);
template <typename T>
    typename T::item_type_t & pnvmIterWrite(PNVMIterator<T> & i,
            typename T::item_type_t & item);
```

Druhý výpis 4.2 deklaruje funkce rozhraní iterátoru pro posun v kolekci a přístup k prvkům. Názvy jednolivých funkcí jsou dostatečně výstižné a není třeba je dále popisovat. Za zmínku však stojí, že zde chybí inverzní operace k posunu v před o jednu položku a o jeden blok. To klade tlak na psaní kódu využívajícího iterátoru jako dopředného, v opačném případě by se degradovala efektivita algoritmu. Nicméně pohyb v rámci bloku je otázkou změny indexu a je tedy efektivní i pro posun vzad. Algoritmy které tak chtějí činit však musí použít funkce pnvmIterSeek.

Přístup k položkám kolekce opět zahrnuje typovou kontrolu, která je užitečná pro ověření generovaného kódu ještě před spuštěním simulátoru.

Generický iterátor

Předvedené rozhraní je platné pro všechny indexovatelné typy, zároveň je však specializováno pro obecné bloky `PNVMHeaderBlock` a `PNVMHeaderTail`. Díky tomu je možné psát obecný kód nad jakýmkoliv objektem bez znalosti jeho typu. Není to však doporučené použití, protože volané funkce jsou podle signatury objektu předávány správnému typu a to znamená značnou režiji. Proto kód, který ví, nad jakým typem kolekce operuje by měl korektně deklarovat typ iterátoru, aby byla použita správná specializace.

Jak již bylo uvedeno dříve (viz 2.1.5), je pro deklaraci typu v těle zprávy nutno použít direktivu:

```
<var: #iter type: 'PNVMIterator'>
```

Tato způsobí, že iterátor `iter` bude pracovat nad obecnou kolekcí a operace budou neefektivní. Pro deklaraci specifického typu iterátoru lze použít jména typů:

- `PNVMStringIter`
- `PNVMArrayIter`
- `PNVMTupleIter`
- `PNVMPlaceIter`
- `PNVMNetInstIter`

V prostředí *Squeak*u používáme pouze jednu třídu, a to `PNVMIterator`. Tyto deklarace jsou proto důležité pouze pro generovaný kód.

Limity současné implementace

Přístup k prvkům nelze uskutečnit v *C++* implementaci s obecným iterátorem [4] a je tedy nutné pro čtení nebo zápis položky iterátor přetypovat na správný typ.

4.3 Správa paměti

Bloky paměti jsou pro hodnoty alokovány po jednom až ve chvíli zaplnění posledního bloku v případě indexovatelného datového typu. Pokud jsou položky odstraňovány, je ověřeno, zda se nezměnil minimální počet bloků nutných k uchování hodnot. Je-li tomu tak, je přebytečný blok odstraněn.

[4] `PNVMIterator`

Dealokaci celých hodnot má na starosti funkce PNVMHeaderBlock » pnvmValueDelete . Je-li volána na neindexovatelnou hodnotu, je odstraněn pouze jeden blok. V opačném případě jsou postupně odstraněny všechny bloky této hodnoty. Než však dojde k samotnému odstranění bloku, jsou prvně zrušeny všechny reference na ostatní objekty.

4.3.1 Čítání referencí

Zde se věnujeme pouze referencovatelným datovým typům a ostatní nebudeme brát v potaz.

Nově vytvořený objekt si při inicializaci nastaví čítač referencí na hodnotu 0, protože dosud není nikým referencován. Jakmile je reference vytvořena, čítač se zvětší o jedničku. Při zrušení reference se neopak sníží. Jakmile při snižování čítač dosáhne hodnoty menší než jedna, je hodnota dealokována.

Čítač referencí je přítomen v každém *header* bloku referencovatelného objektu. K jeho manipulaci je třeba použít metod:

```
PNVMHeaderBlock » pnvmValueReference.
PNVMHeaderBlock » pnvmValueUnreference.
```

Tyto metody by však neměly být používány přímo. Ke správě referencí by měl být výhradně používáno objektu *token*.

Token jako reference

Při určitých operacích nad objektem typu `PNVMToken` se mění čítač referencí odkazovaného objektu. Zde si popíšeme v jakých případech se tak děje a jak to využít v náš prospěch.

Vytvoření reference Při vytvoření nového *token*u nebo přiřazením adresy bloku voláním jedné z níže uvedených funkcí je zvýšen čítač referencí akceptovaného objektu. Pokud *token* již referencuje hodnotu jinou, je její čítač před přiřazením nové hodnoty snížen.

```
// creates new token
PNVMToken pnvmNewNetRef(PNVMHeaderBlock * addr);
PNVMToken pnvmNewPointer(PNVMHeaderBlock * addr);
// clone existing token, making a new reference
PNVMToken pnvmTokenClone(PNVMToken const &t);
// reference new value, unreference previous one
PNVMToken & pnvmTokenSetNetRef(PNVMToken &t, PNVMHeaderBlock *inst);
PNVMToken & pnvmTokenSetPointer(PNVMToken &t, PNVMHeaderBlock *block);
PNVMToken & pnvmTokenAssign(PNVMToken &dest, PNVMToken const &src);
```

Důležitou vlastností *token*u je to, že jeho destrukce nemá vliv na čítač referencí odkazovaného objektu. To je dáno nemožností vynutit zrušení objektu v prostředí *Squeak*, kde má destrukci a volání metody Object » finalize na starosti garbage collector. Proto je nutné „rušit“ *token*y manuálně. K tomu slouží funkce:

```
PNVMToken & pnvmTokenMakeInvalid(PNVMToken &t);
```

Která z *token*u udělá nevalidní hodnotu, přičemž je zrušena jakákoliv reference na případný objekt.

Stejného výsledku dosáhneme přiřazením *token*u hodnoty *Integer*, změnou jeho typu nebo pouze přiřazením jiné adresy. Je však důležité používat k tomuto účelu výše uvedené funkce.

4.4 Globální proměnné a pole

Kromě dynamicky alokovaných objektů se simulátor neobejde bez statických tabulek, globálních proměnných a vyrovnávacích pamětí. Ty zabírají nezanedbatelnou část paměti. Zde si uvedeme jejich výčet a význam.

4.4.1 `netTemplateTable`

Je tabulka o kapacitě `MaxNetTemplates` obsahující ukazatele na šablony sítě. Jedná se o proměnnou třídy PNVMObjectMemory, která je přeložena jako globální, statická proměnná vygenerovaného pluginu (viz deklarace globálních proměnných 2.1). V něm je deklarována jako:

```
static PNVMTemplate *netTemplateTable[MaxNetTemplates];
```

Jedná se o databázi všech šablon simulátoru a zároveň jedinný přístupový bod k instancím sítí, které jsou prováděny. Volné sloty této tabulky mají hodnotu NULL. Počet šablon uložených v databázi je uložen v globální proměnné `numberOfNetTemplates`.

4.4.2 Kalendář

Je prioritní frontou nad obousměrně vázaným seznamem událostí, jenž byly popsány výše (4.1.4). Pro přístup k nim je nutné mít právě dva ukazatele — na začátek a na konec. Těmi jsou opět globální proměnné definované třídou `PNVMObjectMemory`:

```
static PNVMEvent *calendarHead;
static PNVMEvent *calendarTail;
```

Jednotlivé události jsou řazeny pouze podle času a pořadí vložení. Události s menší hodnotou atributu `time` jsou řazeny dříve (mají větší prioritu). Pokud dvě události mají shodný čas spuštnění, je dříve vložená událost řazena před později vloženou.

Je-li kalendář prázný, oba ukazatele mají hodnotu `NULL`.

4.4.3 `netTemplateCode`

Byla zmíněna již u popisu šablon (4.1.4). Je to pole znaků deklarované jako

```
static char netTemplateCode[NetTemplateCodeSize];
```

kam se odkazují objekty šablon na začátky svých kódů. Při načítání nové šablony je její ofset kódu nastaven na počet aktuálně zabraných bytů tohoto pole. To znamená, že kódy šablon jsou v této paměti řazeny těsně za sebou bez jakéhokoliv oddělovače pro úsporu místa. Problém nastává v případě, že šablona je z databáze odstraněna. Kódy šablon následující za kódem šablony odstraněné se musí přesunout na její místo dříve, než se začne nahrávat šablona nová.

Tento princip přesouvání byl zvolen záměrně, aby se předešlo fragmentaci paměti kódu způsobené odstraňováním šablon. Předpokladem je, že k této operaci nedochází často. V opačném případě by tato operace měla významný dopad na výkon simulátoru.

4.4.4 `inputBuffer` a `outputBuffer`

Jsou vyrovnávací paměti pro interakci s okolím simulátoru. Jsou deklarovány jako:

```
static PNVMArrayHead *inputBuffer, *outputBuffer;
```

využívají tedy k uložení hodnot abstrakce typu pole a nejedná se tedy o buffery omezené velikosti. V implementaci pro *Squeak* jsou součástí třídy PNVM. Obsahují *tokeny*, které jsou určeny pro zpracování simulátorem v případě `inputBuffer` nebo pro serializaci na výstup v druhém případě.

4.4.5 Ostatní globální proměnné

Zbývá nám představit neméně důležité proměnné simulátoru *PNVM*. Jsou jimi:

`currentTime` aktuální reálný čas v milisekundách. To, zda se jedná o čas relativní vůči začátku simulace nebo zda jde o absolutní systémový čas je ponecháno na implementaci pro konkrétní platformu. Program `pnvma` používá uplynulý čas od posledního restartu zařízení, zatímco aplikace `pnvmx86` používá systémový čas získaný voláním clock_gettime() definovaného standardem `POSIX.1-2001`. Podobně je tomu v prostředí *Squeak*, kde se čas získává zasláním zprávy now třídě DateAndTime.

`nothingChanged` je proměnná typu bool říkající, zda během posledního kroku simulátoru došlo ke změně. Je to atribut třídy PNVM.

`stepCounter` je čítač provedených kroků od spuštění simulátoru.

Kapitola 5

Interpret

Je označení pro kód *plug-in*u generovaný z prostředí *Squeak* mající za úkol vykonávat kód šablon sítí nad jejich instancemi. Operuje nad rozhraním objektové paměti (viz kapitola 4) a se svým okolím komunikuje prostřednictví zasílání zpráv přes vstupní a výstupní vyrovnávací paměti.

Nejedná se však o samostatný objekt schopný běhu sám o sobě. Je nutné jej volat v cyklu hlavního programu, pro který poskytuje určité rozhraní (viz 5.2).

Celá simulace je rozdělena mezi časově omezené, jednoznačně ohraničené úseky vykonávání pojmenované kroky.

Konvence v této kapitole V textu níže se budeme na odkazovat první instanci první šablony identifikátorem `platform`. Tato instance hraje důležitou úlohu, která bude blíže popsána v sekci 5.2.1.

5.1 Krok

Je konečná sekvence elementárních operací, která vykonává simulační program popsaný šablonami *RefNets*. Jeho opakovaným, postupným vykonáváním se v čase mění vnitřní stav interpretu, jenž reprezentuje stav modelovaného systému.

5.1.1 Stav interpretu

Je reprezentován objekty databáze šablon sítí, instancemi sítí, událostmi kalendáře, vstupní a výstupní vyrovnávací pamětí. Dojde-li ke změně kteréhokoliv jmenovaného objektu, chápeme to, jako změnu stavu celého interpretu.

Stav instance sítě zahrnuje stav všech jejich míst a z nich referencovaných objektů.

Informace, že ke změně došlo, je důležitá pro hlavní program interpretu a je proto uchovávána ve zvláštní proměnné `nothingChanged` zmíněné již v předchozí kapitole 4.4.5. Tato informace je platná od doby první změny stavu až do začátku nového kroku.

5.1.2 Elementární operace

Následuje popis sekvence elementárních operací tak, jak jsou prováděny za sebou během vykonávání jednoho kroku.

1. smazání příznaku poslední změny — nothingChanged := **true**.

2. zpracování vstupu a výstupu
3. aktualizace času
4. zpracování naplánovaných událostí v kalendáři
5. vyhodnocení přechodů všech instancí
6. úklid v databázi šablon a paměti kódu

Zde je vidět, že čas potřebný k provedení jednoho kroku je přímo úměrný počtu přechodů šablon a instancí v objektové paměti. Časová náročnost provedení ostatních operací bude v porovnání s tímto zanedbatelná za předpokladu, že síť intenzivně nekomunikuje se svým okolím zasíláním zpráv, které mohou způsobit zahlcení vyrovnávacích pamětí.

Zpracování vstupu a výstupu

Spočívá v průchodu vstupního bufferu v pořadí, v jakém byly obsažené zprávy přijaty a voláním *downlinku* `platform:input(x)` pro každou z nich. Označení `platform` zde reprezentuje první instanci prvně nahrané šablony v databázi šablon, jenž je vytvořena během inicializace programu simulátoru. V případě, že přechod `uplink` volané sítě uspěl, je zpráva z bufferu odstraněna a příznak poslední změny je nastaven.

Zprávy, pro které volání *downlink*u `platform:input(x)` neuspělo v bufferu zůstávají a v příštím kroku je bude čekat stejná procedura.

Dále je právě jednou volán *downlink* `platform:downlink(x)` na jehož první parametr x se naváže jakákoliv výstupní zpráva sítě `platform` v případě, že přechod uspěje. Tato hodnota je následně zařazena do výstupní vyrovnávací paměti reprezentované proměnnou `outputBuffer`.

Aktualizace času

Čas simulace je uchováván proměnnou `currentTime` rovněž uvedenou v předchozí kapitole 4.4.5. Je aktualizován pouze jednou během kroku simulace a to těsně před zpracováním naplánovaných událostí. Simulační čas musí být konstatní během provádění události [12].

Zpracování událostí

Všechny události, jejichž čas provedení je menší, než hodnota `currentTime` jsou z kalendáře vyjmuty a hodnoty jejich tokenů jsou vloženy do míst instancí specifikovaných atributy události. Pro její detailní popis lze nahlédnout na 4.1.4.

Zpracovány jsou v sestupném pořadí podle priority. Ačkoliv vzhledem k tomu, že jejich zpracování negeneruje ihned žádnou akci ani provedení přechodu, tak na jejich pořadí zpracování nezáleží. V budoucnou je však možné očekávat další typy událostí, kde jejich pořadí vyhodnocení bude hrát roli.

Úklid v databázi šablon a paměti kódu

Během vykonávání přechodů sítě je možné provést operaci *unload* vyhodnocením operátoru `u`, popsaným v sekci 3.1.1. Dojde-li k tomu, vytvoří se v paměti kódu prázdné místo, které by způsobilo fragmentaci kódu později vložených šablon. Tomu je zabráněno tím, že veškerý kód následující po kódu odebrané šablony je posunut o jeho velikost.

Zároveň v tabulce šablon jsou všechny záznamy následující po uvolněném slotu posunuty o index dolů. To zajistí zrychlení procházení této tabulky v následujících krocích simulátoru, neboť je procházeno pouze $x \geq$ `numberOfNetTemplates` slotů tabulky, kde:

$$x = \max\{i | \text{slot}_i \neq 0, \quad 0 \leq i < \text{MaxNetTemplates}\} \tag{5.1}$$

$slot_i$ by v zápisu jazyka C odpovídal: netTemplateTable[i].

5.1.3 Volné a vázané proměnné

Proměnné a parametry přechodů jsou simulátorem reprezentovány objekty typu `PNVMToken` (viz 4.2.1). Kde vázanou poznáme tak, že je validní, tedy zaslání zprávy PNVMToken » pnvmTokenIsValid *token*u vrátí `true`. Pro volnou přoměnnou je tomu naopak.

Seznam proměnných přechodu je potom během jeho vyhodnocování uložen jako n-tice (viz 4.1.4), kde jednotlivé *token*y jsou postupně vázany při unifikaci a uvolňovány při zpětném navracení. Objekt této n-tice je během zpracování přechodů v rámci jednoho kroku neustále přítomen, pouze se mění jeho kapacita v závislosti na počtu parametrů a lokálních proměnných v hlavičce přechodu.

5.1.4 Unifikace

Je algoritmus nebo proces hledající odpověď na otázku, zda existuje substituce za proměnné ve dvou výrazech A a B tak, aby se vyhodnotily jako totožné [21]. Pracuje následovně:

1. jsou-li A i B konstanty, musejí být totožné
2. je-li A volná proměnná a B libovolný objekt, proveď A/B.
3. je-li B volná proměnná a A … — obdoba předchozího
4. jsou-li A i B strukturované objekty stejného typu, pak je lze unifikovat pokud všechny jejich vzájemně si odpovídající položky lze unifikovat stejným algoritmem.

Unifikace v simulátoru *PNVM* podporuje pouze první tři zmíněné případy. Už samotná syntaxe jazyka *PNAL* neumožňuje zápis strukturovaného objektu přímo v elementu přechodu. Tu je sice možné vytvořit evaluací operátoru v akcích případně strážích, ale stále bude chápána algoritmem unifikace jako konstanta. Strukturovaný objekt tedy v našem případě nemůže obsahovat volné proměnné.

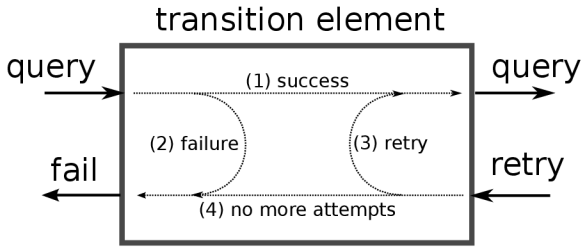
To je sice nedostatek z pohledu možností zápisu, ale vyjádřovací síla zůstává stejná. Pouze je nutné výrazy rozepsat do více oddělených příkazů. Výhodou tohoto přístupu je, že autor sítě má plnou kontrolu nad specifikací pořadí provádění unifikace případně vyhodnocování výrazů a může tak docílit větší efektivity simulace.

5.1.5 Zpětné navracení

Příkazy přechodu jsou prováděny rekurzivním zanořováním až do doby selhání nebo úspěšného provedení celého přechodu. Prohledávání možností unifikace tu probíhá podobně jako v *Prologu* [13] — rekurzivním zanořováním do hloubky. Všechny typy příkazů mají vliv na stav simulátoru. Při úspěšném provedení jenoho příkazu je stav modifikován a změna je platná pro všechny následující příkazy přechodu. V případě úspěšného provedení celého přechodu, jsou změny platné po zbytek simulace.

Některé typy elementů mají více možností provedení. Při první návštěvy příkazu je první nalezená možnost použita, aplikována na současný stav simulátoru a vyhodnocování pokračuje následujícím elementem. Pokud však následující element selže, simulátor se vrátí zpět, aby nalezl jinou možnost provedení tohoto příkazu, pokud ji nalezne, znovu modifikuje svůj stav a pokračuje vykonáváním dalších příkazů.

Tento proces znázorňuje obrázek 5.1. Příkaz je tu znázorněn jako blok s dvěma vstupy a dvěma výstupy. Dále zobrazuje 4 možné průchody tímto blokem. Vstup *query* značí první



Obrázek 5.1: Element přechodu se všemi možnými průchody

návštěvu elementu simulátorem, zde mohou nastat dva odlišné běhy:

1. V prvním případě se úspěšně provede unifikace a simulátor pokračuje výstupem *query* k dalšímu elementu přechodu.

2. V druhém případě unifikace selhala a neexistuje žádná jiná možnost provádění. Simulátor se vřátí výštupem *fail* tohoto bloku do bloku předchozího jeho vstupem *retry*. A pokusí se nalézt jinou možnost provedení tak, aby se opět vrátil naším vstupem *query*.

Druhým vstupem našeho příkazu je *retry*. Zde simulátor nejprve navrátí svůj vnitřní stav do podoby před vstupem do současného bloku a následně se pokusí nalézt jinou možnost provedení. Následuje popis zbylých dvou možností průchodů:

3. Je nalezena možnost, která dosud nebyla zkoušena. Je modifikován stav interpretu a znovu se vrácíme výstupem *query* na vstup následujícího bloku.

4. Žádná další možnost provedení elementu neexistuje, vracíme se k předchozímu bloku výstupem *fail*.

Specifika konkrétních elementů

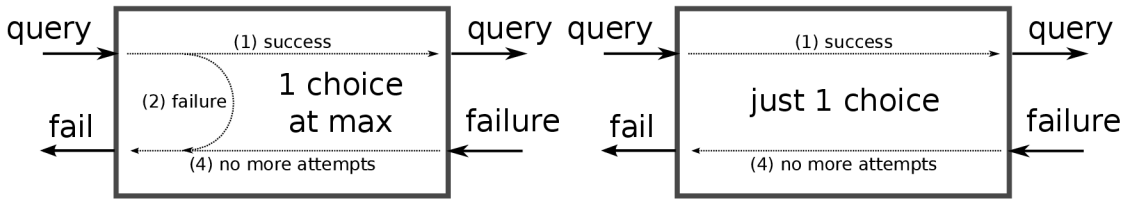
Elementy přechodu můžeme rozdělit na několik typů z pohledu možnosti selhání a zpětného navracení:

- element je podmínkou provedení přechodu, tzn. může selhat a má více možností provedení (elementy `P` a `D`).
- element je podmínkou provedení přechodu, ale má maximálně jedno možné provedení — element `G`.
- element není podmínkou provedení přechodu, má maximálně jedno možné provedení — element `A`.

- element není podmínkou provedení přechodu, má právě jeden způsob provedení a tedy nemůže selhat (elementy `O` a `Y`)

Pro elementy `P` a `D` jsou možné všechny varianty běhu na obrázku 5.1. Element `G` má jedno možné provedení, výsledek vyhodnocení jeho výrazu je ověřen a v případě, že není pravdivý, je výsledkem průchod 2. Element `A` může selhat z důvodu aplikace operátoru na invalidní vstup, např.: zápis hodnoty na pin, kdy se pokoušíme indexovat jiným typem než *Integer*. Proto jsou pro něj, stejně jako pro element `G`, platné průchody 1, 2, a 4 — obrázek 5.2a.

Pro všechny ostatní elementy jsou platné pouze dva průchody, a to 1 a 4. Elementy `O` a `Y` sice nemohou selhat sami o sobě, ale je potřeba brát v úvahu, že v sekvenci příkazů mohou předcházet elementu `A`, který již selhat může. Průchody znázorňuje obrázek 5.2b.



(a) Nanejvýš jedna možnost provedení (b) Právě jedna možnost provedení

Obrázek 5.2: Průchody elementy v závislosti na počtu možných provedení

Historie

Aby mohl být stav simulátoru obnoven při zpětném navracení před znovu-provedením elementu přechodu, je nutné přechozí stav buď:

- celý uložit a při zpětném navracení jej obnovit;
- nebo uchovat pouze inkrementální změny a definovat pro ně funkce, které provedou změnu inverzní.

Simulátor *PNVM* používá právě druhého zmíněného přístupu. Historie je pole n-tic, z nichž každá reprezentuje jednu akci. Je v simulátoru předávána ve formě iterátoru do tohoto pole ukazujícího na konec. Příkaz měnící stav simulátoru si na začátku zálohuje pozici iterátoru, zapíše s jeho pomocí jednu či více akcí na konec a iterátor předá dalšímu příkazu v pořadí. Pokud se k nám vrátí simulátor vstupem *failure*, invertujeme všechny námi provedené změny v opačném pořadí až po námi zálohovanou pozici iterátoru a pokračujeme buď průchodem 3 nebo 4 z obrázku 5.1.

Implementace historie provedených akcí jako pole přináší spoustu výhod. Dává nám prostředek, jak od sebe oddělit aplikaci změny stavu od aplikace funkce inverzní, což zpřehledňuje kód. Dealokace historie po úspěšném provedení přechodu může probíhat na jednom místě a v jednom průchodu jsou tak efektivně odstraněny všechny záznamy akcí díky funkční správě paměti.

5.1.6 Vyhodnocení přechodů

Je rekurzivní unifikace elementů přechodů prováděna iterativně přes všechny instance sítí šablon v databázi. Jsou zde prováděny pouze obecné přechody. Jedná se o výpočetně nejnáročnější část provádění kroku.

Z hlediska efektivity provádění je nejvýhodnější iterovat přes instance sítí uvnitř cyklu iterujícím přes přechody šablony. V pseudokódu to znamená:

```
for net_template in netTemplateTable:
  offset = net_template.transitions   # begin on first transition element
  while offset > 0:     # process all transitions of net_template
    for instance in net_template.instances:
      if process_single_transition_on(offset, instance):
        nothingChanged = False
    # skip to the next transition
    offset = offset_of_next_transition(offset)
```

Pokud se podaří kterýkoliv přechod úspěšně provést, počítá se to jako změna stavu a odpovídající příznak je nastaven (řádek 6).

5.1.7 Kanály

Slouží k synchronní komunikaci mezi sítěmi. Elegantně tak implementuje koncept „nahrazení přechodů“ používaný v hierarchických petriho sítích [17].

Dále jsou využity samotným simulátorem *PNVM* k interakci s *platformou* voláním jejích *uplinků* `:input(x)` a `:output(x)` popsaných v sekci 5.1.2. Jedná se o přechody, které mají navíc parametry. Interně je jejich vyhodnocování zcela shodné s prováděním obyčejných přechodů popsaným v sekci 5.1.6.

5.2 API

Vygenerovaný *plug-in Squeak*u poskytuje své *API* prostřednictvím exportovaných funkcí hlavnímu modulu k použití a očekává od něj, že modul správně zinicializuje, nahraje šablonu platformy, vytvoří její instanci a periodicky bude provádět kroky simulátoru, mezi kterými bude zpracovávat vstupní a výstupní zprávy.

Výpis 5.1 zahrnuje nejdůležitější funkce tohoto rozhraní. Jsou uvedeny v pořadí, v jakém mají být použity. Funkce initializeModule() na řádku 2 musí být zavolána jako první pro inicializaci všech statických proměnných modulu. Po ní následuje nahrání šablony instance platformy (viz 5.2.1 níže) — řádek 4. Která musí být instanciována (rádek 5).

Řádky od 7 dále uvádějí nejdůležitější funkce pro přístup k vstupní a výstupní vyrovnávací paměti. Tyto funkce zároveň obstarávají konverzi hodnot *token*ů na textovou podobu a zpět (viz 3.5). Jakmile program simulátoru obdrží vstupní zprávu v podobně textového zápisu *tokenu*, předá ji *plug-in*u jednou z *input* funkcí. Ty vracejí hodnotu `true` v případě úspešné konverze na *token*.

Přístup ke kalendáři (od řádku 12) je omezen prakticky jen na kontrolu prázdnosti a přístup k první naplánované události. Toho může hlavní program využít k uspání do doby, na kterou je událost naplánována.

Pro samotnou simulaci jsou nejdůležitějšími funkce počínaje řádkem 16, kde:

1. první slouží k ověření, zda v posledním kroku simulace došlo ke změně stavu simulátoru (viz 5.1.1). Což může být v bráno v potaz při rozhodování o ukončení simulace.

2. druhá slouží k samotnému vykonání kroku simulace, který byl podrobně popsán výše 5.1.

Listing 5.1: Rozhraní *plug-in*u

```
// initialization
void initializeModule(void);

int primitiveLoadNetTemplate(char const *msg);
PNVMNetInstHead * primitiveInstantiateTemplate(int tmplId);

// input/output
bool primitiveInputCStr(char const *msg);
bool primitiveOutputBufferEmpty(void);
char * primitiveOutputPopCStr(void);

// calendar access
bool primitiveCalIsEmpty(void);
PNVMEvent * primitiveCalFront(void);

// step
int primitiveNothingChanged(void);
int primitiveStep(void);

// plugin clean up
void primitiveCleanupModule(void);
```

5.2.1 Platforma

Je první instance první nahrané šablony simulátoru. Jsou na ni kladeny následující požadavky:

- musí definovat *uplink* přechod s názvem `:input` akceptující právě jeden parametr. Ten bude volán v okamžiku přijetí vstupní zprávy programu v případě úspěšného převodu na hodnotu *token*u.
- musí definovat *uplink* přechod s názvem `:output` akceptující právě jeden parametr, který unifikuje s výstupní zprávou v případě úspěšného provedení přechodu.

Příklad této šablony byl již uveden v sekci 3.4. O něco složitější síť je výchozí platformou pro aplikaci `pnvmx86`.

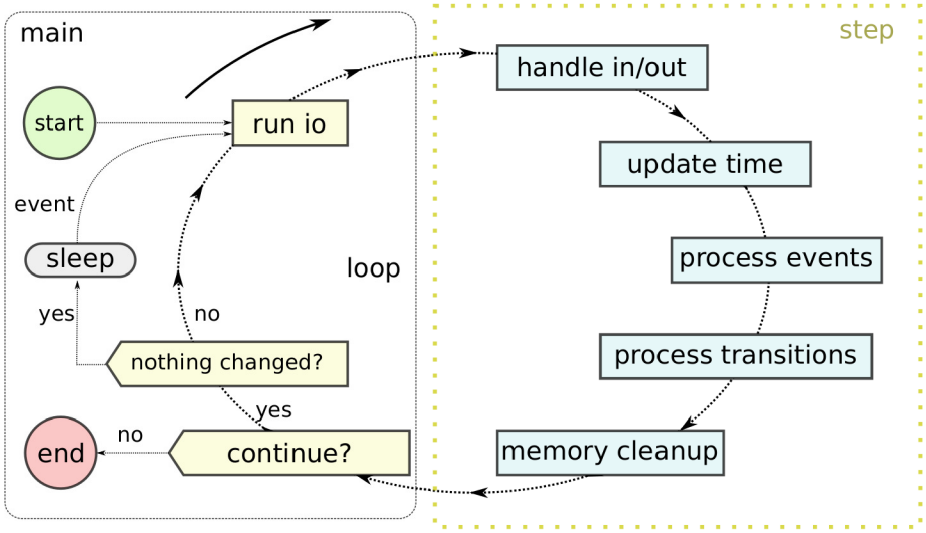
5.3 Hlavní program

V zásadě pouze inicializuje *plug-in* simulátoru prostřednictvím jeho rozhraní a následně ve smyčce volá proceduru kroku.

To, zda je informace o tom, že během posledního kroku simulace došlo ke změně vnitřního stavu, využita k uspání programu do příchodu první události, je ponecháno na konkrétní implementaci. Aplikace `pnvmx86` se uspí do příchodu první události na deskriptoru souboru příchozích textových zpráv nebo souboru pro příjem hodnot vstupních pinů s timeoutem nastaveným na rozdíl naplánovaného spuštění následující události v kalendáři a současné hodnoty času.

Hlavní program `pnvma` pro *Arduino* v současné době uspání nepodporuje, namísto toho aktivně cyklí. Avšak díky jednoduchosti modulu hlavního programu je úprava pro uspání triviální.

Smyčka programu je znázorněna obrázkem 5.3.



Obrázek 5.3: Hlavní smyčka programu

5.4 Experimenty se simulátorem

Je několik vlastností simulátoru, které jsou pro nás zajímavé a zároveň snadno měřitelné. Experimentováním se simulátorem, získáním statisik a jejich analýzou si lze udělat představu o tom, kde simulátor najde uplatnění, jaká jsou jeho slabá místa, jakých konstrukcí se při psaní modelů sítí vyhnout apod.

My si zde předvedeme měření závislosti délky kroku na počtu položek v místě a závislost délky paralelního výpočtu na počtu instancí, které se na něm podílí.

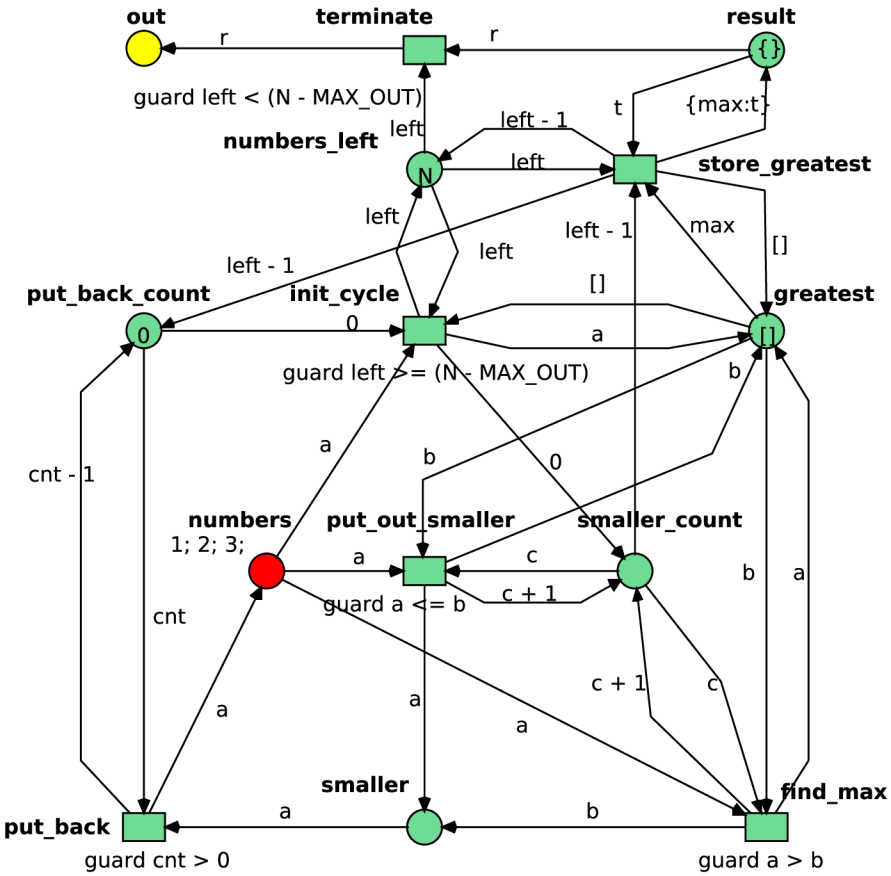
Testovací sestava Je běžný desktopový počítač s čtyř-jádrovým procesorem `Intel Core i5-2540M CPU@2.60GHz` a 8GB operační paměti. Testováno bylo na operačním systému *GNU/Linux* v distribuci *Fedora* 18 s jádrem `3.8.11-200.fc18.x86_64`. Testováno bylo s programem `pnvmx86` zkompilovaným kompilátorem `gcc` ve verzi `4.7.2 20121109 (Red Hat 4.7.2-8)` s parametry kompilace:

```
g++ -Wall -Wextra -I../includes -I. -pthread -O2 -DDEBUG #...
```

Logování bylo povoleno pouze pro chybová hlášení.

5.4.1 Řazení čísel

Pro měření závislosti délky kroku na počtu položek byl vybrán algoritmus řazení čísel. Model sítě v jeho grafické podobě lze nalézt na obrázku 5.4.



Obrázek 5.4: Model sítě `sort` pro řazení čísel

Princip algoritmu Důležitým místem sítě je místo *numbers* vyplněné červenou barvou, obsahující N náhodně vygenerovaných čísel v náhodném pořadí. Jedno číslo může být zastopeno vícekrát. Tyto čísla jsou sítí řazena a shromažďována v místě *result* jako pole seřazené od nejvyššího prvku po nejnižší. Jakmile počet položek v tomto poli dosáhne konstanty `MAX_OUT`, je pole z místa *result* přesunuto do místa *out* a algoritmus se ukončí (žádný přechod není proveditelný). Kritickým místem algoritmu je vybírání položek přechodem `find_max`, který se snaží v místě *numbers* nalézt hodnotu větší, než aktuálně největší nalezená položka umístěná v místě `greatest`. To při větším počtu prvků v daném místě způsobí v každém kroku simulátoru velkou režiji způsobenou zpětným navracením při zkoušení dalších možností výběru čísla, které by bylo větší, než dosud nalezené.

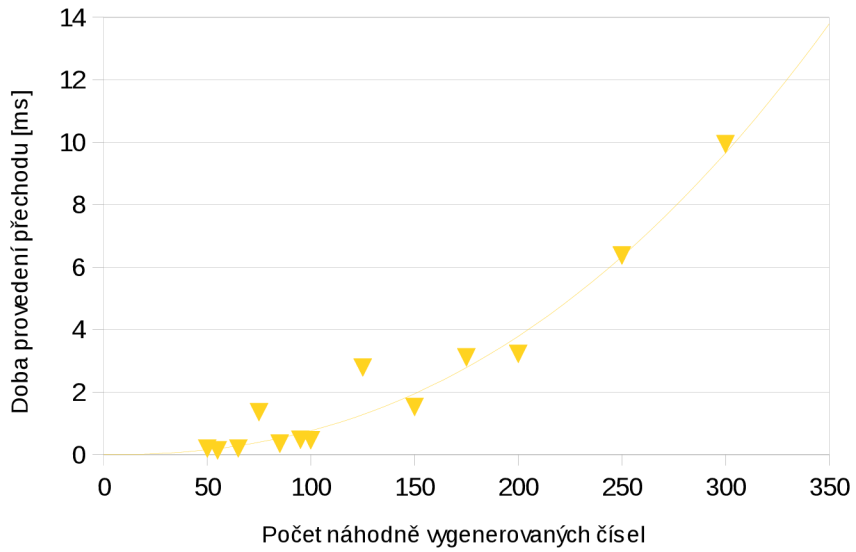
Způsob testování Simulátor `pnvmx86` byl spuštěn s výchozí platformou, které byly předány příkazy pro nahrání šablony `sort` [1] a následně pro nahrání její instance [2] s již vygenerovanými hodnotami čísel v místě *numbers*. Kdyby čísla byla zasílána síti předáváním zpráv, významně by to mohlo ovlinit výsledky. Takto je síť připravena prosimulaci během prvních dvou kroků, které ovlivní výsledky minimálně.

Počet čísel N byl zvyšován v rozmezí 50 až 300 s postupně se zvětšujícím inkrementem.

[1] příkazem „load“ platformy — viz 3.4
[2] příkazem „loadinst“

Pro každé N bylo generováno 10 různých náhodných sekvencí čísel a pro každou bylo provedeno 10 měření, jejichž výsledky byly zprůměrovány.

Výsledky



Obrázek 5.5: Průměrná doba provedení přechodu v závislosti na N

Na grafu 5.5 vidíme exponenciální závislost průměrné doby vykonávání přechodu na počtu čísel. Tento průběh je očekávaný, neboť s narůstajícím počtem položek v místě se zvyšuje režije výběru, kdy položky následující za odebranou se musejí o jednu pozici posunout při každém novém pokusu výběru. Navíc u menších sítí se daleko dříve projeví postupný pokles časové náročnosti provedení přechodu, neboť ke konci simulace je poměr položek k výběru vůči původnímu počtu citelnější, než u sítí s více položkami. Tato klesající tendence se pak promítne v celkových výsledcích.

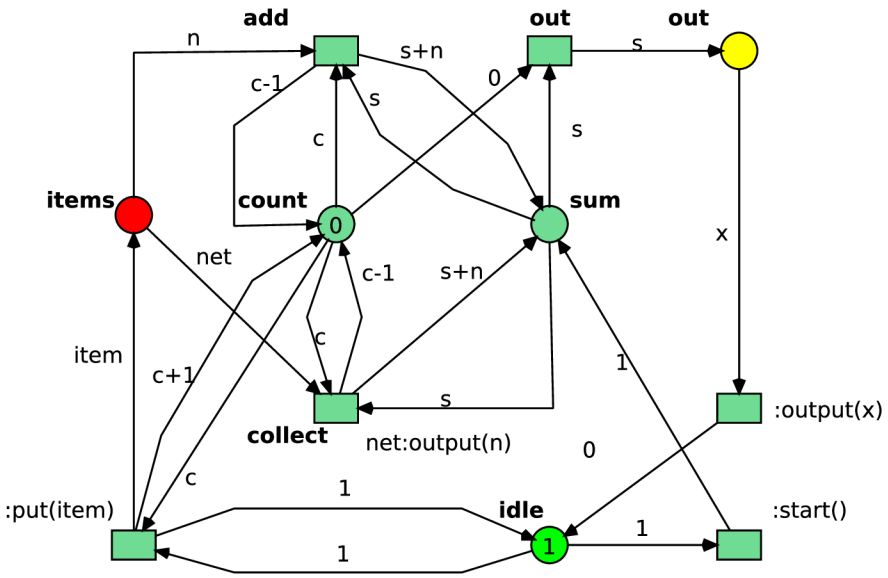
5.4.2 Paralelní suma čísel

Je algoritmus navržený a implementovaný tak, aby otestoval režiji spojenou s během více instancí naráz, které však pracují se stejnou množinou dat. Objem práce je pro každou instanci shodný.

Model sítě lze nalézt na obrázku 5.6.

Princip algoritmu Síť *master* obsahuje v místě *items* K instancí vnořených sítí. Nazvěme je *slaves*. Každá z nich obsahuje ve svém místě N/K čísel, které má za úkol sečíst a uložit do místa *out*. Instance sítě *master* se v každém kroku dotazuje *downlinkem* `net:output(n)` spravovaných *slave* sítí na jejich výsledek. Ty postupně sečte a přemístí do místa *out*.

Podobně jako v předchozím případě je využito výchozí platformy pro nahrání šablony *parallel_sum* a instanciaci všech sítí jedním příkazem „loadinst“.

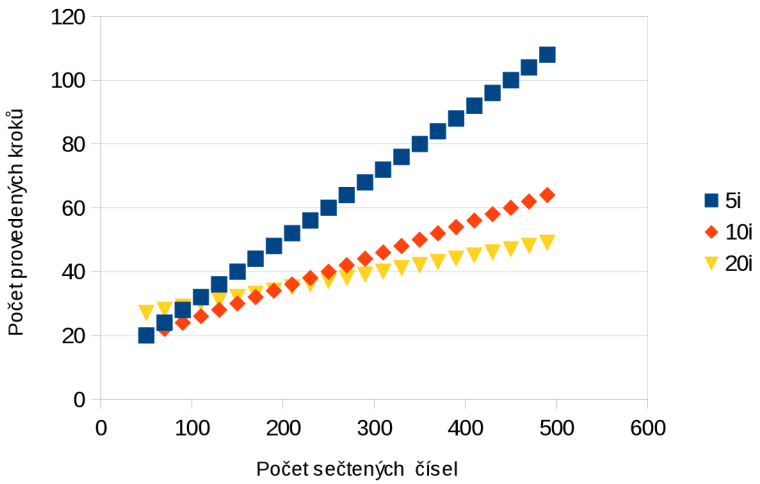


Obrázek 5.6: Model sítě `parallel_sum` pro sumu čísel

Výsledky

Na grafu 5.7 vidíme, že s rostoucím počtem instancí se zvyšuje objem práce vykonaný v jednom kroku a k dokončení celého výpočtu nám tak stačí podstatně méně kroků.

Dále si lze všimnout, že počet kroků je přímo uměrný počtu čísel k seřazení, což je přesně podle očekávání.

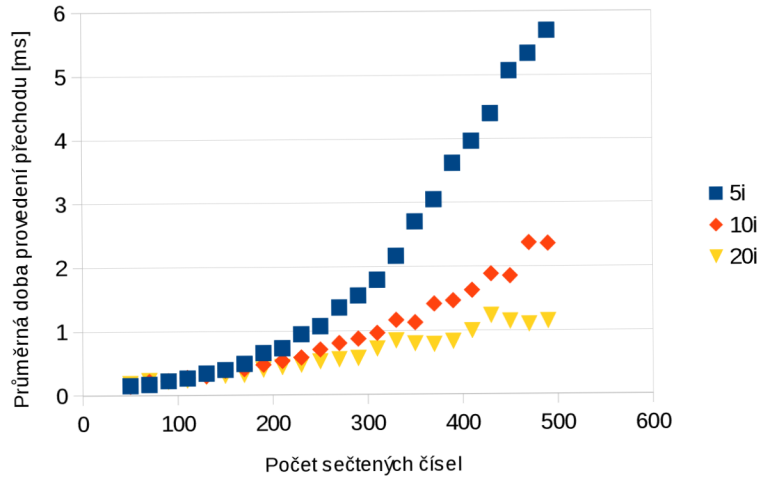


Obrázek 5.7: Počet kroků simulace v závislosti na N

Daleko zajímavější průběh má však závislost doby provedení přechodu na počtu generovaných čísel, zobrazená grafem 5.8. Zde vidíme naprosto opačnou situaci, než bychom očekávali. Totiž, že s přibívajícím počtem instancí se průměrná doba provedení přechodu snižuje. To naznačuje silné rezervy v efektivitě manipulace s bloky. V případě velkého počtu

položek v místě znamená výběr položky (obvykle první) posun všech ostatních na začátek pro zaplnění odebraného místa. Tato režije zcela zasune do pozadí provádění přechodů v násobném množsví v jednom kroku způsobeném navýšením počtu spravovaných instancí.

Je zřejmé, že přístí vylepšení interpretu by se měla zaměřit právě na zlepšení efektivity při práci s rozsáhlými místy.



Obrázek 5.8: Průměrná doba provedení přechodu v závislosti na N

Kapitola 6

Závěr

Referencované Petriho sítě jsou mocným nástrojem pro modelování agentních systémů. Jejich modely stačí na cílových systémech interpretovat bez potřeby dodatečné implementace, pokud je pro ně dostupný interpretr. Aplikace *PNVM* rozšiřuje možnosti přímé interpretace těchto modelů o mikroprocesory *Atmel* a další věstavěné systémy, které však prozatím nebyly testovány.

Současná implementace je velmi jednoduchá a podporuje pouze základní výrazové prostředky referencovaných sítí a jen několik základních typů, z dalších nedostatků jmenujme nemožnost unifikace strukturovaných objektů v zápisu elementů přechodů. Nedokonalost serializace vnořených identických objektů, které se ve výsledku objeví jako separátní. Dalším podstatným nedostatkem je chybějící přísná kontrola zápisu šablon a inteligentnější hlášení chyb, které autora kódu rychle navede k příčině problému.

Tento souhrn neobsahuje zdaleka všechny nedokonalosti a chybějící funkcionalitu interpretu. Je však potřeba brát v úvahu, že se v této verzi interpretu jedná spíše o důkaz použitelnosti konceptu a že interpret petriho sítí pro vestavěné systémy je v praxi implementovatelný, použitelný a dokáže si najít své uplatnění.

Pro širší uplatnění bude nezbytné doplnit a opravit většinu ze zmíněných nedostatků. Avšak v této fázi, kdy existuje funkční prototyp, který dokáže simulovat libovolně složité sítě na několika různých platformách, už nebude problém rychle přidávat interpretu chybějící vlastnosti opravovat chyby.

Shrnutí procesu kompilace To, že je kód psaný ve *Smalltalk*u a překládán do *C* nepředstavuje z hlediska použitelnosti a efektivity generovaného kódu žádný problém. Naopak, je tu řada výhod. To, že je možné kód testovat nezávisle na několika různých platformách, klade větší nároky na robustnost aplikace a správný návrh. Chyby skryté na jedné platformě jsou okamžitě odhaleny na platfomě jiné. Kombinace dynamického, objektového jazyka *Smalltalk*, se staticky typovaným *C++* přináší řadu netušených pozitiv. Díky dynamické povaze *Smalltalku* je většina chyb sice objevena až za běhu, ale s dostupnou dynamickou introspekcí tohoto prostředí je rychle nalezena příčina a odstraněna. Naproti tomu generováním kódu do *C* a pouhým pokusem o jeho překlad je ihned objeveno spousta dalších, *Squaku* ukrytých, chyb. A to bez nutnosti simulátor v přostředí *Squeak*u spustit.

Možnost tohoto „přepínání“ mezi implementacemi umožňuje velmi rychlé nalezení a odstranění příčin problémů.

Stinné stránky současné implementace Největším nedostatkem se z hlediska překladu jeví redundance kódu. Kdy je třeba datové typy a rozhraní implementovat pro obě

prostředí zvlášť. Nicméně *Squeak* je snadno rozšiřitelný a generátor kódu relativně čitelný, proto lze očekávat, že tento problém najde brzy své řešení.

Podstatným problémem se však ukázalo být i použití šablon jazyka *C++*. Při nadměrném používání šablon *C++* je příliš mnoho kódu inlinováno, což sice přispívá k jeho efektivitě, na druhou stranu však aplikace narůstá na objemu [11]. Výsledný slinkovaný program je příliš veliký pro uložení v 32kB paměti *flash* mikroprocesoru *Atmel*. Je tu tedy ještě prostor pro úpravu objektového návrhu a celého rozhraní abstraktních datových typů. Velká paměťová náročnost je citelnou překážkou v migraci na další věstavěné systémy.

Komentář k experimentům Z naměřených závislostí zobrazených na grafech v závěru předchozí kapitoly (5.4) vyplývá, že co se efektivity provádění simulace týče, je tu velký prostor pro optimalizaci. Na vině může být i chybný návrh řešení uložení dat v paměti. Na druhou stranu po stránce redukce nároků na úložný prostor lze už jen s těží dále optimalizovat. To si vybírá daň na efektivitě provádění. A ta už z důvodu zaměření simulátoru na věstavené systémy ani nemůže nijak být závratná.

Literatura

[1] Banzi, M.: *Getting Started with Arduino*. Sebastopol, CA: Make Books - Imprint of: O'Reilly Media, druhé vydání, 2011, ISBN 1449309879, 9781449309879.

[2] Barr, M.; Massa, A.: *Programming embedded systems - with C and GNU development tools: thinking inside the box: includes real-time and Linux examples (2. ed.)*. O'Reilly, 2006, ISBN 978-0-596-00983-0, I-XXI, 1-301 s.

[3] Black, A.; Nierstrasz, O.; Ducasse, S.; aj.: *Squeak by Example*. Gloria Black, 2007, ISBN 9783952334102.

[4] Cabac, L.; Duvigneau, M.; Moldt, D.; aj.: Modeling dynamic architectures using nets-within-nets. In *Proceedings of the 26th international conference on Applications and Theory of Petri Nets*, ICATPN'05, Berlin, Heidelberg: Springer-Verlag, 2005, ISBN 3-540-26301-2, 978-3-540-26301-2, s. 148–167, doi:10.1007/11494744_10.

[5] Christensen, S.; Hansen, N. D.: Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, London, UK, UK: Springer-Verlag, 1994, ISBN 3-540-58152-9, s. 159–178.

[6] Community, S.: SlangBrowser. http://www.squeaksource.com/VMMaker/, [Online; accessed 16-May-2013].

[7] Community, S.: Speech. http://www.squeaksource.com/Speech/, [Online; accessed 6-Jan-2013].

[8] Community, S.: Toothpick. http://www.metaprog.com/Toothpick/, [Online; accessed 16-May-2013].

[9] Community, S.: FFI. http://source.squeak.org/FFI.html, Červenec 2009, [Online; accessed 6-Jan-2013].

[10] Community, S.: VMMaker. http://squeaksource.com/VMMaker.html, Září 2011, [Online; accessed 6-Jan-2013].

[11] Fisher, J. A.; Faraboschi, P.; Young, C.: *Embedded computing - a VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann, 2005, ISBN 978-1-55860-766-8, I-XXVI, 1-671 s.

[12] Fishman, G. S.: *Discrete-event simulation*. London, UK, UK: Springer-Verlag, 2001, ISBN 0-387-95160-1.

[13] Flach, P. A.: *Simply logical - intelligent reasoning by example*. Wiley professional computing, Wiley, 1994, ISBN 978-0-471-94152-1, I-XV, 1-240 s.

[14] Foundation, A. S.: Toothpick. http://www.metaprog.com/Toothpick/, [Online; accessed 16-May-2013].

[15] Greenberg, A. C.: Extending the Squeak Virtual Machine. Technická zpráva, Srpen 2000.

[16] Ingalls, D.; Ingalls, D.; Kaehler, T.; aj.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *In Proceedings OOPSLA '97, ACM SIGPLAN Notices*, ACM Press, 1997, s. 318–326.

[17] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2003, ISBN 9783540609438.

[18] Kummer, O.: Simulating Synchronous Channels and Net Instances. 1998.

[19] Matloff, N.: *Introduction to Discrete-Event Simulation and the SimPy Language*. Únor 2008.

[20] Meyers, S.: *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005, ISBN 0321334876.

[21] Nilsson, U.; Maluszynski, J.: *Logic, programming and Prolog*. Wiley, 1990, ISBN 978-0-471-92625-2, I-XIV, 1-289 s.

[22] Valk, R.: Concurrent object-oriented programming and petri nets. kapitola Concurrency in communicating object petri nets, Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001, ISBN 3-540-41942-X, s. 164–195.

[23] Češka, M.; Marek, V.; Novosad, P.; aj.: Petriho sítě PES – Studijní opora. Prosinec 2009.