SMV VERIFIKACE PLC PROGRAMŮ

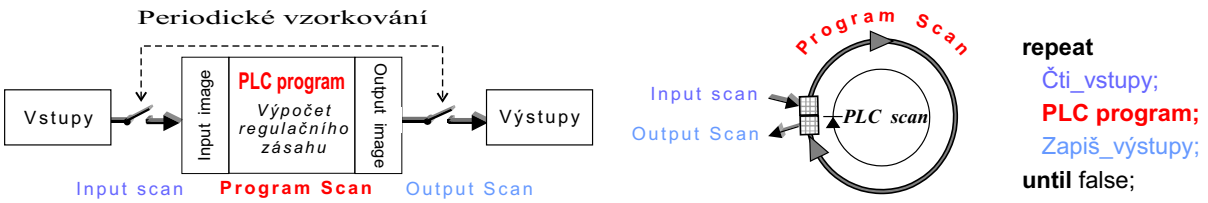
ŠPRDLÍK OTAKAR[1], ŠUSTA RICHARD[2]
Katedra řídicí techniky, ČVUT-FEL, Technická 2, Praha 6,
tel. +420 224 357 359, fax. + 420 224 918 646
email[1]: sprdlo1@fel.cvut.cz, email[2]: susta@control.felk.cvut.cz

Abstract: Programovatelné logické automaty (PLC) tvoří nedílnou stavební prvek pro automatizaci výroby. Zatímco spolehlivost jejich hardwaru je obecně velmi vysoká, verifikace jejich programů je pořád otevřenou otázkou. Článek předkládá postup možné verifikace PLC programů, respektive jejich dílčích částí, pomocí běžně dostupného verifikačního nástroje SMV.
Předložená metoda navazuje na publikovaný APLCTrans algoritmus (Říp 2003), který převádí PLC kód na paralelní přiřazovací příkazy. Algoritmus popsaný v hlavní části článku převádí výstup APLCTrans do specifikačního jazyka SMV, což lze prakticky využít pro ověření dílčích bloků PLC programu či univerzálních knihovních PLC podprogramů. Postup se demonstruje na jednoduchém příkladě spolu s výsledky verifikace. V závěru příspěvku se pak demonstruje podstatně vyšší rychlost verifikace modelu oproti postupu jiných autorů.

Klíčová slova: PLC, programovatelné logické automaty, verifikace, SMV

1 ÚVOD

Označení PLC (Programmable Logic Controller) se používá pro celou třídu produktů speciálně vyvinutých pro řízení v průmyslovém prostředí. PLC se svou činností podobající více regulátorům než běžným počítačům, protože vykonávají periodické operace — čtení vstupů z periférií (*input scan*) do paměti vstupů (*input image*), výpočet zásahu (*program scan*) a zápis výstupů (*output scan*) z výstupní paměti (*output image*) do periférií. Tyto fáze se u většiny typů PLC dají znázornit diagramem:



Zatímco spolehlivost hardwaru PLC bývá obecně vysoká, programy tvoří neznámé veličiny, a proto se usiluje o ověřování jejich korektnosti. Zkoušení PLC programu se zpravidla provádí sledováním jeho reakcí na vhodné sekvence vstupů. Tato technika dostala jméno "debugging" [1]

[1]Debugging byl údajně pojmenovaný na základě mrtvé můry nalezené mezi kontakty relé během vývoje počítače Mark II. (*'bug' = hmyz*)

a patří mezi časté postupy, nicméně je špatným diagnostickým nástrojem. "*Testování prokáže jen přítomnost chyb, ale nikoliv jejich absenci.*" (E.W. Dijkstra) Chyby v programu dovedou vyloučit některé nástroje pro ověření modelů, avšak dnešní stav je příliš vzdálený ideální situaci naznačené na obrázku dole: "Vlož PLC program a kritéria, která má splňovat, a po chvilce 'pilné práce' bude zverifikováno."

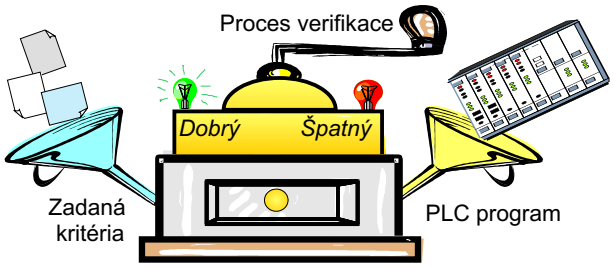


Figure 1 - Ideální verifikátor

PLC řídí totiž nejen technologii, ale i pomocné prvky, jako například panely operátorů, výkonné elementy, snímače, a případné paralelně spolupracující řídicí jednotky, a proto jeho program zahrnuje také operace pro spolupráci a konfiguraci podobných elementů, a též i některých programovatelných I/O modulů. Podobné operace příliš závisí na konkrétním hardwaru, a proto můžeme směle tvrdit, že ideální verifikátor z našeho obrázku nebude s nejvyšší pravděpodobností nikdy existovat. Podobné tvrzení vyvolává otázku, má-li tedy vůbec cenu zabývat se verikací PLC programů? Má, ale samozřejmě nikoliv pro celé PLC programy, pouze pro jejich dílčí částí, třeba pro podprogramy. Jejich verifikace se v poslední době provádět i v praxi, zejména u knihovních modulů.

2 SMV

Jedním z mnoha dostupných verifikačních nástrojů je SMV [3]. Umožňuje ověřování vlastností systémů s konečným počtem stavů. Tyto vlastnosti jsou zadávány ve formě formulí CTL (Computation Tree Logic). Pro popis systémů SMV používá vlastní jazyk, kterým lze modelovat širokou škálu systémů od synchronních automatů přes asynchronní logické obvody až například ke komunikačním protokolům.

2.1 Stručný popis modelovacího jazyka

Napřed popíšeme modulární skladbu programu popisujícího model systému a následně některé z jeho základních příkazů.

Model systému se skládá z jednoho nebo více modulů. Rozepsání do více modulů může zpřehlednit model a umožňuje snadno popisovat systémy s opakujícími se prvky. Hlavním a povinným modulem v SMV je main, ve kterém můžeme definovat instance dalších modulů. Ověřovaný program PLC bude v main definován jako jeden modul, v něm budou případně definovány další moduly obsahující části popisu chování programu. Pokud budeme ověřovat chování samotného PLC vůči libovolným vstupům a nerozložíme jeho popis do více modulů, mohl by nám stačit main. Jestliže omezíme vstupy PLC popisem řízeného procesu, prostředí PLC, jehož chování bývá vůči běhu řídícího programu asynchronní, budeme tento proces modelovat jako jeden nebo více dalších modulů.

Základními datovými typy vstupního jazyka jsou binární číslo a skalár, ten je buď ve formě čísla ze zadaného konečného intervalu celých čísel nebo ve formě výčtového typu. V našem případě postačí k popisu programu PLC pouze binární proměnné, protože výstupem

APLCTrans je automat vyjádřený jako přiřazení výrazů binární logiky binárním proměnným. Definice proměnných a instancí modulů se uvádějí v sekci VAR.

Následuje příklad definice proměnných a modulů.

```
MODULE PLC(remote, button, topLimit, botLimit, beam)
VAR
 closing : boolean;
 opening : boolean;
...
MODULE gate(op, cl)
VAR
 topLimit : boolean;
 botLimit : boolean;
...
MODULE main
VAR
 remote : boolean;
 button : boolean;
 beam : boolean;
 g : gate(C.opening, C.closing);
 C : PLC(remote, button, g.topLimit, g.botLimit, beam);
```

Příklad 1. Definice proměnných a modulů

Takto definované moduly g a C se vyhodnocují oba v každém stavovém přechodu SMV programu. Druhou variantou definice modulů je použití klíčového slova *process* v definici instance, potom moduly běží v prokládaném režimu, tj. vždy je náhodně vybrán jeden z nich, jehož tělo se podílí v daném okamžiku na příštích hodnotách proměnných. Toto je jeden ze způsobů, jak lze modelovat asynchronní souběh systémů popsaných jednotlivými moduly.

Program popisující systém jazykem SMV je soustava rovnic, které určují následující stav systému. Typicky je popsán v sekci ASSIGN, která obsahuje přiřazení výrazů jednotlivým proměnným v daném modulu. Nejzákladnějším způsobem popisu stavového prostoru je příkaz `next(`*proměnná*`) := výraz`, který určuje hodnotu proměnné v následujícím stavu. Výrazem může být logický nebo aritmetický výraz, ale také kostrukce *case* nebo nedeterministické přiřazení.

```
MODULE gate(op, cl)
VAR
...
 state : {mdown, mup, stop};
ASSIGN
 init(topLimit) := (!botLimit) union 0;
 next(topLimit) := case
  topLimit&!(state=mdown) : 1;
  topLimit : {1,0};
  (state=mup)&!botLimit : {0,1}
  1 : topLimit;
 esac
...
```

Příklad 2. Ukázka přiřazení

Konstrukce case se vyhodnocuje postupně – je-li splněna první podmínka, přiřadí se výraz za ":" a vyhodnocování se ukončí, jinak se pokračuje další podmínkou, přiřazen je tedy první výraz, jehož podmínka je splněna. Nedeterministické přiřazení se zapisuje jako množina možných hodnot, v ukázkovém příkladu vypsáním mezi složené závorky nebo pomocí slova

union. Hodnota proměnné je pak určena náhodně z této množiny. Kromě přiřazení příští hodnotě lze přiřazovat i momentálním hodnotám proměnných, ale tuto možnost nebudeme využívat. V sekci ASSIGN můžeme určit i počáteční hodnotu proměnné příkazem *init*, jehož způsob zápisu je shodný s příkazem next.

Nedeterministická přiřazení jsou dalším prostředkem k popisu asynchronních systémů. Například zápis `(state=mup)&!botLimit : {0,1}` v příkladu 2 znamená, že jsou-li splněny podmínky pro přechod *topLimit* z 0 do 1, stav se změní buď ihned nebo zatím ne, změna pak může nastat až někdy v budoucnosti, se zpožděním. Modelujeme tak situaci, kdy odezva řízeného systému na řídící veličiny nemusí být okamžitá.

2.2 Verifikace kritérií

Popsali jsme podmnožinu možností popisu systému nástrojem SMV, kterou použijeme při modelování řídicího programu. K verifikaci tohoto modelu uvedeme za popis modulů jednu nebo více sekcí SPEC obsahujících ověřované formule CTL logiky.

CTL logika [3], je jednou z temporálních logik, systémů pro vytváření výroků o změnách v čase. Umožňuje popsat různé situace. Z verifikace programu, na němž na dalších stránkách demonstrujeme metodu vytváření vstupního souboru pro SMV, uvádíme na tomto místě některé příklady kritérií zapsaných v syntaxi SMV.

absence deadlocku, například `AG ((EF C.closing) & EF !C.closing)` – vždy existuje nějaká možnost, jak může *C.closing* nabýt hodnoty 1, a možnost, jak může nabýt 0,

vzájemné vyloučení, například `AG !(C.closing&C.opening)` – *C.closing* a *C.opening* nemohou nikdy být zároveň rovny 1,

nutný důsledek, například `AG ((beam&C.closing&!g.topLimit) -> AX C.opening)` – jestliže je výraz na levé straně implikace pravdivý, musí být v následujícím kroku (pro nás též scanu PLC) pravdivý *C.opening*.

Výsledek verifikace může být dvojí. Buď je daná formule ve verifikovaném modelu pravdivá, potom tuto skutečnost SMV oznámí, nebo pravdivá není, potom se navíc pokusí vypsat posloupnost stavů, která vede k nesplnění formule. Příklad takového výpisu se nachází v odstavci 5.1.1.

Posledním zde zmíněným prvkem jazyka SMV je sekce FAIRNESS. Formule CTL uvedená v této sekci je splněná nekonečně často. Při verifikaci se tedy ignorují trajektorie, ve kterých nekonečně často splněna není.

3 APLCTRANS

Jazyk SMV se dobře hodí pro verifikaci PLC programů předevených pomocí APLCTRANS algoritmu popsaného detailně v [4] a v [5] lze najít vysvětlení jeho hlavních principů. V této části proto pouze naznačíme hlavní principy.

Práce algoritmu vychází z předpokladu, že PLC program lze popsat ve fromě šestice:

$$\mathcal{P}_{PLC} \stackrel{df}{=} \langle \Sigma, \Omega, V, A_\Sigma, \delta_P, q_0 \rangle \tag{1}$$

kde

- Σ je konečná množina PLC vstupů, tj. *input image*,
- Ω je konečná množiny PLC výstupů, tj. *output image*,
- V reprezentuje vnitřní proměnné PLC programu,
- A_Σ označuje vstupní abecedu tvořenou všemi kombinacemi vstupů, na než program reaguje,

- δ_P zastupuje program popsatelný nějakou vhodnou přechodovou funkcí
- q_0 označuje počáteční stav.

$\mathcal{P}_{PLC}$ provádí pak operace nad množinou proměnných S, která představuje předně sjednocení tří disjunktních množin Σ, Ω a V, k níž musíme přidat ještě registry PLC procesoru, nejméně $@f$ register, [2] a zásobník E_{stack} pro hodnoty $@f$. Na základě toho můžeme definovat paměť S, nad níž se provádějí operace jako $S = \Sigma \cup \Omega \cup V \cup \{@f\} \cup E_{stack}$. Hodnoty E_{stack} a $@f$ se inicilizuji na začátku každého scanu programu a i mezi jednotlivými bloky, takže se neobjeví ve výstupech a jsou v S zahrnuté pouze pro vyjádření dílčích operací během výpočtu instrukcí.

Pokud S obsahuje pouze binární proměnné, v [4] se definuje pro tento případ i operační semantika APLC automatu (APLC machine) a syntaxe APLC jazyka, který bude tímto automatem interpretován. [3] Struktura APLC jazyka se opírá o několikaletá studia PLC různých výrobců (jmenovitě Allen-Bradley Rockwell Automation, Siemens a Omron) a byla navržena tak, aby dovolovala snadnou konverzi nejčastěji používaných typů PLC. V případě potřeby se dá i snadno rozšířit o další instrukce. V tomto článku bude použita pro popis programu v příkladu, protože pro APLC instrukce byla v [4] vytvořena již převodní tabulka na trans-množiny a existuje pro ně i převádějící program APLCTRANS.

Pokud bychom měli několika slovy vystihnout hlavní podstatu trans-množin, mohli bychom je přirovnat k jakýmsi programovým diferencím. Každá trans-množina svým způsobem zachycuje provedené změny mezi dvěmi následujícími operacemi, či programovými bloky, jakousi jejich přechodovou funkci. Zhruba lze říct, že trans-množiny formalizují současné výpočet několika výrazů najednou.

Předpokládejme, že máme proměnné x, y dva přiřazovací výrazy "$x := 2*x$; $y := x+1$;". Jejich postupné provedení vede na "$y := 2 * x + 1$;" pro y proměnnou, ale jejich současný výpočet používaný trans-množinami dává "$y := x + 1$;", protože proměnným x, y se přiřadí nové hodnoty až po výpočtu pravých stran obou výrazů. Současná přiřazení se proto mohou uvádět v libovolném pořadí a lze vytvořit jejich množinu. Příklad nahoře koresponduje trans-množině $\widehat{X} = \{\hat{x}[\![2 * x]\!], \hat{y}[\![x + 1]\!], \}$, kde stříška vždy označuje proměnnou, jejichž hodnotu mění příslušný výraz $[\![e]\!]$.

Trans-množiny dovolují přehledně popsat dokonce i složité operace, například "push x" do 3-úrovňového zásobníku e_1, e_2, e_3 lze vyjádřit jako $\widehat{Y} = \{\hat{e_1}[\![x]\!], \hat{e_2}[\![e_1]\!], \hat{e_3}[\![e_2]\!]\}$.

Vhodně definované skládání trans-množin má asociativní charakter, což dovoluje vytvoření efektivního APLCTRANS algoritmu, který ve většině případů převede PLC program na trans-množinu v lineárním čase vzhledem k počtu instrukcí a množina všech trans-množin $\widehat{\mathcal{S}}(S)$ definovaných nad S množinou proměnných spolu s operací kompozice $\circledcirc$ trans-množin tvoří monoid, důkaz asociativity a existence monoidu je provedený v [4] na str. 66–70.

3.1 Přehled použitých pojmů trans-množin

Definice 3.1 *Nechť bexp je přípustný výraz dle nějaké gramtiky Gbexp, pak domain bexp je definovaný jako:*

$$\mathrm{dom}(bexp) \stackrel{df}{=} \{b_i \in S \mid b_i \textit{ je použito v } bexp\} \tag{2}$$

Množina S byla již definovaná výše. Příslušnou gramatiku může pro náš případ tvořit libovolná gramatika generující booleovský výraz s operátory $\neg$,$\vee$ a $\wedge$

[2] $@f$ se používá pro výpočet podmínky dané kombinací logických instrukcí. Svou hodnotu ovlivňuje činnost výstupních instrukcí příslušného programovacího jazyka, jako třeba žebříčkového diagramu, funkčních bloků či instrukčního kódu.

[3] Definice lze sice rozšířit i o běžné aritmetické proměnné a o časovače, avšak v tomhle článku zůstaneme pro jednoduchost jen u binárních proměnných.

Definice 3.2 *Nechť $b \in S$ je libovolná binární proměnná z konečné paměti S, $bexp \in Bexp^+$ je libovolný výraz splňující* $\mathrm{dom}(bexp) \subset S$ *a* $b := [\![bexp]\!]\, S$ *je přiřazovací operace bexp pro b proměnnou počítaná vzhledem k hodnotám v S. Definujeme*

$$\begin{aligned} \text{t-přiřazení:} \quad & \hat{b}[\![bexp]\!] \stackrel{df}{=} b := [\![bexp]\!]\, S \\ \text{domain pro t-přiřazení:} \quad & \mathrm{dom}(\hat{b}[\![bexp]\!]) \stackrel{df}{=} \mathrm{dom}(bexp) \\ \text{codomain pro t-přiřazení:} \quad & \mathrm{co}(\hat{b}[\![bexp]\!]) \stackrel{df}{=} b \end{aligned}$$

T-přiřazení $\hat{b}[\![bexp]\!]$ se nazývá kononické t-přiřazení, *když $bexp \equiv b$. Množinu všech t-přiřazení pro S proměnné nazveme $\widehat{\mathcal{B}}(S)$.*

Definice 3.3 *Nechť $\hat{y}[\![bexp_y]\!], \hat{x}[\![bexp_x]\!] \in \widehat{\mathcal{B}}(S)$ jsou dvě t-přiřazení. Binary relace $\hat{x} \mathrel{\widehat{=}} \hat{y}$ je definována jako současné splnění dvou následujících podmínek:* $\mathrm{co}(\hat{x}) = \mathrm{co}(\hat{y})$ *and* $bexp_x \equiv bexp_y$.

Pokud $\mathrel{\widehat{=}}$ relace nebude splněna pro nějaká t-přiřazení $\hat{x}, \hat{y} \in \widehat{\mathcal{B}}$, pak tento fakt zdůrazníme negovaným symbolem $\hat{x} \mathrel{\widehat{\neq}} \hat{y}$.

Definice 3.4 (Trans-množina) *Podmnožinu $\widehat{X} \subseteq \widehat{\mathcal{B}}(S)$ nazveme* trans-množinou na S, *když $\widehat{X}$ splňuje pro všechna $\hat{x}_i, \hat{x}_j \in X$, že* $\mathrm{co}(\hat{x}_i) = \mathrm{co}(\hat{x}_j)$ *implikuje $i = j$. Množinu všech trans-množin pro S proměnné označíme jako $\widehat{\mathcal{S}}(S)$, čili $\widehat{X} \in \widehat{\mathcal{S}}(S)$.*

Jinými slovy, trans-množina obsahuje pro každou proměnnou z S nejvýše jedno t-přiřazení.

Definice 3.5 *Binární relace $\widehat{\in}$ na množině S a $\widehat{\mathcal{B}}(S)$ je definována pro všechna $\widehat{X} \in \widehat{\mathcal{S}}(S)$ a $x \in S$ jako*

$$\begin{aligned} \widehat{\in} \stackrel{df}{=} \quad & x \mathrel{\widehat{\in}} \widehat{X} \quad \textit{iff } \exists\, \hat{x}[\![bexp]\!] \in \widehat{X} \textit{ takové, že } x = \mathrm{co}(\hat{x}[\![bexp]\!]) \\ & x \mathrel{\widehat{\notin}} \widehat{X} \quad \textit{v opačném případě.} \end{aligned}$$

Definice 3.6 *Nechť $\widehat{X} \in \widehat{\mathcal{S}}(S)$ je libovolná trans-množina definovaná na S. Rozšíření $\widehat{X}$, označené jako $\widehat{X} \uparrow S$, je trans-množina s mohutností $|\widehat{X} \uparrow S| = |S|$, jejíž prvky jsou $\hat{x}_i$ t-přiřazení definovaná pro všechna $x_i \in S$ jako*

$$\hat{x}_i \stackrel{df}{=} \begin{cases} \hat{x}_i & \textit{if } x_i \mathrel{\widehat{\in}} \widehat{X} \textit{ a proto } \exists \hat{x}_i \in \widehat{X} \textit{ splňující } \mathrm{co}(\hat{x}_i) = x_i \\ \hat{x}_i\,[\![x_i]\!] & \textit{if } x_i \mathrel{\widehat{\notin}} \widehat{X} \end{cases}$$

Operátor $\uparrow$ přidá do trans-množiny kanonická t-přiřazení pro všechny proměnné z S, pro něž tam chybějí. Operace je důležitá pro asociativitu a existuje k ní i opačná operace komprese $\downarrow$, která naopak odstraní všechna kanonická t-přiřazení odstraňuje, takže redukuje prostor pro uložení.

Definice 3.7 *Nechť $\widehat{X} \in \widehat{\mathcal{S}}(S)$ je libovolná trans-množina definovaná na S, pak její* codomain *a* domain *se rovnají*

$$\mathrm{co}(\widehat{X}) = \bigcup_{i=1}^{|\widehat{X}|} \mathrm{co}(\hat{x}_i) \qquad \mathrm{dom}(\widehat{X}) = \bigcup_{i=1}^{|\widehat{X}|} \mathrm{dom}(\hat{x}_i) \qquad \textit{kde } \hat{x}_i \in \widehat{X} \tag{3}$$

Domain a codomain trans-množiny je pouhé sjednocení domain a codomain všech t-přiřazení, která do ní patří. Pro řadu případů však potřebujeme jejich zúžení na proměnné, které jsou důležité pro vyjádření stavu PLC programu. Musíme proto vyloučit vstupy a vnitřní proměnné PLC procesoru.

Definice 3.8 *Nechť $\widehat{X} \in \widehat{\mathcal{S}}(S)$ je libovolná trans-množina definovaná na S a $(\Omega \cup V) \subseteq S$, pak definujeme*

$$\begin{aligned} \widehat{\mathrm{co}_P}(\widehat{X}) &= \left\{ \hat{x} \in \left((\widehat{X} \mathbin{\widehat{\cap}} (\Omega \cup V)) \downarrow \right) \;\middle|\; \hat{x} \mathbin{\widehat{\neq}} \hat{x}\,[\![0]\!] \;\wedge\; \hat{x} \mathbin{\widehat{\neq}} \hat{x}\,[\![1]\!] \right\} \\ \mathrm{co}_P(\widehat{X}) &= \mathrm{co}\left(\widehat{\mathrm{co}_P}(\widehat{X}) \right) = \left\{ x \in S \mid x \mathbin{\widehat{\in}} \widehat{\mathrm{co}_P}(\widehat{X}) \right\} \end{aligned} \tag{4}$$

Trans-množina $\widehat{\mathrm{co}_P}(\widehat{X})$ a množina $\mathrm{co}_P(\widehat{X})$ se nazývají PLC codomain trans-množiny $\widehat{X}$.

4 PŘEKLAD ŘÍDICÍHO PROGRAMU DO SMV

S využitím pojmů z předchozí sekce, můžeme poměrně snadno zapsat převod PLC programu již konvertovaného pomocí APLCTrans na trans-množiny do jazyka SMV.

4.1 Modul paralelní části programu

Modul pro realizaci paralelní části $\widehat{D}_i$ nazvěme M_i a množinu vstupů této části $vstupyM_i$. Vstupy určíme

$$vstupyM_i = \mathrm{dom}(\widehat{D}_i) - \mathrm{co}(\widehat{D}_i). \tag{5}$$

Modul potom zapíšeme tak, že

- hlavička bude `MODULE` $M_i(vstupyM_i)$,
- v sekci VAR definujeme všechny proměnné z $\mathrm{co}(\widehat{D}_i)$ jako boolean,
- sekce ASSIGN obsahuje zápis všech t-asigments $\widehat{X}_j = \{x_j := e_j\}$ z $\widehat{D}_i$ ve tvaru `next` x_j `:=` e_j`;`

4.2 Modul řídicího programu

Nechť $\widehat{C}$ je trans-množina, kterou jsme získali překladem IL programu pomocí APLCTrans, Σ je množina všech vstupů programu a $@f$ flag register APLC. Jestliže $@f \notin \mathrm{dom}(\widehat{C})$, vyjměme $\widehat{@f}$ z $\widehat{C}$. Množina vstupů modulu potom je

$$I = (\mathrm{dom}(\widehat{C}) - \mathrm{co}(\widehat{C})) \cap \Sigma \tag{6}$$

a množina vnitřních proměnných

$$V = \mathrm{co}(\widehat{C}) \cup (\mathrm{dom}(\widehat{C}) - I). \tag{7}$$

Trans-množinu PLC programu získáme

$$\widehat{PLC} = \widehat{C} \uparrow V. \tag{8}$$

Struktura modulu řízení se liší podle toho, zda provádíme nebo neprovádíme dekompozici. Bez dekompozice bude

- hlavička `MODULE PLC(`I`)`,
- v sekci VAR definujeme všechny proměnné z V jako boolean,
- sekce ASSIGN obsahuje zápis všech t-asigments $\widehat{X}_j = \{x_j := e_j\}$ z $\widehat{PLC}$ ve tvaru `next` x_j `:=` e_j`;`

Modul řídicího programu rozloženého na paralelní části realizované moduly M_i má hlavičku stejnou jako modul bez dekompozice, jeho vstupy jsou stejné. Stejné je i jeho chování z pohledu vnitřních proměnných, které jsou obsažené v použitých modulech paralelních částí automatu. Sekce těla modulu mají jinou strukturu.

- VAR obsahuje definice všech proměnných z $V - \mathrm{co_P}(\widehat{C})$ a definice všech instancí modulů paralelních částí, které obsahují některou verifikovanou proměnnou nebo proměnnou, která je vstupem nějakého modulu popisu prostředí PLC.
- ASSIGN obsahuje zápis všech t-asigments $\widehat{X}_j = \{x_j := e_j\}$ z $\widehat{PLC} - \widehat{\mathrm{co_P}}(\widehat{C})$ ve tvaru `next` x_j `:=` e_j`;`

4.3 Hlavní modul SMV programu

V hlavní části programu budeme definovat instance modulů popisu okolí PLC, instanci hlavního modulu řídicího programu a vstupy, které nejsou popsány žádným modelem. Vstupem PLC může tedy být proměnná definovaná v modulu main nebo proměnná definovaná v popisu řízeného procesu. Vstupem modulu procesu může být také proměnná hlavní části programu, proměnná z jiného modulu procesu a nebo navíc vnitřní proměnná řídícího programu. Při deklarování vstupů daného modulu zapisujeme jméno proměnné i se jménem instance modulu, v němž je definovaná. V příkladu 1 je ukázka kódu obsahující tento zápis. Při deklaraci instancí modulů popisu prostředí neuvádíme klíčové slovo process, protože prokládaný režim nevystihuje souběh působení výstupů PLC na řízený proces s vyhodnocováním nových hodnot výstupů. Veškerou neurčitost rychlosti reakce popisují nedeterministická přiřazení.

5 PŘÍKLADY PŘEVODU A VERIFIKACE

Na dvou jednoduchých příkladech ukážeme model PLC programu v jazyku SMV. V prvním případě provedeme verifikaci tohoto modelu a rozšíření o popis řízeného procesu. V případě druhém ukážeme model, jehož popis rozdělíme paralelní dekompozicí na více modulů.

5.1 Řízení garážových vrat

Navrhněte řízení garážových vrat, jeho chování má být následující.

- v garáži je jedno tlačítko a jedno tlačítko je na dálkovém ovládání.
- když je stisknuto tlačítko, vrata se začnou pohybovat nahoru nebo dolů.
- když je tlačítko stisknuto během pohybu vrat, vrata se zastaví, další stisknutí způsobí pohyb v opačném směru.
- k zastavení pohybu vrat slouží horní a dolní koncový spínač.
- v dolní části prostoru vrat svítí paprsek světla, který detekuje přítomnost předmětu, je-li přerušen během zavírání dveří, pohyb se změní na druhý směr.

Pro zápis programu i jeho verifikaci zavedeme následující pojmenování proměnných.

closing je výstup pro pohyb vrat směrem dolů,
opening pro pohyb vrat směrem dolů,
topLimit je vstup signalizující sepnutí horního koncového spínače,
botLimit sepnutí dolního koncového spínače,
beam signalizuje přerušení paprsku světelného detektoru.

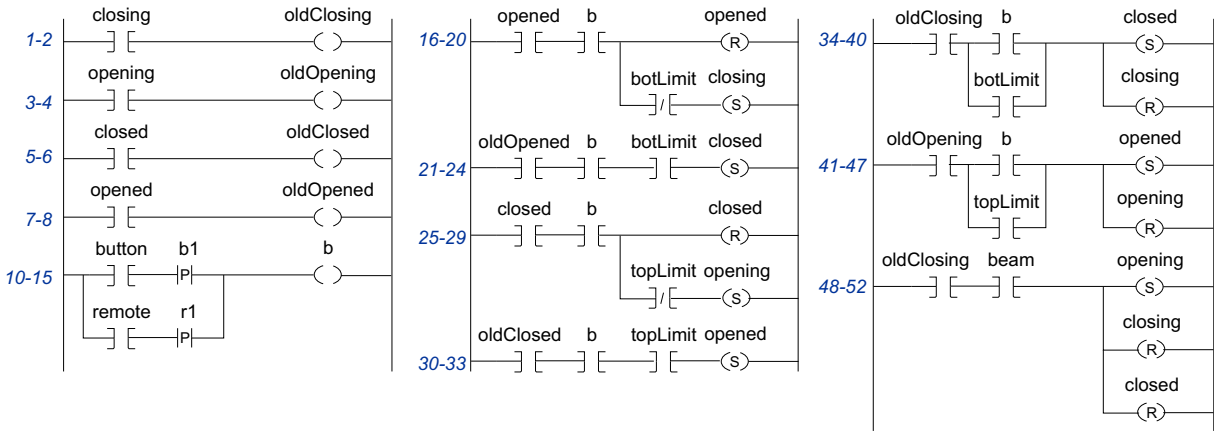


Figure 2 - Program pro řízení zavírání vrat

Žebříčkový diagram řídicího programu, vstup pro převod APLCTrans algoritmem, je ukázaný na na obrázku 2. Přitom předpokládáme, že kromě tohoto kódu je někde v PLC naprogramována i inizializace proměnných $opened = 1, closed = closing = opening = 0$. Po jeho převodu do instrukcích APLC dostaneme (modrá čísla v obrázku nahoře odpovídají číslům u APLC instrukcí):

```
 1 Load closing
 2 Store oldClosing
 3 Load opening
 4 Store oldOpening
 5 Load closed
 6 Store oldClosed
 7 Load opened
 8 Store oldOpened
 9 Load button
10 REdge b1
11 Push
12 Load remote
13 Redge r1
14 TOr
15 Store b
16 Load opened
17 And b
18 Res opened
19 And !botLimit
20 Set closing
21 Load oldOpened
22 And b
23 And botLimit
24 Set closed
25 Load closed
26 And b
27 Res closed
28 And !topLimit
29 Set opening
30 Load oldClosed
31 And b
32 And topLimit
33 Set opened
34 Load oldClosing
35 Push
36 Load b
37 Or botLimit
38 TAnd
39 Set closed
40 Res closing
41 Load oldOpening
42 Push
43 Load b
44 Or topLimit
45 TAnd
46 Set opened
47 Res opening
48 Load oldClosing
49 And beam
50 Set opening
51 Res closing
52 Res closed
```

Správnou funkci programu z velké části popisují CTL formule

$$\mathrm{AG}(\neg(closing \wedge opening)), \tag{9}$$

znamená, že nikdy nemůže být nastaven výstup pro pohon na jednu i na druhou stranu zároveň,

$$\mathrm{AG}((\mathrm{EF}\, closing) \wedge (\mathrm{EF}\, \neg closing)) \tag{10}$$

ověřuje neexistenci deadlocku na proměnné closing,

$$\mathrm{AG}(topLimit \Rightarrow \mathrm{AX}\ \mathrm{A}[\neg opening\, \mathrm{U}\, \neg topLimit]) \tag{11}$$

znamená, že pokud dosáhnou vrata horní polohy, nebude pohon směrem nahoru zapnutý do té doby, než vrata horní polohu opustí,

$$\mathrm{AG}(\mathit{botLimit} \Rightarrow \mathrm{AX}\ \mathrm{A}[\neg \mathit{closing}\,\mathrm{U}\,\neg \mathit{botLimit}]) \tag{12}$$

má stejný význam jako předchozí formule, tentokrát pro dolní polohu a pohon směrem dolů,

$$\mathrm{AG}((\mathit{beam} \wedge \mathit{closing} \wedge \neg \mathit{topLimit}) \Rightarrow \mathrm{AX}\,\mathit{opening}), \tag{13}$$

jestliže je během zavírání přerušen světelný paprsek a vrata nejsou v horní poloze, zapne se ihned pohon směrem nahoru.

Následně výše uvedeným postupem získáme soubor připravený pro verifikaci nástrojem SMV, do kterého doplníme specifikaci správné funkce jako CTL formule v sekcích SPEC. Příklad 3 ukazuje tento soubor bez doplněných specifikací.

```
MODULE PLC(remote, button, topLimit, botLimit, beam)
VAR
 opened : boolean;
 opening : boolean;
 closed : boolean;
 closing : boolean;
 oldClosing : boolean;
 oldOpening : boolean;
 oldClosed : boolean;
 oldOpened : boolean;
 b1 : boolean;
 r1 : boolean;
 b : boolean;

ASSIGN
init(opened) := 1;
init(closed) := 0;
init(closing) := 0;
init(opening) := 0;

next(closing) := !closing&opened&remote&!r1&!botLimit |
  !closing&opened&button&!b1&!botLimit | closing&b1&!remote&!botLimit&!beam |
  closing&b1&r1&!botLimit&!beam | closing&!button&!remote&!botLimit&!beam |
  closing&!button&r1&!botLimit&!beam;
next(oldClosing) := closing;
next(opening) := closing&beam | opening&b1&!remote&!topLimit |
  opening&b1&r1&!topLimit | opening&!button&!remote&!topLimit |
  opening&!button&r1&!topLimit | !opening&opened&remote&!r1&botLimit&!topLimit |
  !opening&opened&button&!b1&botLimit&!topLimit |
  !opening&closed&remote&!r1&!topLimit | !opening&closed&button&!b1&!topLimit;
next(oldOpening) := opening;
next(closed) := !closing&closed&b1&r1 | !closing&closed&b1&!remote |
  !closing&closed&!button&r1 | !closing&closed&!button&!remote |
  closing&remote&!r1&!beam | closing&button&!b1&!beam | closing&botLimit&!beam |
  closed&b1&r1&!beam | closed&b1&!remote&!beam | closed&!button&r1&!beam |
  closed&!button&!remote&!beam;
next(oldClosed) := closed;
next(opened) := opening&remote&!r1 | opening&button&!b1 | opening&topLimit |
  closed&remote&!r1&topLimit | closed&button&!b1&topLimit | opened&b1&!remote |
  opened&b1&r1 | opened&!button&!remote | opened&!button&r1;
next(oldOpened) := opened;
```

```
next(b1) := button;
next(r1) := remote;
next(b) := remote&!r1 | button&!b1;

MODULE main
VAR
 remote : boolean;
 button : boolean;
 beam : boolean;
 topLimit : boolean;
 botLimit : boolean;
 C : PLC(remote, button, topLimit, botLimit, beam);
```

Příklad 3. Řídicí program převedený do SMV

5.1.1 Výsledek verifikace

Soubor obsahující SMV program 3 a ověřované CTL formule v sekcích SPEC zpracujeme nástrojem SMV. Řádky výpisu pocházejí z verifikace programem NuSMV.

```
-- specification AG (!(C.closing & C.opening)) is true

-- specification AG (EF C.closing & EF (!C.closing)) is false
-- as demonstrated by the following execution sequence
-> State 1.1 <-
    remote = 1
    button = 1
    beam = 1
    topLimit = 1
    botLimit = 1
    C.opened = 1
    C.opening = 0
    C.closed = 0
    C.closing = 0
    C.oldClosing = 1
    C.oldOpening = 1
    C.oldClosed = 1
    C.oldOpened = 1
    C.b1 = 1
    C.r1 = 1
    C.b = 1
-> State 1.2 <-
    remote = 0
    C.oldClosing = 0
    C.oldOpening = 0
    C.oldClosed = 0
    C.b = 0
-> State 1.3 <-
    remote = 1
    C.r1 = 0
```

Výpis ukazuje posloupnost stavů, po které došlo k nesplnění formule. Můžeme si povšimnout, že jsou jedničkové vstupy od obou koncových spínačů zároveň. Jak bychom mohli v interaktivním módu NuSMV ověřit, ve stavu následujícím po State 1.3 jsou proměnné

$$opened = opening = closed = closing = 0.$$

Poté už nemůže žádná z nich být rovna 1.

Pro verifikaci následujících dvou kritérií bylo třeba nastavit sekce FAIRNESS !*topLimit* a FAIRNESS !*botLimit*. Jinak jsou jako posloupnosti nesplňující dané formule zobrazeny cykly, ve kterých nikdy nenastane !*topLimit*, resp. !*botLimit*. Formule s operátory until s těmito výrazy na pravé straně nebudou splněny ani v případě, že v těchto cyklech jsou stále splněny formule na levé straně. Zapsání uvedených sekcí FAIRNESS způsobí, že se budou procházet pouze cesty, ve kterých jsou nekonečně často porušeny formule na pravých stranách operátoru until a tak ověřujeme skutečně pouze proti tomu, aby výraz na levé straně byl nesplněn před splněním výrazu na pravé operátoru until.

```
-- specification AG (topLimit -> AX A [ (!C.opening) U (!topLimit) ] ) is false
```

Nyní výpis, který zde pro jeho délku neuvádíme, mimo jiné ukázal, že nesplnění formule předchází stav, ve kterém jsou proměnné *topLimit*, *closing* a *beam* rovné logické 1.

```
-- specification AG (botLimit -> AX A [ (!C.closing) U (!botLimit) ] ) is true

-- specification AG (((beam & C.closing) & !topLimit) -> AX C.opening) is true
```

5.1.2 Zavedení popisu řízeného procesu

Výsledek verifikace ukázal, že nesplnění formule (10) předcházel stav, při kterém byly vstupy od obou koncových spínačů aktivní, což není při normálním provozu možné. K ověření funkce řídicího programu v situaci, kdy se řízený proces chová tak, jak od něj čekáme, přidáme do popisu systému model očekávaného chování procesu. Tím je pro náš případ popis dynamiky vrat, například takový, jaký ukazuje příklad 4.

```
MODULE gate(op, cl)
VAR
 topLimit : boolean;
 botLimit : boolean;
 state : {mdown, mup, stop};

ASSIGN
 init(topLimit) := (!botLimit) union 0;
 next(topLimit) := case
  topLimit&!(state=mdown) : 1;
  topLimit : {1,0};
  (state=mup)&!botLimit : {0,1};
  1 : topLimit;
 esac;
 next(botLimit) := case
  botLimit&!(state=mup) : 1;
  botLimit : {1,0};
  (state=mdown)&!topLimit : {0,1};
  1 : botLimit;
 esac;
 next(state) := case
  (state=mdown)&cl&!botLimit : mdown;
  (state=mup)&op&!cl&!topLimit : mup;
  (state=mdown)&!op : {mdown, stop};
  (state=mdown) : {mdown, stop, mup};
  (state=mup)&!cl : {mup, stop};
  (state=mup) : {mdown, stop, mup};
  (state=stop)&cl : {mdown, stop};
  (state=stop)&op : {mup, stop};
  1 : stop;
 esac;
```

Příklad 4. Popis chování vrat v jazyku SMV

Modul main se oproti předchozímu případu změní. Jeho nynější podoba je v příkladu 1.

Verifikace ověřila, že deadlock už nemůže nastat. Formule (11) je opět neplatná, výpis by ukázal, že se tak stane po stavu podobném tomu, po kterém nebyla tato formule splněna v odstavci 5.1.1.

5.1.3 Částečná úprava kódu

Na základě odhalení stavu způsobujícího nesplnění (11) provedeme úpravu programu, při které posledních 5 řádků původního kódu nahradí následující sekvence.

```
48 Load oldClosing
49 And beam
50 Push
51 And !topLimit
52 Set opening
53 Res closing
54 Res closed
55 Pop
56 And topLimit
57 Res closing
58 Res closed
59 Set opened
```

Při verifikaci s modelem vrat jsou nyní všechny formule splněny, bez modelu neplatí formule (10), deadlock nastane po $botLimit \wedge topLimit = 1$. Z pohledu verifikovaných kritérií by se dal program tedy využít, pokud bychom se spolehli na to, že se skutečně budou signály od koncových spínačů chovat podle popisu chování vrat.

Opravený program ukazuje obrázek 3

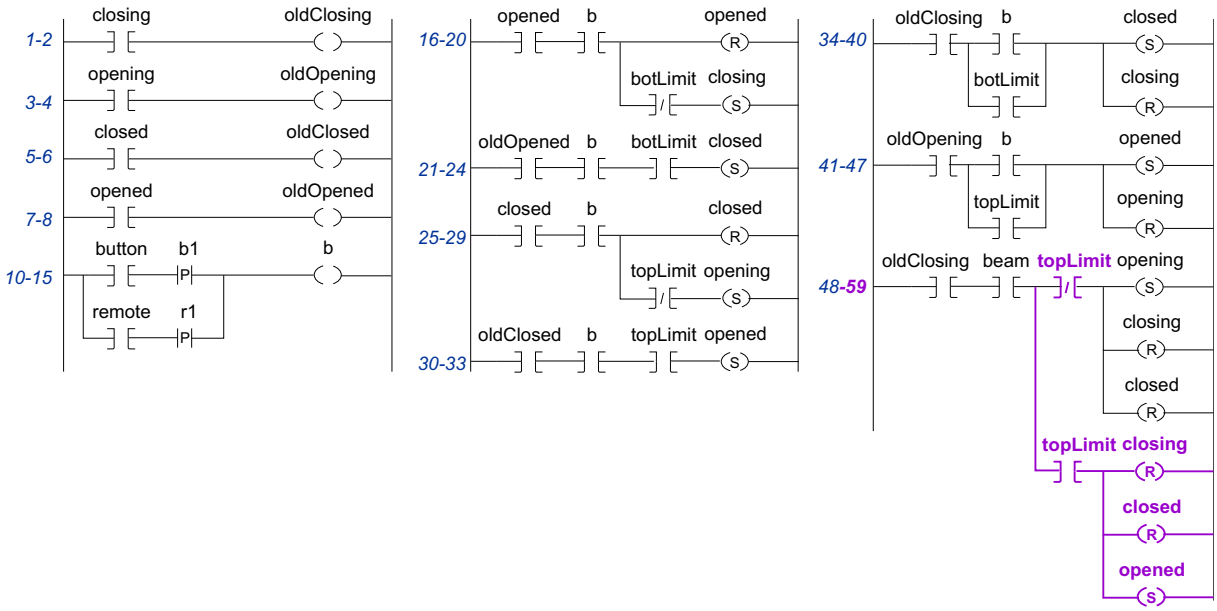


Figure 3 - Opravený program pro řízení zavírání vrat

5.2 Přípravná nádrž

Pro ukázku modelu programu rozloženého na paralelní části použijeme příklad řízení přípravné nádrže, na kterém je v [4] demonstrována dekompozice. Zde uvádíme pouze konečný kód SMV obsahující všechny paralelní části programu definované jako samostatné moduly.

```
MODULE M1(up, upe, dn, dne, swrspd)
VAR
 pmp : boolean;
ASSIGN
 next(pmp) := dn&!swrspd&!up | dne&!swrspd&!up&!upe | pmp&!up&!upe;

MODULE M2(dn, dne, pmpspd)
VAR
 swr : boolean;
ASSIGN
 next(swr) := !dn&!dne&!pmpspd | !dn&!dne&swr;

MODULE M3(clr, upe, dne)
VAR
 eu1 : boolean;
 erru : boolean;
 errd : boolean;
 clredge : boolean;
ASSIGN
 next(eu1) := clr;
 next(erru) := !clr&upe | !clr&erru | eu1&upe | eu1&erru;
 next(errd) := !clr&dne | !clr&errd | eu1&dne | eu1&errd;
 next(clredge) := clr&!eu1;

MODULE PLC(clr, up, upe, dn, dne, swrspd, pmpspd)
VAR
  C1 : M1(up, upe, dn, dne, swrspd);
  C2 : M2(dn, dne, pmpspd);
  C3 : M3(clr, upe, dne);

MODULE main
VAR
 clr : boolean;
 up : boolean;
 upe : boolean;
 dn : boolean;
 dne : boolean;
 swrspd : boolean;
 pmpspd : boolean;
 C : PLC(clr, up, upe, dn, dne, swrspd, pmpspd);

SPEC
 AG ((upe=1 & clr=0) -> AX(C.C3.erru))
SPEC
 AG !(C.C1.pmp&C.C2.swr)
```

Příklad 5. Program řízení přípravné nádže v jazyku SMV

Když verifikujeme jen jednu z uvedených formulí, druhou můžeme zakomentovat, stejně tak i deklarace instancí modulů nepotřebných pro verifikaci dané formule. Například při verifikaci první formule můžeme zakomentovat deklaraci C1 a C2 v modulu PLC. Zmenší se tak verifikovaný model.

Table 1 - Srovnání náročnosti výpočtu verifikace paralelně rozložitelných modelů

	CMU SMV		NuSMV		NuSMV -coi		Cadence SMV	
počet	čas	nody	čas	nody	čas	nody	čas	nody
2	0,02	1782	0,08	1944	0,08	1944	0,09	997
4	0,04	6599	0,1	8159	0,09	2441	0,09	997
8	0,13	11257	0,3	42571	0,09	3707		
12	0,4	30995	0,95	167280	0,1	5405	0,09	997
14	3,62	110910	8,05	44898				
16	13,4	284691	18,4	122879	0,12	7535	0,09	997
17	24,8	421009	33,6	60432				
18	64,3	652847	99,6	305503	0,14	8762	0,09	997

6 DISKUSE

6.1 Význam paralelní dekompozice pro verifikaci v SMV

Zabývejme se vlivem odstranění části automatu nepotřebné k verifikaci na časovou a paměťovou náročnost zpracování nástrojem SMV. Náročnost verifikace má obecně exponenciální závislost na velikosti ověřovaného modelu. Algoritmem paralelní dekompozice, který má složitost $O(n^3)$, kde $n = \mathrm{co}_{\mathrm{P}}(\widehat{C})$, získáme navzájem nezávislé části modelu řídicího programu. Nepotřebné paralelní části nedeklarujeme ve verifikovaném souboru. Tímto postupem lze snížit velikost výsledného modelu a tedy i náročnost verifikace.

Různé implementace SMV však poskytují ještě radikálnější způsoby zmenšení verifikovaného modelu. NuSMV tak činí po použití přepínače `coi` (cone of influence). Cadence SMV implicitně odstraňuje při verifikaci všechny proměnné, které neovlivňují hodnoty verifikovaných proměnných. Tato redukce je ještě účinnější než paralelní dekompozice, protože odstraňuje i ty proměnné, které jsou ve stejné paralelní části jako potřebná proměnná, ale nemají na její hodnotu vliv, tedy jsou na ní závislé, ale ne ona na nich. Navíc se tato redukce provádí na celém modelu, nejen na popisu řízení.

Pro demonstraci jsme vytvořili soubor obsahující stejné moduly obsahující 3 navzájem závislé proměnné. Přiřazení next jsme volili pseudonáhodně, Stejně tak i množinu vstupních proměnných každého modulu a jejich pořadí. Vstupní proměnné jsme vybírali ze společné množiny proměnných definovaných v modulu main. Žádné dva moduly neměly definovány zcela stejné vstupy. Verifikovaná CTL formule byla postavena na proměnných z prvních dvou modulů. V tabulce 1 jsou časy verifikace a počty nodů příslušných BDD diagramů. Redukcí pomocí paralelní dekompozice bychom dosáhli odstranění všech modulů kromě 2 potřebných, platil by tedy první řádek tabulky. Tabulka a obrázek 4 ukazují, jak se s modelem vypořadaly různé implementace SMV. Výsledek pro Cadence SMV a pro NuSMV s přepínačem `coi` by byl oproti paralelní dekompozici lepší, kdyby model obsahoval proměnné, které paralelní dekompozice neodstraní. Takové proměnné se však v PLC programech přirozeně vyskytují. Například v programu řízení garáže při verifikaci libovolné podmnožiny uvedených formulí mezi ně vždy patří pomocné proměnné *old*... a proměnná *b*.

Poznamenejme ještě, že redukce modelu nemá význam pouze ve snížení náročnosti výpočtu, ale také ve zpřehlednění výpisu stavových posloupností, které po redukci neobsahují proměnné nepotřebné k verifikaci.

[1]Při verifikaci se nyní ukázalo výhodné zapnout dynamické řazení proměnných v OBDD diagramu, v CMU SMV parametrem reorder, v NuSMV dynamic.

[2]Výpočet byl přerušen.

Table 2 - Náročnost výpočtu verifikace paralelně rozložitelných modelů

	CMU		-reorder[1]		NuSMV		-dynamic[1]		Ca SMV	
metoda	čas	nody	čas	nody	čas	nody	čas	nody	čas	nody
Šusta	0,08	10028	0,08	10028	0,2	9299	0,23	3880	0,1	1242
Canet	>238[2]	>2038423	5,35	17063	110	637180	1,2	45365	1,1	30720

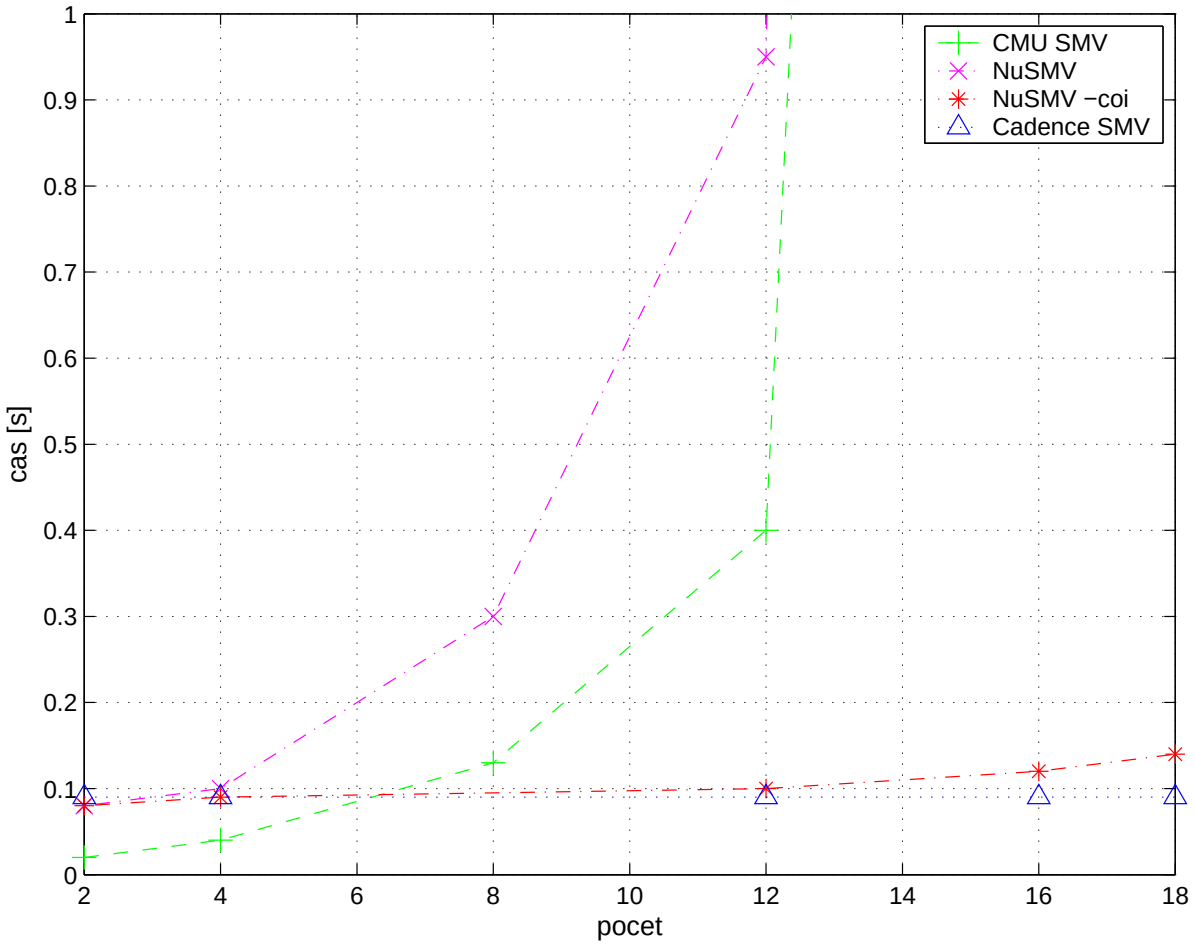


Figure 4 - Graf srovnání náročnosti verifikace paralelně rozložitelných modelů

6.2 Srovnání s jinými metodami verifikace PLC programů v SMV

Program řízení garáže jsme namodelovali i postupem podle [1]. Jejich přístup je zcela jiný. Nemodelují jeden PLC scan jako jeden krok v SMV, ale každou instrukci jako jeden přechod. Mohou se tak popsat širší možnosti PLC (práce s celými čísly, smyčky v programu,...). Některé CTL se hůř vyjadřují, například následující scan nelze určit jednoduchým použitím operátoru X. Výpis posloupnosti stavů obsahuje změny po každé provedené instrukci, obsahuje tedy více informací, na druhou stranu však je velice dlouhý a méně přehledný. Nevýhodou modelu sestaveného podle [1] je větší časová a paměťová náročnost jeho verifikace, jak ukazuje tabulka 2 obsahující charakteristiky náročnosti ověření formulí (9) a (10).

References

[1] CANET, G. et al. Towards the automatic verification of PLC programs written in instruction list. In *Proc. IEEE Int. Conf. Systems, Man and Cybernetics* (SMC'2000). Nashville, TN, USA, 2000. Dostupné z <http://www.lsv.ens-cachan.fr/Publis>.

[2] JACK, H. *Automating Manufacturing Systems with PLCs* [online]. Version 4.2, April 3, 2003 [cite 2004-03-23].
<http://claymore.engineer.gvsu.edu/~jackh/books/plcs/pdf/plcbook4_2.pdf>

[3] McMILLAN, K. L. *Symbolic Model Checking, An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[4] ŠUSTA, R. *Verification of PLC programs.* PhD thesis, Revised edition. CTU FEE Prague, 2003. Dostupné z <http://dce.felk.cvut.cz/susta/publications/thesis.htm>.

[5] ŠUSTA, R. *APLCTRANS Algorithm for PLC Verification.* 14th International Conference Process Control 2003, June 8-11, 2003, Štrbské Pleso, Slovakia, Proceedings ISBN 80-227-1902-1, CD-ROM.