České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

DIPLOMOVÁ PRÁCE

Verifikace s SMV

Otakar Šprdlík

2005

Vedoucí: Ing. Richard Šusta, Ph.D.
Katedra řídicí techniky
ČVUT v Praze, Fakulta elektrotechnická

Oponent: Ing. Pavel Burget
Katedra řídicí techniky
ČVUT v Praze, Fakulta elektrotechnická

Abstrakt

Formální verifikační techniky ověřují vlastností různých systémů, mimo jiné se mohou využít pro verifikaci programů programovatelných logických automatů (PLC). K provedení verifikace PLC programů je nutné jejich modelování ve verifikačním nástroji, například SMV nebo UPPAAL. Práce předkládá postup modelování programu převedeného algoritmem APLCTRANS na soustavu logických rovnic, které tvoří abstrakci vhodnou pro modelování ve zmíněných nástrojích. Navržený postup dokáže modelovat pouze programy, které APLCTRANS umí zpracovat, to znamená programy, které využívají omezených programovacích možností. Ukazuje se však, že výsledné modely mohou vést na nižší výpočetní nároky na proces verifikace než modely vytvořené obecnějšími postupy založenými na slučování modelů jednotlivých programových elementů (např. instrukcí). Postup se hodí pro modelování a verifikaci krátkých programů nebo dílčích bloků či funkcí rozsáhlejších řídicích algoritmů.

Abstract

Formal verification methods check properties of miscelaneous systems. For example, they can be used for validation of Programmable Logic Controller (PLC) programs. Verification of PLC programs can be done by their modelling in a universal verification tool as SMV or UPPAAL. This diploma thesis proposes a modelling procedure which creates a SMV or UPPAAL model of PLC program given as system of logical equations got by the APLCTRANS algorithm. The automaton described by this equation system is an abstraction suitable for modelling in the mentioned tools. Proposed procedure is able to model only programs, which can be proceeded by APLCTRANS, that means all programming facilities cannot be used in the modelled program. Obtained model can lead to lower computational burden than models derived from particular models of program elements (for instance instructions). The procedure is suitable for modelling and verification of short control programs or program fragments and functions.

Obsah

1 Úvod **6**

2 Model činnosti PLC **8**
2.1 Základní rysy PLC 8
2.2 Klasifikace modelů PLC 10
2.3 Abstraktní PLC 10
2.3.1 Model binárního PLC 11
2.3.2 Trans-množiny 13
2.3.3 Automat generovaný APLC programem 14

3 Model Checking **16**
3.1 Temporální logika 16
3.1.1 Úvod do temporální logiky 16
3.1.2 Syntaxe a sémantika CTL 17
3.1.3 Praktické CTL formule pro verifikaci 18
3.1.4 TCTL 19
3.2 Verifikační algoritmy 19
3.2.1 Značkovací algoritmus 19
3.2.2 Symbolic model checking s OBDD 21
3.2.3 Bounded model checking 24
3.2.4 Verifikace časových automatů 24
3.3 SMV 26
3.3.1 Modelovací jazyk 26
3.3.2 Verifikace kritérií 29
3.3.3 Přehled implementací SMV 30
3.4 Uppaal 31
3.4.1 Modelování 31
3.4.2 Verifikace 33
3.4.3 Souborové formáty 34

4 Převod automatu do SMV **35**
4.1 Převod Mealyho automatu 35
4.1.1 Příklad 37
4.2 Úpravy postupu 38

4.2.1 Konstantní přiřazení různé od počáteční hodnoty . . . 38
4.2.2 Dosazení konstant do výrazů . . . 39
4.2.3 Neinicializované proměnné . . . 40
4.3 Automat typu Moore . . . 40
4.4 Vstupy automatu . . . 41
4.5 Inicializační procedury . . . 42
4.6 Specifikace verifikovaných formulí . . . 43

5 Převod automatu do Uppaalu 44
5.1 Model automatu . . . 44
5.2 Inicializace . . . 48
5.3 Časovače . . . 50

6 Implementace 53
6.1 Jednotlivé kroky převodu . . . 54
6.1.1 Vložení trans-množiny . . . 54
6.1.2 Přidání časovačů . . . 55
6.1.3 Určení programové inicializační metody . . . 56
6.1.4 Korekce určení typu proměnných . . . 56
6.1.5 Nastavení počátečních hodnot . . . 57
6.1.6 Zjištění množiny použitých proměnných . . . 57
6.1.7 Vytvoření modelu . . . 58
6.2 Možnosti nastavení . . . 58
6.2.1 Modelovací volby . . . 58
6.2.2 Ostatní nastavení . . . 59

7 Příklady převodu a verifikace 60
7.1 Řízení garážových vrat . . . 60
7.2 Řízení čerpací jednotky . . . 63
7.2.1 Verifikovaný program . . . 63
7.2.2 Modelování programu a zápis požadavků . . . 64
7.2.3 Výsledky verifikace . . . 65

8 Závěr 67

Literatura 69

Použité zkratky a symboly 71

Seznam obrázků 72

Seznam tabulek 74

A DTD souborů XML pro Uppaal 75

B SMV model programu řízení vrat 76

C Tutorial **78**
C.1 Přípravná nádrž 79
C.2 Spouštění současným stiskem dvou tlačítek 84

D Obsah přiloženého CD **88**

Rejstřík **89**

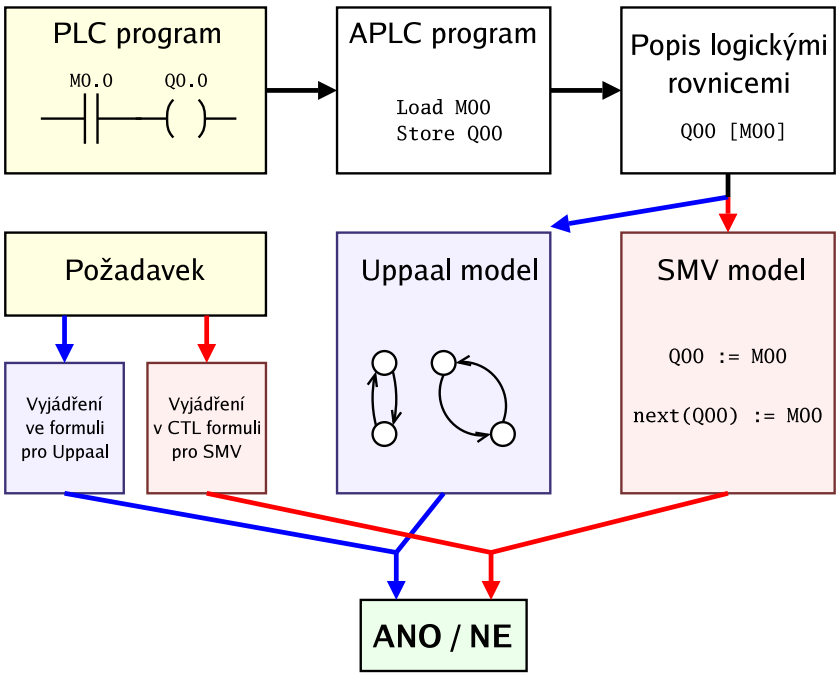
Kapitola 1

Úvod

Verifikace modelu řídicího systému patří mezi možnosti vyšetřování jeho správné funkce před aplikací v reálném světě. Důležitou část řídicí techniky tvoří programovatelné logické automaty (Programmable Logic Controllers, PLC), často využívané k řízení výrobních i nevýrobních procesů, ve kterých může nesprávný návrh řídicího algoritmu způsobit vysoké škody. Verifikační techniky aplikované na PLC programy mohou pomoci k odhalování závažných chyb.

Verifikace PLC programu je proces jehož vstup tvoří verifikovaný program a požadavky na něj kladené. Výstupem je rozhodnutí, zda program splňuje požadavky či nikoliv. V případě nesplnění požadavků může být jako další výstup získána posloupnost stavů programu, která vede k porušení požadavku.

Jednou z možností verifikace PLC programů je postup uvedený na obr. 1.1. PLC program se nejdříve převede do vstupní formy pro algoritmus APLCTRANS [14], který vrátí popis programu soustavou logických rovnic pro použité proměnné. Dalším krokem postupu je mo-



Obrázek 1.1. Kroky verifikace PLC programu pomocí převodu na popis logickými rovnicemi

delování automatu popsaného logickými rovnicemi v některém verifikačním nástroji. Stejně jako se do verifikačního nástroje převádí model programu, je třeba vyjádřit i naše požadavky ve formě, kterou daný nástroj využívá. Většinou se jedná o různé varianty tzv. temporální logiky (kap. 3.1). Na model a formalizované požadavky pak už pouze zbývá spustit verifikační procedury použitého nástroje, které rozhodnou o splnění či nesplnění požadavků.

Vstupem použitého přístupu k verifikaci je program a požadavky (žluté bloky na obr. 1.1), výstupem pak rozhodnutí o splnění požadavků. Červená cesta vyznačuje postup pro verifikaci s SMV a modrá postup pro verifikaci s UPPAALem.

Existují i další metody verifikace PLC programů. Řada z nich je založena na modelování jednotlivých elementů programu – instrukcí nebo příček žebříčkového diagramu. Výsledný model pak vzniká jejich sloučením do modelu pro použitý verifikační nástroj. Jako příklady lze uvést metody popsané v [6] pro SMV a v [15] a [16] pro UPPAAL. Další skupinou modelovacích postupů jsou techniky pracující s programem v podobě sekvenčního diagramu nebo Petriho sítě.

Kapitola 2 ukazuje některé rysy PLC a způsoby, jakými lze pohlížet na jejich činnost při vytváření modelu jejich programu. Je zde také zmíněn algoritmus APLCTRANS použitý pro překlad programu do soustavy logických rovnic. K abstrakci PLC programu jako automatu popsaného soustavou rovnic se váže řada pojmů, z nichž některé využívám k popisu převodu, proto se v této kapitole nacházejí jejich definice.

Třetí kapitola obsahuje úvod do některých verifikačních technik založených na ověřování modelu systému (model checking). Je zde zahrnut i popis verifikačních nástrojů SMV a UPPAAL.

Vlastní návrh postupu modelování automatu popsaného logickými rovnicemi do SMV předkládá kap. 4 včetně některých rozšíření.

Kapitola 5 obsahuje popis modelování automatu v nástroji UPPAAL, o které jsem po dohodě s vedoucím diplomové práce rozšířil rozsah zadání původně zaměřeného pouze na modelování v SMV. Navrženým postupem lze modelovat i PLC programy pracující s některými typy časovačů.

Šestá kapitola zabývající se implementací převodu popisuje především postup specifikace parametrů převodu pomocí prvků uživatelského rozhraní vytvořené aplikace.

V kapitole 7 předkládám příklady použití převodu, výsledky verifikace získaných modelů a srovnání časové náročnosti verifikace modelů získaných různými variantami převodu.

Kapitola 2

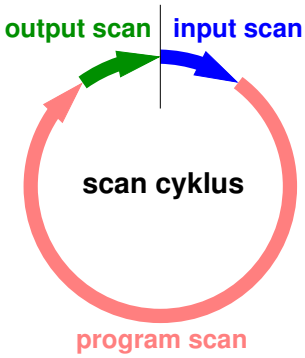
Model činnosti PLC

2.1 Základní rysy PLC

Programovatelné logické automaty (PLC) se využívají pro řízení řady různých úloh, hlavně v oblasti průmyslové automatizace na úrovni řízení procesu. Na trhu se vyskytuje množství různých PLC lišících se v počtu vstupů a výstupů, velikostí paměti, možnostmi rozšíření o přídavné moduly a v mnoha dalších ohledech. Klasická PLC mají společný základní princip činnosti. Program se provádí v cyklech nazývaných *scan cykly*, které dělíme na tři základní fáze,

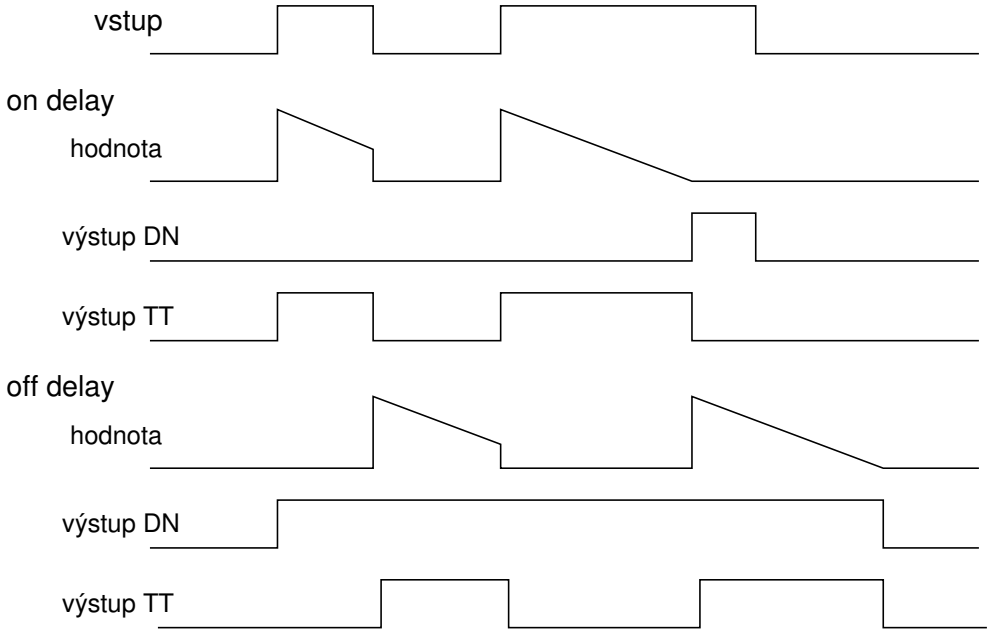
- nejdříve cyklu jsou načteny vstupy PLC do svých obrazů v paměti (*input scan*),
- provede se jeden průchod programem (*program scan*),
- obrazy výstupů se zapíší na fyzické výstupy (*output scan*).

Program scanem rozumíme vykonání uživatelsky naprogramovaného zpracování dat z paměti PLC včetně obrazů vstupů a výstupů. Během průchodu program nepřistupuje k fyzickým vstupům a výstupům, výsledky jeho výpočtu se navenek projeví až během output scanu. Běžně se využívá spuštění scan cyklu ihned po skončení předchozího. Mezi další možnosti může patřit periodické spouštění s danou periodou nebo spouštění řízené událostmi.



Obrázek 2.1. Scan cyklus PLC

Velká část vyráběných PLC umožňuje i složitější chod. Typickými rozšířeními jsou přerušovací systém nebo asynchronní vstupy a výstupy potřebné především v případě využívání



Obrázek 2.2. Odezva časovačů on delay a off delay

komunikace, která je příliš pomalá, než aby ji bylo možné vyřídit během input a output scanu. Některá PLC mohou provádět více programů zároveň.

PLC často umožňují programátorovi provést inicializaci řídicího programu. Mezi obvyklé možnosti patří poskytnutí systémové proměnné, která svou hodnotou indikuje první scan cyklus po spuštění, v programu pak pomocí ní může být detekován první průběh a provedena inicializace. Další variantou je zvláštní programový blok, jenž se vykoná pouze jednou po spuštění programu – poté se cyklicky vykonává hlavní program.

Mezi základní funkce PLC patří čtení a zápis logických a číselných proměnných z a do paměti a operace s nimi. Další typickou funkcí je práce s čítači a časovači. Čítače zvyšují nebo snižují hodnotu své celočíselné stavové proměnné o 1 při vstupním signálu (například při náběžné hraně na jejich vstupu), jejich binární výstup se změní, když jejich hodnota překročí určitou mez. Časovače zvyšují nebo snižují hodnotu své stavové proměnné lineárně v čase, hodnota výstupu se také změní při překročení meze. Jako konkrétní příklady lze uvést časovače typu on delay (TON) a off delay (TOFF). On delay zpožďuje náběžnou hranu výstupu (dn) za náběžnou hranou vstupu. Off delay zpožďuje sestupnou hranu. Obrázek 2.2 představuje odezvu časovačů obou zmíněných typů na stejný vstupní průběh. Časová proměnná časovače je při se při spouštěcí hraně vstupu nastavena na určitou hodnotu (časové zpoždění, timeout), při běhu časovače její hodnota klesá a při dosažení nuly se změní jeho výstup.

K zápisu programů slouží několik druhů programovacích jazyků. Mezi ně patří

IL (instruction list) – seznam instrukcí podobný assembleru, pracuje s instrukcemi pro načtení proměnné (místa v paměti) do akumulátoru, uložení obsahu akumulátoru do paměti, podmíněné a nepodmíněné skoky, logické a aritmetické operace a mnoho dalších,

LD (ladder diagram) – žebříčkový diagram vycházející z reléových schémat,

FBD (function block diagram) – grafický jazyk popisující program jako propojení funkčních

bloků, například bloky logických operací zobrazuje jako hradla,

SFC (Sequential Function Chart) – grafický jazyk, který popisuje program jako sekvenční posloupnost řídicích funkcí,

ST (structured text) – strukturovaný programovací jazyk podobný Pascalu nebo C.

2.2 Klasifikace modelů PLC

Verifikace PLC algoritmu začíná jeho modelováním v použitém verifikačním nástroji. Způsob, jakým program modelujeme, rozhoduje o tom, jak přesně výsledný model popisuje ověřovaný systém a o tom, jaké paměťové a časové nároky bude klást výpočet zjišťující splnění požadavků na program. Způsoby modelování PLC programů můžeme rozdělit podle jejich přístupu ke scan cyklu, jak uvádějí [9, 14]. Některé modely popisují poměrně přesně chování PLC včetně časových charakteristik, jiné výrazně zjednodušují pohled na činnost PLC.

Statické modely PLC nepopisují cyklické chování ani časové charakteristiky scan cyklu. Mohou být použity například pro určení datové nezávislosti paralelních částí nebo pro detekci nedosažitelného kódu.

Explicitní modely obvykle mívají určenou horní a dolní mez doby vykonávání scan cyklu. Explicitní modely jsou z popisovaných nejblíže realitě a hodí se pro důsledné zkoumání časových souvislostí. S jejich pomocí lze modelovat i asynchronní komunikaci.

Implicitní modely popisují cyklické chování PLC, ale doba jednoho scan cyklu je pevně dána. Jsou tudíž méně obecné než explicitní modely. Hodí se například pro modelování PLC s konstantní dobou scan cyklu. Uplatní se i tam, kde je kritická pouze maximální doba běhu cyklu a místo určení horní a dolní meze nastavíme dobu scan cyklu modelu na maximální dobu cyklu skutečného PLC.

Abstraktní modely předpokládají, že všechny fáze scan cyklu proběhnou v nulovém čase. Můžeme jimi modelovat programy, které řídí procesy velmi pomalé ve srovnání s PLC. Interpretace cyklu nulové délky v prostředí reálného času může vést k situaci, že během konečné doby proběhne nekonečně mnoho scan cyklů. Tomuto jevu lze zabránit více způsoby. Jedním z nich je spouštět scan cyklus jen v okamžicích změny vstupních signálů. Další možností je scan cyklus spouštět v pevných nebo proměnných časových intervalech. Výsledný model se potom podobá svým charakterem explicitnímu resp. implicitnímu. Nulová doba běhu jednoho cyklu ale způsobuje

- nemožnost modelovat asynchronní komunikaci,
- modelování časovačů je nepřesné v tom smyslu, že časovač nezíská žádné zpoždění mezi svým spuštěním a koncem scan cyklu.

2.3 Abstraktní PLC

Abstraktní PLC (APLC machine, [14]) tvoří základ pro převod PLC programu v IL formě do popisu určujícího výstup po program scanu jako funkci hodnot proměnných před jeho

provedením. APLC pracuje s instrukcemi jako s operacemi nad konfigurací $\langle C, f_{reg}, E, S, D\rangle$, kde

- C je kód, seznam instrukcí k provedení,
- f_{reg} je flag register,
- E je zásobník pro logické hodnoty z f_{reg},
- S je množina vstupních, vnitřních a výstupních proměnných,
- D je úložiště, které je buď prázdné nebo obsahuje množinu $\langle C', f'_{reg}, E', S', D'\rangle$.

Instrukce APLC programu je operace nad konfigurací. APLC program jako seznam instrukcí lze algoritmem APLCTRANS [14] zpracovat jako kompozici operací pro jednotlivé instrukce. Operaci může popisovat množina přiřazení jednotlivým proměnným nazývaná trans-množina (kap. 2.3.2). Kompozicí trans-množin pro jednotlivé operace vznikne trans-množina celého programu. Jednotlivá přiřazení výsledné množiny určují hodnoty proměnných po průchodu programu jako funkce hodnot proměnných před jeho provedením.

APLC program může obsahovat instrukce pracující s binárními proměnnými a zásobníkem a instrukce skoku, například

INIT nastaví f_{reg} na 0 a vyprázdní zásobník,

LOAD načte argument do f_{reg},

AND uloží do f_{reg} jeho logický součin s argumentem instrukce,

STORE uloží obsah f_{reg} do proměnné,

JP provede nepodmíněný skok.

Program může obsahovat pouze dopředné skoky.

2.3.1 Model binárního PLC

Pokud se omezíme na pouze na základní cyklickou činnost a binární proměnné bez použití časovačů, můžeme program PLC popsat automatem jak uvádí [13, 14]. Nejprve definujme pojem abecedy nad množinou binárních proměnných.

Definice 2.1 Nechť je V je neprázdná uspořádaná množina $n = |V|$ binárních proměnných. Abecedou generovanou množinou V binárních proměnných nazveme množinu všech n-tic $\alpha(V) = \{0, 1\}^n$, kde $\{0, 1\}^n$ označuje n-násobný kartézský součin. □

PLC s programem omezeným na práci s binárními proměnnými můžeme definovat jako binární PLC. Jeho stav určuje stav vnitřních a výstupních proměnných. Program je zobrazení z abecedy generované množinou všech vstupních, vnitřních a výstupních proměnných do abecedy generované množinou vnitřních a výstupních proměnných. Jinak řečeno, hodnoty vnitřních a výstupních proměnných program určí jako funkce hodnot vstupních, vnitřních a výstupních proměnných.

Definice 2.2 Binární PLC je šestice

$$BPLC = \langle \Sigma, \Omega, V, A_\Sigma, \delta_\mathrm{P}, q_0 \rangle,$$

kde

- Σ je neprázdná konečná uspořádaná množina binárních vstupů, tj. obsah paměti obrazu vstupů,
- Ω neprázdná konečná uspořádaná množina binárních výstupů, tj. obsah paměti obrazu výstupů,
- V neprázdná konečná uspořádaná množina binárních vnitřních proměnných,
- A_Σ neprázdná podmnožina $A_\Sigma \subseteq \alpha(\Sigma)$ rozpoznaných vstupů,
- δ_P PLC program popsaný jako zobrazení $\delta_\mathrm{P}(q, x) : \alpha(V) \times \alpha(\Omega) \times A_\Sigma \rightarrow \alpha(V) \times \alpha(\Omega)$, kde $q \in \alpha(V) \times \alpha(\Omega)$ a $x \in A_\Sigma$,
- q_0 počáteční stav PLC programu, $q_0 \in \alpha(V) \times \alpha\Omega)$.

□

Zobrazení δ_P nemusí být definované pro všechny možné kombinace vstupních proměnných. Pro některé vstupy ($\notin A_\Sigma$) program neprovede žádný zápis do proměnné z $V \cup \Omega$, pro ně není zobrazení δ_P definované. Funkci $BPLC$ popisuje například Mooreův automat generovaný $BPLC$ podle následující definice.

Definice 2.3 Nechť $BPLC = \langle \Sigma, \Omega, V, A_\Sigma, \delta_\mathrm{P}, q_0 \rangle$ je binární PLC, pak automat M generovaný $BPLC$ je šestice

$$M = \langle X, Y, Q, \delta, q_0, \omega \rangle,$$

kde je

- X vstupní abeceda, $X = \alpha(\Sigma)$,
- Y výstupní abeceda, $Y = \alpha(\Omega)$,
- Q množina stavů automatu, $Q = \alpha(V) \times Y$,
- δ přechodová funkce definovaná jako zobrazení

$$\delta = \begin{cases} \delta_\mathrm{P}(q, x) \text{ pro všechna } \langle q, x \rangle \in Q \times X, \text{ pro která je } \delta_\mathrm{P} \text{ definováno,} \\ q \text{ v ostatních případech,} \end{cases}$$

- q_0 počáteční stav automatu,
- ω výstupní funkce $\omega(\langle v, y \rangle) = y$, kde $\langle v, y \rangle \in Q$, $y \in Y$ a $v \in \alpha(V)$.

□

Množina dosažitelných stavů automatu může mít velikost až $2^{|V|+|\Omega|}$, roste tedy exponenciálně s počtem proměnných. Výstup PLC nezávisí na okamžitých hodnotách vstupů, ale pouze na stavu automatu, jde tedy o automat typu Moore. Jeden scan cyklus odpovídá jednomu přechodu mezi stavy automatu. Výstupní funkce pouze určuje, který výstup je určený kterou stavovou proměnnou (prvkem vektoru $q \in \alpha(V) \times Y$) automatu.

2.3.2 Trans-množiny

APLC program a jeho instrukce se dají popsat množinami přiřazení funkcí proměnných před provedením operace hodnotám proměnných po provedení operace. Jedno takové přiřazení pro proměnnou $b \in S$ můžeme zapsat jako $b := [\![e]\!]\, S$, kde e je výraz nad proměnnými z S ohodnocený momentálním stavem S. Nad takovými přiřazeními a jejich množinami [14, 12] definují pojmy, z nichž některé budou použity k popisu automatu generovaného APLC programem a jeho převodu do SMV a UPPAALu.

Definice 2.4 Nechť $bexp$ je přípustný výraz dle nějaké gramtiky $Gbexp$, pak domain $bexp$ je definovaný jako:

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

□

Příslušnou gramatiku může pro náš případ tvořit libovolná gramatika generující booleovský výraz s operátory $\neg$,$\vee$ a $\wedge$.

Definice 2.5 Nechť $b \in S$ je libovolná binární proměnná z konečné paměti S, $bexp$ je libovolný výraz dle gramatiky $Gbexp$ a splňující $\mathrm{dom}(bexp) \subset S$ a $b := [\![bexp]\!]\, S$ je přiřazovací operace $bexp$ pro proměnnou b počítaná vzhledem k hodnotám v S. Definujeme

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

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

Definice 2.6 (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 $\mathrm{co}(\hat{x_i}) = co(\hat{x_j})$ implikuje $i = j$. Množinu všech trans-množin pro S proměnné označíme jako $\widehat{S}(S)$, čili $\widehat{X} \in \widehat{S}(S)$. □

Jinými slovy, trans-množina obsahuje pro každou proměnnou z S nejvýše jedno t-přiřazení. Všechna přiřazení se vykonávají současně, to znamená, že se nejprve vyhodnotí všechny výrazy a teprve poté se všem proměnným x_i, pro něž existuje $\hat{x}_i \in \widehat{C}$, přiřadí vypočtené hodnoty.

Definice 2.7 Binární relace $\widehat{\in}$ na množině S a $\widehat{\mathcal{B}}(S)$ je definována pro všechna $\widehat{X} \in \widehat{S}(S)$ a $x \in S$ jako

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

□

Definice 2.8 Nechť $\widehat{X} \in \widehat{S}(S)$ 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

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

□

Rozšíření přidává do trans-množiny kanonická přiřazení pro proměnné, jejichž přiřazení nebyla v původní trans-množině zastoupena. K operaci rozšíření $\uparrow$ existuje i opačná operace $\downarrow$ (komprese), která z trans-množiny odebírá kanonická přiřazení.

Definice 2.9 Nechť $\widehat{X} \in \widehat{S}(S)$ je libovolná trans-množina definovaná na S, pak její *codomain* a *domain* se rovnají

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

□

Codomain je množinou všech proměnných, pro které existuje t-přiřazení v dané trans-množině. Domain obsahuje právě ty proměnné, které se vyskytují alespoň ve výrazu $bexp_i$ alespoň jednoho t-přiřazení $\hat{x}_i$ v dané trans-množině.

Mezi trans-množinou a množinou proměnných můžeme definovat průnik.

Definice 2.10 Nechť $\widehat{X} \in \widehat{S}(S)$ je trans-množina definovaná na množině proměnných S a U je podmnožina S, pak $\widehat{X} \,\widehat{\cap}\, U = \left\{ \hat{x} \in \widehat{X} \mid \mathrm{co}(\hat{x}) \in U \right\}$ □

Definice 2.11 Nechť $\widehat{X} \in \widehat{S}(S)$ je libovolná trans-množina definovaná na S a $(\Omega \cup V) \subseteq S$, pak definujeme

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

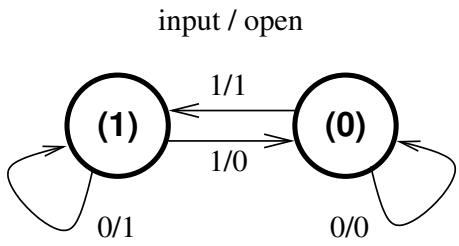
Trans-množina $\widehat{\mathrm{dom_p}}(\widehat{X})$ se nazývá *PLC domain* trans-množiny $\widehat{X}$. Trans-množina $\widehat{\mathrm{co_p}}(\widehat{X})$ a množina $\mathrm{co_p}(\widehat{X})$ se nazývají *PLC codomain* trans-množiny $\widehat{X}$. □

2.3.3 Automat generovaný APLC programem

Překladem APLC programu algoritmem APLCTRANS získáme trans-množinu obsahující t-přiřazení pro proměnné APLC. Jak můžeme pomocí této trans-množiny sestrojit automat, který vystihuje činnost APLC programu? Jednou z možností by mohlo být popsat trans-množinou přechodovou funkci Mooreova automatu generovaného binárním PLC (def 2.3). Binární PLC ale popisuje i Mealyho automat, který může mít méně stavů než automat typu Moore.

```
load !input
jpc end
load !open
store open
end:
end
```

```
Composed in 0.04 seconds.
Decodeing result...
Result={ @f[1],
open[!input.open+input.!open] }
```



Obrázek 2.3. Program APLC, výsledek překladu, graf přechodů automatu

Pro definici přechodové funkce automatu využijeme mapování trans-množin do abecedy $\alpha()$ generované množinou použitých binárních proměnných. Mapování $\mathrm{map_p}(\widehat{X})$: $\alpha(\mathrm{dom}(\widehat{X})) \to \alpha(\mathrm{co_p}(\widehat{X}))$ určuje vektor hodnot pro proměnné z $\mathrm{co_p}(\widehat{X})$ jako funkci hodnot proměnných z $\mathrm{dom}(\widehat{X})$. Funkce je pro jednotlivé prvky vektoru dána t-přiřazeními pro příslušné proměnné. Pro přesnou definici $\mathrm{map_p}$ viz [14].

Definice 2.12 Nechť AL je APLC program definovaný na vstupech Σ, výstupech Ω a vnitřních proměnných V. Jestliže trans-množina $\widehat{C}$ programu byla vytvořena algoritmem APLCTRANS, pak automat generovaný AL je šestice

$$M(\widehat{C}) = \langle X, Y, Q, \delta, q_0, \omega \rangle,$$

kde

$X = \alpha\left(\mathrm{dom}(\widehat{C}) \cap \Sigma\right)$	je vstupní abeceda,
$Y = \alpha\left(\mathrm{co}(\widehat{C}) \cap \Omega\right)$	výstupní abeceda,
$Q = \alpha\left(\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C})\right)$	množina stavů,
$\delta = \mathrm{map_p}\left(\widehat{\mathrm{co_p}}(\widehat{C}) \cap \widehat{\mathrm{dom_p}}(\widehat{C})\right)$	přechodová funkce $\delta : Q \times X \to Q$,
q_0	počáteční stav automatu,
$y = \mathrm{map_p}\left(\widehat{C} \widehat{\cap} \Omega\right)$	výstupní funkce $\omega : Q \times X \to Y$,

□

Výsledný automat má stav definovaný hodnotami proměnných z $\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C})$. Množina těchto stavových proměnných je podmnožinou stavových proměnných automatu z def. 2.3 ($V \cup \Omega$), protože $\mathrm{co_p}(\widehat{C}) \subseteq V \cup \Omega$. Výstup obecně nezávisí jen na stavu automatu, ale i na jeho momentálním vstupu, jedná se tedy o automat typu Mealy.

Program z obr. 2.3 má jediný vstup `input` a výstup `open`. T-přiřazení pro `open` není kanonické, proto $\mathrm{co_p}(\widehat{C}) = \{\mathtt{open}\}$. PLC domain získané trans-množiny je $\mathrm{dom}(\{\widehat{\mathtt{open}}\}) = \{\mathtt{input}, \mathtt{open}\}$. Množina stavů generovaného automatu je $\alpha(\mathtt{open}) = \{(0), (1)\}$. Přechodovou i výstupní funkci určuje výraz pro `open`. Graf přechodů výsledného automatu je na obr. 2.3 vpravo.

Kapitola 3

Model Checking

S rozvojem počítačových a řídicích systémů se ukazuje důležitost možnosti ověření jejich správného návrhu. Základní možností je testování výsledného produktu (číslicového obvodu, programu, . . .). Jako další způsob ověřování vznikly formální verifikační metody (techniky).

Testování je založeno na vyzkoušení funkčnosti systému na omezeném množství různých vstupních posloupností, jeho kladný výsledek tedy nezaručuje, že bude výsledek příznivý i pro netestované situace. Naproti tomu verifikační techniky ověřují vlastnosti či chování systému za všech okolností.

Model checking znamená verifikaci ověřující, zda model systému splňuje zadaný požadavek. Verifikace probíhá zcela automaticky podle určitého algoritmu. Mezi rysy metody model checking patří

- skutečnost, že není ověřován přímo samotný systém, ale jeho model, který je pouze jeho abstrakcí,
- požadavky na systém jsou zadávany v *temporální logice*, která nemusí umožňit postihnutí všech našich požadavků,
- časová složitost samotné verifikace má obecně exponenciální závislost na velikosti verifikovaného modelu.

3.1 Temporální logika

Temporální logika byla navržena pro vytváření výroků o změnách v čase. Jejím zvláštním případem je *Computation tree logic* [7, 10], zkráceně CTL, jež je využívána jako prostředek pro vyjádření požadavků na verifikovaný systém v nástroji SMV. Z CTL vychází *Timed CTL* (TCTL), jejíž podmnožinu lze využít pro specifikaci požadavků v UPPAAL.

3.1.1 Úvod do temporální logiky

Základ temporální logiky tvoří skutečnost, že výrok nemusí být pouze staticky pravdivý nebo nepravdivý, ale jeho pravdivost se může měnit s časem. Pravdivost formule vytvořené podle pravidel výrokové logiky je v temporální logice posuzovaná vzhledem k jednomu stavu systému v jednom časovém okamžiku. Stav systému se mění s časem.

Abychom mohli vytvářet výroky o pravdivosti v čase, má temporální logika časové operátory. Formule $\mathrm{F}\phi$ je v přítomnosti pravdivá, jestliže je ϕ pravdivá v některém okamžiku budoucnosti. Formule $\mathrm{G}\phi$ odpovídá $\neg\mathrm{F}\neg\phi$, je tedy pravdivá právě tehdy, když je ϕ pravdivá v každém okamžiku budoucnosti. Podobné operátory existují i pro minulost.

Obvykle myslíme časem lineárně uspořádanou množinu. Pro dva stavy s a t z množiny stavů vývoje systému platí buď $s < t$, $s = t$, nebo $t < s$. Relace $<$ vyjadřuje časové pořadí. Můžeme pak psát, že formule $\mathrm{F}\phi$ je pravdivá ve stavu s tehdy a jen tehdy, když existuje stav t takový, že $s < t$ a ϕ je v t pravdivá.

Temporální logika bývá rozšířena o operátor U (until). Formule $\phi\,\mathrm{U}\,\psi$ je pravdivá ve stavu s, jestliže existuje nějaký stav t takový, že $s < t$, formule ψ je v t pravdivá a pro všechny u splňující $s < u < t$ je pravdivá ϕ.

Pokud uvažujeme čas jako diskrétní veličinu, má každý stav přímého předchůdce a následníka. Můžeme pak definovat operátor příštího okamžiku. Formule $\mathrm{X}\phi$ je pravdivá ve stavu s, když existuje jeho přímý následník t, ve kterém je ϕ pravdivá.

Dalším pohledem na čas může být tzv. *branching-time*. Vývoj stavu má stromovou strukturu. Minulost každého stavu je jednoznačná, lineárně uspořádaná, ale jeho budoucnost je nedeterministická. Formule mohou nabývat různých pravdivostních hodnot v různých větvích budoucího vývoje. Proto byly zavedeny další operátory. Operátor A znamená „ve všech větvích“, E znamená „alespoň v jedné větvi“. Například $\mathrm{EF}\phi$ je pravdivá, jestliže alespoň v jedné větvi budoucího vývoje existuje nejméně jeden stav, ve kterém je ϕ pravdivá.

3.1.2 Syntaxe a sémantika CTL

CTL je temporální logikou využívající branching-time model času. Definici jejích formulí uvádí [7].

Definice 3.1 CTL formule definujeme induktivně pomocí Backus-Naurovy formy:

$$\begin{aligned}\phi \quad ::= \quad & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid \mathrm{AX}\phi \mid \mathrm{EX}\phi \mid \\ & \mathrm{A}[\phi\,\mathrm{U}\,\phi] \;\mid\; \mathrm{E}[\phi\,\mathrm{U}\,\phi] \;\mid\; \mathrm{AG}\phi \;\mid\; \mathrm{EG}\phi \;\mid\; \mathrm{AF}\phi \;\mid\; \mathrm{EF}\phi,\end{aligned}$$

kde p je atomická formule a $\top$, $\bot$ jsou symboly pro tautologii, resp. kontradikci. □

Operátory času ani operátory větví (A a E) se podle definice nemohou v CTL formuli nikdy vyskytnout samostatně, ale vždy jen ve dvojici. Časovému operátoru musí vždy předcházet operátor větví. CTL nemá operátory pro minulost.

Definice 3.2 Kripkeho model (Kripkeho struktura) je $\mathcal{M} = (S, \rightarrow, L)$. S je množina stavů, přechodová relace $\rightarrow$ je binární relace na S taková, že pro každý stav $s \in S$ existuje $s' \in S$ takový, že $s \rightarrow s'$. L je značkovací funkce, která každému stavu s přiřazuje množinu $L(s)$ atomických výroků, které jsou v s pravdivé. □

Definice 3.3 To, že CTL formule ϕ je na modelu $\mathcal{M}$ splněna ve stavu s zapíšeme $\mathcal{M}, s \models \phi$. Relace $\models$ je dána strukturální indukcí na všech CTL formulích. Celá definice pro všechny tvary formulí z def. 3.1 se nachází v [7]. Zde jsou uvedeny pouze příklady $\models$ pro některé tvary

- $\mathcal{M}, s \models \top$ a $\mathcal{M}, s \not\models \bot$ ve všech $s \in S$,
- $\mathcal{M}, s \models p$ iff $p \in L(s)$,
- $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$,
- $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \neg\phi_1$ a $\mathcal{M}, s \models \neg\phi_2$,
- $\mathcal{M}, s \models \mathrm{EX}\phi$ iff existuje s_1 takové, že $s \to s_1$ a platí $\mathcal{M}, s_1 \models \phi$, tedy EX říká „alespoň v jednom přímo následujícím stavu“,
- $\mathcal{M}, s \models \mathrm{AF}\phi$ iff pro všechny cesty $s_1 \to s_2 \to s_3 \to \ldots$, kde $s_1 \equiv s$, existuje alespoň jeden s_i takový, že $\mathcal{M}, s_i \models \phi$,
- $\mathcal{M}, s \models \mathrm{E}[\phi_1 \,\mathrm{U}\, \phi_2]$ iff existuje cesta $s_1 \to s_2 \to s_3 \to \ldots$, kde $s_1 \equiv s$ taková, že na ní platí $\phi_1 \,\mathrm{U}\, \phi_2$, tedy existuje s_i na této cestě takové, že $\mathcal{M}, s_i \models \phi_2$ a pro všechny $j < i$ je $\mathcal{M}, s_j \models \phi_1$. □

V posledních dvou případech budoucnost obsahuje přítomnost. V CTL totiž všechny větve budoucího vývoje od stavu s obsahují i samotný stav s. Z toho vyplývá například tautologie $\phi \Rightarrow \mathrm{AF}\phi$, protože pro jakýkoliv stav s_0 platí: Je-li splněna ϕ ve stavu s_0, pak všechny větve budoucího vývoje obsahují alespoň jeden stav, kde je splněna ϕ – tím stavem je s_0.

Mezi formulemi platí ekvivalence

$$\begin{array}{rclrcl}
\neg\mathrm{AF}\,\phi & \equiv & \mathrm{EG}\,\neg\phi, & \neg\mathrm{EF}\,\phi & \equiv & \mathrm{AG}\,\neg\phi, \\
\mathrm{AF}\,\phi & \equiv & \mathrm{A}\,[\top \,\mathrm{U}\, \phi], & \mathrm{EF}\,\phi & \equiv & \mathrm{E}\,[\top \,\mathrm{U}\, \phi], \\
\neg\mathrm{AX}\,\phi & \equiv & \mathrm{EX}\,\neg\phi, & \mathrm{A}\,[p \,\mathrm{U}\, q] & \equiv & \neg(\mathrm{E}[\neg q \,\mathrm{U}\, (\neg p \wedge \neg q)] \wedge \mathrm{EG}\,\neg q).
\end{array}$$

Lze dokázat že úplným systémem spojek pro CTL jsou i množiny s méně spojkami než použitými v def. 3.1, například $\{\bot, \neg, \wedge, \mathrm{AF}, \mathrm{EU}, \mathrm{EX}\}$, ostatní spojky můžeme pomocí uvedených ekvivalencí nahradit.

3.1.3 Praktické CTL formule pro verifikaci

Jak bylo výše uvedeno, CTL je důležitým prostředkem pro vyjádření požadavků na systém pro účely verifikace formálním nástojem. Příklady takových požadavků s použitím několika atomických formulí, které mohou představovat například proměnné řídicího systému, jsou

- absence deadlocku testovaná na jedné proměnné, např. $\mathrm{AG}\,((\mathrm{EF}\,run) \wedge \mathrm{EF}\,\neg run)$, vždy existuje nějaká cesta, ve které lze dosáhnout, aby zařízení běželo, i cesta, jak dosáhnout, aby zařízení neběželo,
- vzájemné vyloučení, například $\mathrm{AG}\,\neg(closing \wedge opening)$, nikdy nevysíláme zároveň řídicí signál pro otevření i pro zavření,
- okamžitý nutný důsledek, okamžitá reakce, například $\mathrm{AG}\,(error \Rightarrow \mathrm{AX}\,\neg opening)$, jestliže byla detekována chyba, nesmí být v následujícím okamžiku nastaven signál $opening$,
- nutný důsledek bez nároku na rychlost, $\mathrm{AG}\,(requested \Rightarrow \mathrm{AF}\,acknowledged)$, po každém požadavku musí někdy v budoucnu následovat potvrzení,

- $\mathrm{AG}\,(topLimit \Rightarrow \mathrm{AX}\ \mathrm{A}[\neg opening\,\mathrm{U}\,\neg topLimit])$ znamená, že pokud je dosažen $topLimit$, nebude od následujícího okamžiku $opening$ nastaven až do té doby, než bude $topLimit$ opuštěn.

3.1.4 TCTL

Logika CTL popisuje vzájemné pořadí v čase, neumožňuje však čas kvantifikovat. Bylo proto navrženo její rozšíření, TCTL (Timed computation tree logic, [1]). Spojky jako EF nebo AG jsou rozšířeny o časové omezení. Například formule $\mathrm{EF}_{<5}\,p$ znamená, že existuje cesta, kde je formule p splněna pro nějaký čas menší než 5 časových jednotek.

Mezi formule TCTL patří

- $\mathrm{AF}_{\sim c}\,\phi$ – ve všech cestách existuje v čase $t \sim c$ stav, kde platí ϕ,
- $\mathrm{E}[\phi_1 \mathrm{U}_{\sim c}\,\phi_2]$ – existuje cesta taková, že na ní v čase $t \sim c$ je stav, kde platí ϕ_2 a v jakémkoliv čase $0 \leq t' \leq t$ na této cestě platí ϕ_1.

Znak $\sim$ představuje relaci $<, \leq, =, \geq,$ nebo $>$ a c je přirozené číslo. Celou defininici TCTL uvádí [1].

Poznamenejme, že čas je v TCTL spojitý, nemůže být tedy definován následující časový okamžik. Proto nemá operátor X.

3.2 Verifikační algoritmy

Verifikační úloha model checking řeší otázku, zda model s daným počátečním stavem splňuje požadovanou formuli, vyjádřenou např. v CTL logice

$$\mathcal{M}, s_0 \overset{?}{\models} \phi,$$

nebo otázku nalezení všech stavů, ve kterých je daná formule splněná.

3.2.1 Značkovací algoritmus

Značkovací algoritmus uvedený v [7] dává základní představu, jak je možné verifikovat CTL formule. Jeho vstupem je model $\mathcal{M}$ (podle def. 3.2) a verifikovaná CTL formule ϕ, výstupem pak množina všech stavů, které splňují ϕ. Pracuje tak, že značkuje stavy formulemi, podle toho, zda jsou v nich splněny či nikoliv. Nejprve značkuje stavy atomickými formuleni, poté zespodu značkuje stále složitějšími subformulemi verifikované formule ϕ až nakonec označí stavy samotnou ϕ, jí označené stavy jsou výstupem algoritmu.

Rekurzivní značkování probíhá podle tvaru formule následujícím předpisem. Formulemi ψ_1 a ψ_2 je již model označkován. Předpis je uveden pouze pro úplný systém spojek $\{\bot, \neg, \wedge, \mathrm{AF}, \mathrm{EU}, \mathrm{EX}\}$.

- $\bot$: žádný stav neoznač $\bot$.
- p: označ všechny stavy, pro které $p \in L(s)$.

- $\psi_1 \wedge \psi_2$: označ stavy, které jsou již označené formulí ψ_1 i ψ_2.
- $\neg\psi_1$: označ všechny stavy neoznačené ψ_1.
- $\mathrm{AF}\,\psi_1$: všechny stavy označené ψ_1 označ i formulí $\mathrm{AF}\,\psi_1$, opakuj: Jestliže existuje neoznačený stav s, který má všechny následníky označené $\mathrm{AF}\,\psi_1$, pak jí označ i s. Když žádný další stav splňující danou podmínku neexistuje, skonči.
- $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$: všechny stavy označené ψ_2 označ i formulí $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$, opakuj: Označ stav, který je označen ψ_1 a alespoň jeden jeho následník je již označen $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$. Když není možná žádná další změna označení, skonči.
- $\mathrm{EX}\,\psi_1$: označkuj všechny stavy, jejichž alespoň jeden následník je označen ψ_1.

Složitost značkovacího algoritmu je $\mathcal{O}(f \cdot V \cdot (V + E))$, kde f je počet spojek ve formuli, V počet stavů modelu a E počet jeho hran.

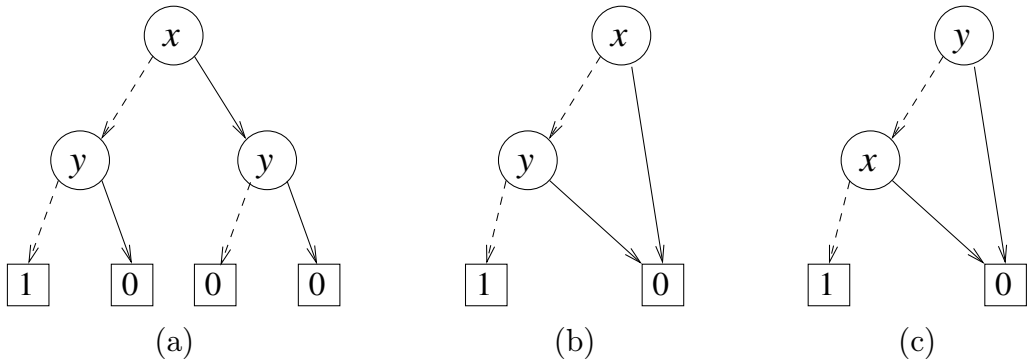
Formule $\mathrm{EG}\,\psi$ se může vyjádřit jako $\neg\mathrm{AF}\neg\psi$ a verifikovat výše uvedeným předpisem, můžeme jí ale značkovat stavy přímo efektivnějším předpisem založeným na hledání silně souvislých komponent popsaný v [7]. Značkovací algoritmus využívající tento předpis pro $\mathrm{EG}\,\psi$ místo předpisu pro $\mathrm{AF}\,\psi$ (tedy definovaného pro úplný systém spojek $\{\bot, \neg, \wedge, \mathrm{EG}, \mathrm{EU}, \mathrm{EX}\}$), má složitost $\mathcal{O}(f \cdot (V + E))$. Složitost algoritmu závisí lineárně na velikosti modelu. Bohužel, velikost modelu může růst exponenciální vzhledem k počtu proměnných systému. Pro snížení výpočetní náročnosti verifikace se používají různé metody, například

Datové struktury popisující verifikovaný model efektivním způsobem, např. seřazené binární rozhodovací diagramy (ordered binary decision diagrams, OBDD), které nepopisují systém výčtem jeho stavů, ale reprezentují celé množiny stavů, zmiňuje se o nich kap. 3.2.2.

Abstrakce nahradí model jiným, menším modelem, který je s původním modelem ekvivalentní z pohledu verifikované formule. Jednoduchým způsobem abstrakce je odstranění proměnných, které nemají na danou formuli vliv – *cone of influence* (COI) redukce [4]. Pro systém popsaný rovnicemi $x_i = f(x_0, \ldots, x_n)$ lze COI redukci provést tak, že

1. do množiny proměnných, které mají vliv na formuli (množina vlivu), dáme proměnné obsažené ve formuli,
2. opakovaně přidáváme proměnné, které se vyskytují na pravé straně rovnice pro některou z proměnných v množině vlivu,
3. skončíme, když neexistuje proměnná, která by byla na pravé straně rovnice pro některou z proměnných z množiny vlivu a sama by v ní nebyla,
4. z modelu odstraníme všechny proměnné, které neobsahuje množina vlivu, a také rovnice, které určují jejich hodnoty.

Trans-množina (def. 2.6) obsahuje t-přiřazení – rovnice, na jejichž levých stranách se vždy vyskytuje jedna proměnná a na pravých stranách funkce představovaná logickým výrazem a posunem o jedno provedení přiřazení. Aplikací COI redukce na trans-množinu popisující přechodovou funkci automatu můžeme dosáhnout odstranění některých proměnných a jejich t-přiřazení a tím zjednodušení modelu.



Obrázek 3.1. Různé OBDD reprezentující stejnou funkci

3.2.2 Symbolic model checking s OBDD

Symbolic model checking znamená verifikační techniku pracující s množinami stavů namísto práce s jednotlivými stavy. Mezi prostředky pro reprezentaci množiny stavů patří seřazený binární rozhodovací diagram (OBDD, [7, 10]).

OBDD reprezentuje logickou funkci orientovaným acyklickým grafem s jedním kořenem a dvěma druhy uzlů. Koncové uzly, listy, se označují logickou hodnotou (0 nebo 1). Nekoncové uzly jsou označeny proměnnou, podle jejíž hodnoty se v daném uzlu strom větví. Z každého nekoncového uzlu vedou dvě hrany, jedna pro logickou 0, druhá pro 1. Ohodnocení funkce určuje cestu v grafu: V každém nekoncovém uzlu sledujeme hranu pro danou hodnotu příslušné proměnné, koncový uzel určuje hodnotu funkce. Na obr. 3.1 jsou tři různé OBDD reprezentující funkci $f = \neg x \wedge \neg y$ (čárkovaná čára představuje hranu pro log. 0).

OBDD má určené pořadí proměnných, podle kterého se vyskytují na cestě od kořene do listu. Diagramy na obr. 3.1(a) a 3.1(b) mají řazení $\langle x, y \rangle$, diagram 3.1(c) má řazení $\langle y, x \rangle$. Žádná orientovaná cesta v grafu nesmí pořadí porušit – je-li proměnná x_1 v pořadí před x_2, nesmí se na žádné cestě objevit x_2 před x_1, ale v grafu může být cesta, kde se některá z proměnných nevyskytuje. OBDD na obr. 3.1(a) může mít kromě řazení $\langle x, y \rangle$ např. i $\langle k, x, l, y, m \rangle$ – obě řazení jsou pro daný OBDD kompatibilní (diagram vytvořený podle jednoho z nich neporušuje druhé).

Některé uzly a hrany v grafu 3.1(a) jsou zbytečné, stejnou funkci lze se stejným řazením reprezentovat i menším grafem. Jak redukovat OBDD? Odpovědí jsou tři kroky, které můžeme opakovat do doby, kdy již žádný z nich nelze provést:

Odstranění duplicitních listů. Jestliže OBDD má více než jeden list pro 0, zachováme pouze jeden z nich a hrany původně vedoucí do odstraněných uzlů do něj přesměrujeme. Stejné provedeme s listy pro 0.

Odstranění redundantních hran. Pokud z některého uzlu n vedou obě hrany do jednoho uzlu m, odstraníme n a všechny hrany, které do něj vedly, přesměrujeme do m.

Odstranění duplicitních nekoncových uzlů. Jestliže jsou dva uzly n a m kořeny dvou identických podgrafů, jeden z uzlů odstraníme včetně jeho podgrafu a hrany vedoucí do něj přesměrujeme do druhého.

Provedením těchto operací získáme z grafu na obr. 3.1(a) graf 3.1(b), na který již nelze žádný z kroků aplikovat.

Věta 1 Redukovaný OBDD reprezentující danou funkci f je unikátní. Nechť B_1 a B_2 jsou dva OBDD s kompatibilním řazením proměnných. Jestliže oba reprezentují stejnou logickou funkci, pak mají identickou strukturu. □

Věta 1 umožňuje pomocí OBDD provádět řadu testů nad logickými funkcemi, například

- jestliže jsou dvě funkce reprezentované diagramy se stejným řazením, můžeme rozhodnout o jejich sémantické ekvivalenci redukcí jejich diagramů – jejich redukované diagramy jsou stejné tehdy a jen tehdy, když jsou si obě funkce ekvivalentní,
- funkce ekvivalentní s tautologií má redukovaný OBDD obsahující pouze jeden uzel – list pro 1,
- splnitelná funkce (taková, že alespoň pro jedno pravdivostní ohodnocení je pravdivá) nemá redukovaný graf obsahující pouze list pro 0.

Pro operace s redukovanými OBDD existují efektivní algoritmy, [7] uvádí

`reduce` pro redukci OBDD,

`apply` pro logické operace. Vstupem jsou dva redukované OBDD reprezentující logické funkce, výstupem pak redukovaný OBDD reprezentující výsledek operace. Například `apply(`$\cdot$, B_f, B_g) vrátí výsledek operace logického součinu nad funkcemi f a g.

`restrict` provede dosazení hodnoty za proměnnou, např. `restrict`(0, x, B_f) vrátí redukovaný OBDD reprezentující funkci f po dosazení 0 za x (zapíšeme $f[0/x]$),

`exists` odstraňuje závislost funkce na určité proměnné, výsledkem `exist`(x, f) je funkce $f[0/x] + f[1/x]$. Varianta `exist`($\hat{x}$, f) provede stejnou operaci pro celou množinu proměnných $\hat{x}$.

Reprezentace množiny stavů logickou funkcí

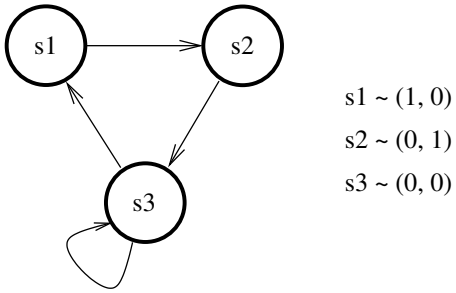
Nechť S je konečná množina (stavů). Budeme reprezentovat její různé podmnožiny pomocí OBDD.

OBDD reprezentuje logickou funkci, a proto potřebujeme elementy S kódovat logickými proměnnými. Kódování můžeme provést přiřazením jedinečného vektoru logických hodnot $(v_1, \ldots v_n)$, $v_i \in \{0, 1\}$ každému elementu. Pak množinu $T \subseteq S$ reprezentuje logická funkce $f_T : \{0, 1\}^n \rightarrow \{0, 1\}$. Funkce f_T přiřazuje vektorům odpovídajícím prvkům T hodnotu 1, ostatním 0.

Vektor stavových proměnných modelu z obr. 3.2 pojmenujme $\hat{x} = (x_1, x_2)$. Například stav $s1$ potom reprezentuje logická funkce $x_1 \cdot \overline{x}_2$, množinu stavů $\{s_2, s_3\}$ reprezentuje funkce $\overline{x}_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2 = \overline{x}_1$.

Reprezentace přechodové funkce logickou funkcí

Přechodovou funkci automatu můžeme reprezentovat obdobně jako množiny stavů. Logická funkce vyjadřující přechodovou relaci není definovaná pouze nad momentálními hodnotami stavových proměnných, ale i nad jejich budoucími hodnotami. Nechť x je stavová proměnná,



Obrázek 3.2. Jednoduchý model a reprezentace jeho stavů vektory binárních hodnot

pak x' reprezentuje její hodnotu v následujícím stavu. Například přechod z s_1 do s_2 uvažovaného modelu z obr. 3.2 reprezentuje funkce $x_1 \cdot \overline{x}_2 \cdot \overline{x}'_1 \cdot x'_2$. Přechodovou funkci pak určuje logický součet funkcí pro jednotlivé přechody.

Verifikace pomocí OBDD

Značkovací algoritmus ze str. 20 využívá pro EX a EU funkci, která vrací všechny stavy, ze kterých existuje přechod do stavu ze zadané množiny

$$\mathrm{pre}_\exists(X) = \{s \mid \exists s', (s \rightarrow s' \wedge s' \in X)\},$$

podobně algoritmus pro AF může využívat funkci, která vrací stavy, ze kterých vedou všechny přechody do zadané množiny

$$\mathrm{pre}_\forall(X) = \{s \mid \forall s', (s \rightarrow s' \Rightarrow s' \in X)\}.$$

Platí

$$\mathrm{pre}_\forall(X) = S - \mathrm{pre}_\exists(S - X),$$

kde S je množina všech stavů modelu. Pro verifikaci stačí tedy najít OBDD variantu funkce $\mathrm{pre}_\exists(X)$.

Jestliže B_X je redukovaný OBDD množiny stavů X a $B_\rightarrow$ OBDD přechodové funkce, zapíšeme funkci $\mathrm{pre}_\exists(X)$

1. přejmenuj proměnné v B_X na jejich varianty s čárkami, diagram pojmenuj $B_{X'}$,
2. vypočítej nový OBDD použitím funkce `exists`($\hat{x}'$,`apply`($\cdot$,$B_\rightarrow$, $B_{X'}$)), výsledný diagram reprezentuje všechny stavy, ze kterých vede přechod do některého ze stavů v množině reprezentované B_X.

S využitím tohoto postupu lze značkovací algoritmus z kap. 3.2.1 převést na operace s OBDD reprezentující množiny stavů.

Pro efektivní využití symbolické reprezentace množin stavů je vhodné nevytvářet popis množin z výčtu všech stavů a jejich následného kódování, ale přímo z popisu systému. Takovou tvorbu logické funkce reprezentované OBDD provádí verifikační nástroj SMV (kap. 3.3).

3.2.3 Bounded model checking

Bounded Model Checking (BMC, [5]) je symbolickou verifikační technikou, která pro reprezentaci logické funkce (a tedy množiny stavů – viz předchozí část) nevyužívá OBDD. Místo toho se logické funkce reprezentují jinými prostředky a jejich splnitelnost se posuzuje procedurami, které řeší úlohu určení splnitelnosti funkcí výrokové logiky (SAT).

BMC hledá příklad dané délky k, který by potvrdil pravdivost či nepravdivost formule. Nejprve hledá příklad délky $k = 1$, pokud ho nenajde, zvýší k o 1 a pokračuje s hledáním. Verifikace má zadanou horní mez délky příkladu. Pokud není nalezen příklad, který by potvrzoval danou formuli, pro k menší nebo rovnou zadané mezi, verifikace skončí bez toho, aby rozhodla o pravdivosti formule.

Konečná horní mez délky příkladu může způsobit nerozhodnutí o pravdivosti. BMC technika ale často dosáhne výsledku verifikace rychleji a s menšími paměťovými nároky než verifikace pomocí OBDD, navíc je v případě nepravdivosti vždy nalezen nejkratší možný protipříklad.

3.2.4 Verifikace časových automatů

Časový automat obsahuje narozdíl od klasických automatů obsahuje další prvek – hodiny, anglicky clock. Hodinyy jsou proměnnou, jejíž hodnota se pohybuje v rozsahu nezáporných reálných čísel a stoupá lineárně v čase. Pokud je v automatu více hodin, jejich hodnota stoupá stejnou rychlostí. Existuje více definic časového automatu, definice uvedená v [2], ze které vychází model v nástroji UPPAAL, je následující.

Definice 3.4 Nechť C je množina hodin. Nechť $B(C)$ je je množina konjunkcí nad jednoduchými podmínkami ve formě $x \bowtie c$ a $x - y \bowtie c$, kde $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$ a c je přirozené číslo. Časový automat nad C je šestice (L, l_0, E, g, r, I), kde L je množina míst, $l_0 \in L$ je počáteční místo, $E \in L \times L$ je množina hran, $g : E \to B(C)$ přiřazuje hranám střežení, $r : E \to 2^C$ přiřazuje hranám hodiny, které se mají resetovat (nulovat), a $I : L \to B(C)$ přiřazuje hranám invarianty. □

Sémantika časového automatu vede díky reálným hodnotám hodin na nespočetný stavový prostor. Existuje však abstrakce, která rozděluje prostor $\mathbb{R}^C$ na konvexní mnohostěny nazývané zóny. Tato abstrakce vede na symbolickou sémantiku časového automatu.

Definice 3.5 Nechť $Z_0 = \bigwedge_{x \in C} x \geq 0$ je počáteční zóna. Symbolická sémantika časového automatu (L, l_0, E, g, r, I) nad množinou hodin C je definována jako systém $(S, s_0, \Rightarrow)$ nazývaný simulační graf, kde $S = L \times B(C)$ je množina symbolických stavů, $s_0 = (l_0, Z_0 \wedge I(l_0))$ je počáteční stav a relace $\Rightarrow = \{(s, u) \in S \times S \mid \exists e, t : s \overset{e}{\Rightarrow} t \overset{\delta}{\Rightarrow}\Rightarrow u\}$ je přechodová relace a

- $(l, Z) \overset{\delta}{\Rightarrow} (l, norm(M, (Z \wedge I(l))^{\uparrow} \wedge I(l)))$
- $(l, Z) \overset{e}{\Rightarrow} (l', r_e(g(e) \wedge Z \wedge I(l)) \wedge I(l'))$ pro $e = (l, l') \in E$,

kde $Z^{\uparrow} = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ (operace posunu do budoucnosti) a $r_e(Z)$ vrací mnohostěn, který má oproti Z vynulované hodiny, které se nulují s hranou e. Funkce $norm$:

```
W = {(l_0, Z_0 ∧ I(l_0))}
P = ∅
while W ≠ ∅ do
  (l, Z) = W.popstate()
  if testProperty(l, Z) then return true
  if ∀(l, Z) ∈ P : Z ⊈ Y then
    P = P ∪ {(l, Z)}
    ∀(l', Z') : (l, Z) ⇒ (l', Z') do
      if ∀(l', Y') ∈ W : Z' ⊈ Y' then
        W = W ∪ {(l', Z')}
      endif
    done
  endif
done
return false
```

Obrázek 3.3. Algoritmus splnitelnosti v časovém automatu

$\mathbb{N} \times B(C) \rightarrow B(C)$ normalizuje omezení hodin s ohledem na maximální konstantu M časového automatu. □

Relace $\overset{\delta}{\Rightarrow}$ obsahuje posuny v čase a relace $\overset{e}{\Rightarrow}$ přechody po hranách automatu. S využitím symbolické sémantiky lze přímo sestrojit algoritmus z obr. 3.3 [2] testující dosažitelnost symbolického stavu, který splňuje požadovanou vlastnost.

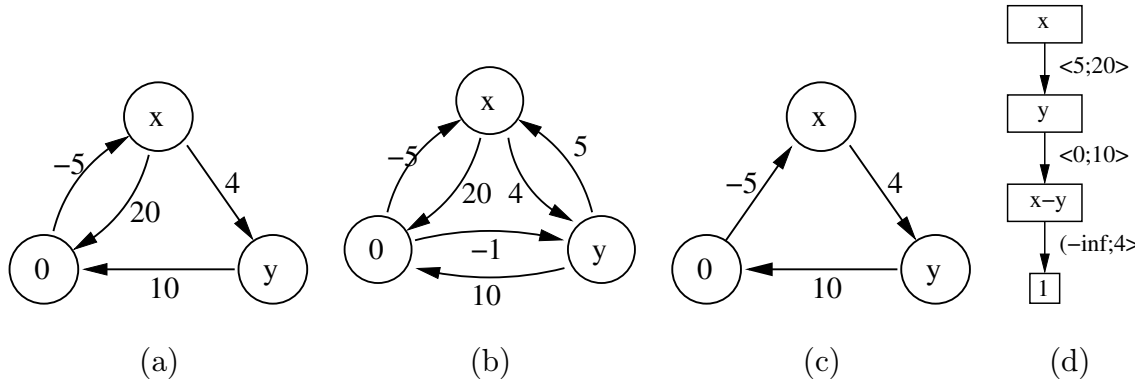
V pseudokódu pro algoritmus je P seznam prozkoumaných symbolických stavů, W je seznam stavů čekajících na prozkoumání. Funkce $testProperty(l, Z)$ vyhodnocuje, zda daný stav splňuje testovanou vlastnost (formuli) – například, testujeme-li dosažitelnost místa l_i, funkce vrací „true“ pro $l = l_i$, jinak vrací „false“.

Pro efektivní implementaci algoritmu je důležité použít vhodné datové struktury a algoritmy pro reprezentaci a manipulaci s časovými zónami. Patří mezi ně

Diference bounded matrices (DBM) – matice, jejichž prvky m_{ij} odpovídají časovým omezením $x_i - x_j \leq m_{ij}$. Členy x_i a x_j jsou hodinami automatu nebo hodnotou 0. Některé prvky matice odpovídají přímo zadaným omezením na hodiny, další lze získat nalezením nejkratších cest mezi všemi uzly váženého grafu reprezentovaného zadanými omezeními. (Uzly grafu odpovídají hodinám, případně hodnotě 0, hrany mají váhy dané hodnotou m_{ij}, pokud je definovaná, jinak hrana z x_i do x_j neexistuje.) Výhodou DBM je snadná realizace testu, zda je jedna zóna podmnožinou jiné.

Minimal constraint representation je méně paměťově náročnou alternativou DBM. Jedná se o graf získaný odstraněním přebytečných hran grafu reprezentovaného DBM (Přebytečná hrana je taková, k níž existuje cesta se stejným počátečním a koncovýmn uzlem a nižší cenou). Tato úprava často dosahuje značných úspor paměti a někdy i času.

Clock difference diagrams (CDD) se svou podstatou podobají OBDD, lze na ně aplikovat i podobné algoritmy. Uzly grafu odpovídají hodinám nebo jejich rozdílům, jejich pořadí



Obrázek 3.4. Množina omezení $\{x \geq 5; x \leq 20; y \leq 10; x - y \leq 4\}$ vyjádřená jako graf (a), graf nejkratších cest (b) (po odstranění redundantních hran (c)) a CDD (d)

je dané seřazením stejně jako u OBDD. Z každého nekoncového uzlu nevedou narozdíl od OBDD dvě hrany, ale jedna nebo více hran označených celočíselným intervalem, který určuje, pro jaké hodnoty hrana plati. Oblast reprezentovaná CDD se určí jako sjednocení zón daných konjunkcemi omezení podél cest do listu „1“ – cesty do „0“ se tudíž ani nemusejí v diagramu uvádět. Výhodou CDD je možnost realizace sjednocení více zón (to nemusí být konvexní) jedním diagramem a nižší paměťová náročnost oproti DBM.

3.3 SMV

Jedním z dostupných verifikačních nástrojů je *Symblolic Model Verifier* (SMV, [10]). Umožňuje ověřování požadavků systémů s konečným počtem stavů metodou symbolic model checking (kap. 3.2.2). Požadavky mají formu formulí CTL. 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.

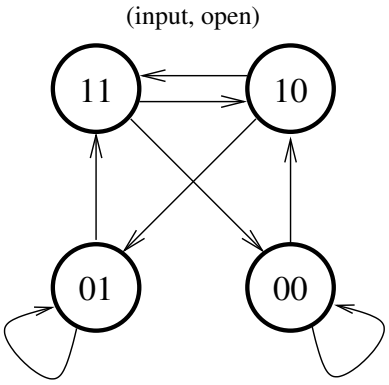
3.3.1 Modelovací jazyk

Jazyk SMV popisuje systém soustavou rovnic nad proměnnými, které určují stav systému. Základním prvkem jazyka je rovnice `next(`*`proměnná`*`) := `*`výraz`*, která přiřazuje hodnotu nebo množinu možných hodnot proměnné v následujícím stavu. Obr. 3.5 vlevo ukazuje příklad kódu v SMV, vpravo je potom zobrazen model jako grafická reprezentace nedeterministického automatu.

Mezi další možnosti popisující přechodovou funkci modelu patří přiřazení *`proměnná`* `:=` *`výraz`* udávající hodnotu proměnné ve stavu, ve kterém se systém právě nachází, a rovnice *`výraz`* `=` *`výraz`* v sekci TRANS, která omezuje přechodovou funkci na přechody mezi stavy vyhovující rovnici.

Datovými typy vstupního jazyka jsou binární proměnná, skalár (číslo ze zadaného konečného intervalu celých čísel nebo výčtový typ – `state` z obr. 3.5) a pole pevné délky. Definice proměnných se uvádějí v sekci VAR.

```
MODULE main
VAR
 input : boolean;
 open : boolean;
ASSIGN
 next(open) := !input&open | input&!open;
```



Obrázek 3.5. Ukázka modelu v SMV

Kromě množiny možných příštích stavů můžeme určit i počáteční hodnoty proměnných a tedy i množinu možných počátečních stavů. Inicializaci provedou přiřazení `init(`*`proměnná`*`) := `*`výraz` v sekci ASSIGN, případně rovnice *`výraz = výraz` v sekci INIT, která je obdobou sekce TRANS, ovšem omezuje počáteční stavy systému.

Výrazem může být logický nebo aritmetický výraz, ale také kostrukce *case* (obr. 3.6) nebo nedeterministický výraz. 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 výrazy popisují systémy, jejichž chování má alespoň zčásti náhodný charakter. Příkladem může být systém s neznámou dobou reakce z obr. 3.6. Nedeterministický výraz se zapisuje jako množina možných výrazů vypsaných mezi složené závorky nebo spojených pomocí slova *union*. Výraz je určen náhodně z této množiny. Lze ho využít jak v přiřazení next, tak i init. Příklady zápisu inicializace a nedeterministických výrazů obsahuje

```
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;
 ...

DEFINE
 noLimit := !botLimit & !topLimit;
```

Obrázek 3.6. Inicializace, nedeterministické výrazy, definice symbolu

```
MODULE cell(s, r)
VAR
 value : boolean;
ASSIGN
 next(value) := case
  r&!s : 0;
  s&!r : 1;
  1 : value;
 esac;

MODULE main
VAR
 a : boolean;
 b : boolean;
 bit1 : cell(a, b);
 bit2 : cell(b, a);
ASSIGN
 init(a):=1;
 init(b):=0;
```

Obrázek 3.7. Model rozložený na více modulů

kód na obr. 3.6. Jestliže některá proměnná nemá přiřazení, které by určovalo její hodnotu (nebo množinu možných hodnot), ani ji neomezují rovnice v TRANS, může nabýt jakékoliv hodnoty z oboru svých hodnot (`input` v obr. 3.5).

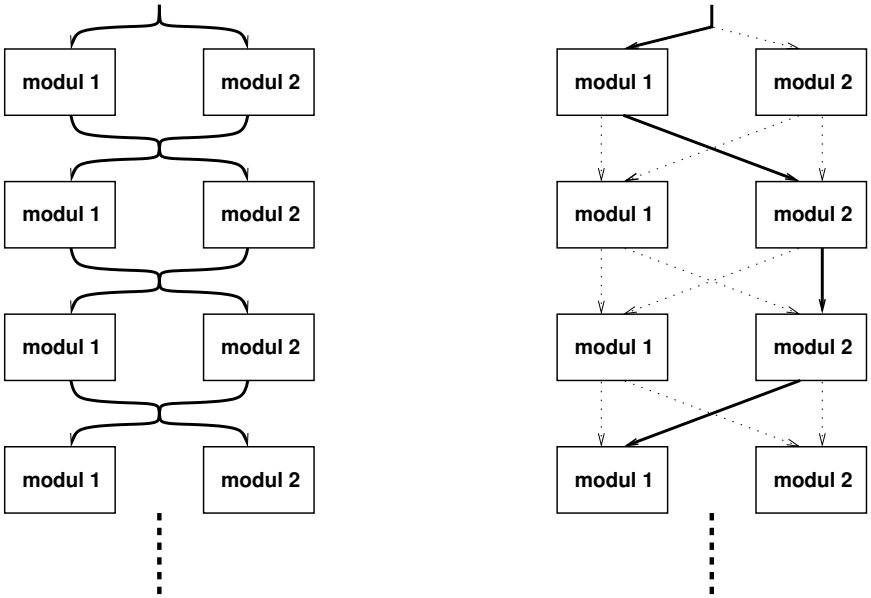
Sekce DEFINE definuje symboly, např. (`noLimit` v obr. 3.6). Symboly narozdíl od proměnných nezvětšují stavový prostor, jejich nevýhodou je nemožnost nedeterministického přiřazení. Místo zmíněného symbolu `noLimit` bychom mohli definovat proměnnou:

```
VAR
 noLimit : boolean;
ASSIGN
 noLimit := !botLimit & !topLimit;
```

Model systému se může skládat z jednoho či 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 lze definovat instance dalších modulů.

Definice modulu začíná klíčovým slovem MODULE následovaným názvem modulu. Za názvem modulu může být v závorkách uveden seznam parametrů. Pro využití modulu je třeba vytvořit jeho instanci definicí *jméno instance* := *název modulu*(*seznam parametrů*) v sekci VAR jiného modulu. Vzniká tak stromová hierarchická struktura s kořenem v modulu main. Pro přístup k proměnné definované uvnitř podřízeného modulu se zapíše *instance* . *proměnná* (na obr. 3.7 poslední dva řádky sekce VAR modulu main).

Moduly `bit1` a `bit2` definované v příkladu běží v souběžném režimu, vyhodnocují se oba v každém stavovém přechodu SMV programu. Množinu stavů dává soustava rovnic obou modulů. Druhou variantou definice modulů je použití klíčového slova *process* v definici instance, např.



Obrázek 3.8. Souběžný a prokládaný režim

```
bit1 : process cell(a,b);
bit2 : process cell(b,a);
```

Moduly pak běží v prokládaném režimu, tj. v kažném okamžiku se vyhodnocují rovnice poze jednoho (kteréhokoliv) z nich. Toto je jeden ze způsobů, jak lze modelovat asynchronní souběh systémů popsaných jednotlivými moduly. Diagramy na obr. 3.8 nastiňují běh modelu o dvou modulech v souběžném (vlevo) a prokládaném (vpravo) režimu. Šipky v diagramu prokládaného režimu představují možné posloupnosti běhu, jednu takovou posloupnost zvýrazňuje silná čára.

Dalším prostředkem k popisu asynchronních systémů jsou nedeterministická přiřazení. Například zápis `(state=mup)&!botLimit : {0,1}` v příkladu 3.6 znamená, že jsou-li splněny podmínky pro přechod `topLimit` z 0 do 1, hodnota se změní buď ihned nebo zatím ne, změna pak může nastat až někdy v budoucnosti, se zpožděním. Modeluje se tak situace, kdy odezva systému na změny řídicích veličin nemusí být okamžitá.

3.3.2 Verifikace kritérií

Pro verifikaci modelu uvedeme v jeho popisu jednu nebo více sekcí SPEC obsahujích ověřované formule CTL logiky. Výsledný soubor potom slouží jako vstup verifikačního nástroje, který rozhodne, zda model požadavky splňuje.

Pro ověření některých vlastností modelu z obr. 3.7 přidáme na konec souboru s popisem modelu řádky

```
SPEC
 AX AG (bit1.value = !bit2.value)
SPEC
 AG (a&!b -> AX A[bit1.value U b&!a])
```

Splnění první formule znamená, že od druhého stavu si hodnoty bitů nikdy nebudou rovny. Druhá vyjadřuje požadavek, aby po hodnotách vstupů určených pro nastavení `bit1.value` na

```
-- specification AX AG bit1.value = (!bit2.value) is true
-- specification AG ((a & !b) -> AX A [ bit1.value U (b & !a) ] ) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a = 1
  b = 0
  bit1.value = 0
  bit2.value = 0
-- Loop starts here
-> State: 1.2 <-
  bit1.value = 1
-> State: 1.3 <-
```

Obrázek 3.9. Výsledek verifikace modelu

hodnotu 1 byla od následujícího stavu skutečně tato hodnota nastavena, a to na dobu nejméně do okamžiku, kdy vstupy vyjadřují požadavek na změnu hodnoty na 0.

Výsledek verifikace může být dvojí. Buď je daná formule ve verifikovaném modelu pravdivá pro všechny počáteční stavy, potom tuto skutečnost SMV oznámí, nebo pravdivá není, potom se navíc pokusí vypsat posloupnost stavů, která vede k nesplnění formule.

Obrázek 3.9 ukazuje výsledek verifikace výše uvedených formulí na modelu z obr 3.7. První formule je pravdivá. Druhá pravdivá není – SMV vypsal sekvenci stavů, která ji nesplňuje. V získané sekvenci `bit1.value` ve 2. stavu nastaven na 1, ale nikdy v budoucnu není na vstupech kombinace (b=1, a=0) potřebná pro nastavení do 0. Formule `A [bit1.value U b&!a]` pak nemůže být nikdy splněna (viz popis operátoru U v kap. 3.1.1).

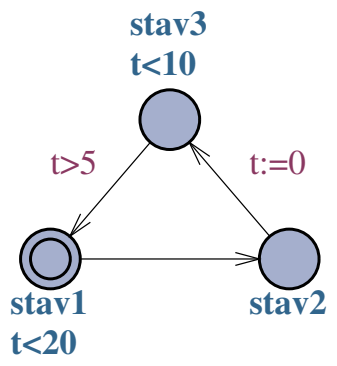
Posledním zde zmíněným prvkem jazyka SMV je sekce FAIRNESS. Formule CTL uvedená v této sekci je během každé nekonečně dlouhé posloupnosti stavů splněná v nekonečně mnoha okamžicích, to znamená, že pro každý okamžik existuje nějaký budoucí stav, kdy je splněná. Při verifikaci se pak ignorují trajektorie, ve kterých je splněna pouze v konečném počtu okamžiků (existuje okamžik, po kterém není splněna nikdy). Jako formuli lze využít i speciální proměnnou *running*, která je pravdivá v okamžicích, kdy se vykonává tělo příslušného procesu.

Přidáním „`FAIRNESS b&!a`“ do modulu main z 3.7 zabráníme procházení cest, ve kterých v nějakém okamžiku neexistuje budoucí stav, ve kterém platí `b&!a`. Vyloučí se tak například i sekvence získaná jako protipříklad v předchozí verifikaci. Pak už je formule `AG(a&!b -> AX A[bit1.value U b&!a])` splněna:

```
-- specification AG ((a & !b) -> AX A [ bit1.value U (b & !a) ] ) is true
```

3.3.3 Přehled implementací SMV

Systém SMV napsal K. McMillan [10]. Většinu prvků modelovacího jazyka, se kterým pracuje jeho první implementace, popisuje tato kapitola. SMV umožňuje nastavit některé parametry verifikace pomocí volby argumentů na příkazové řádce. Zdrojové kódy systému a binární



Obrázek 3.10. Časový automat v UPPAALu

soubory pro některé platformy jsou volně přístupné ke stažení na stránkách Carnegie Mellon University [17].

Skupina formálních metod na „Centre for scientific and technological research“ univerzity v Trentu vyvinula ve spolupráci s Carnegie Mellon University reimplementaci nazvanou NuSMV. Tato imlementace rozšiřuje SMV o možnost verifikace formulí *Linear Time Logic* (LTL) redukcí na verifikaci CTL nebo technikou Bounded Model Checking. NuSMV poskyje tzv. *interactive chell*, textové prostředí pro interaktivní načítání modelu, spouštění verifikace jednotlivých formulí a simulaci modelu. Na stránkách [18] je přístup k manuálům a k binárním i zdrojovým kódůn NuSMV.

Cadence SMV dostupné na [19] je verifikačním nástrojem, který se podobá původnímu SMV, ale rozšiřuje ho o mnoho dalších možností.

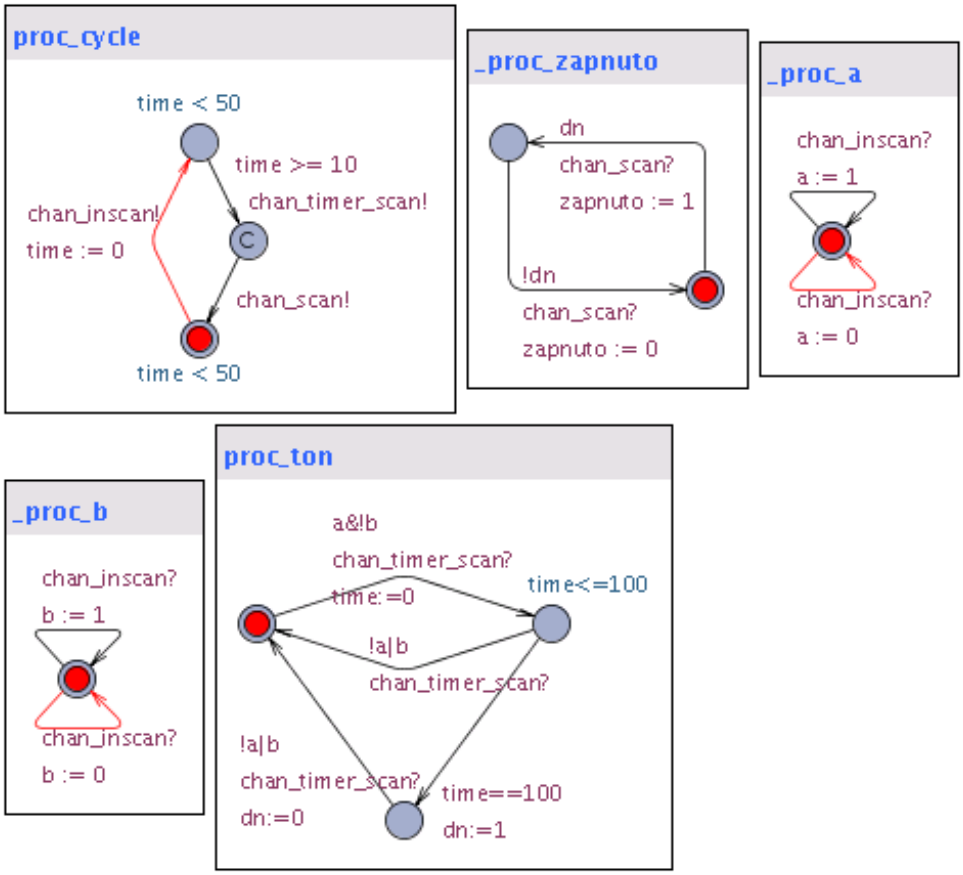
3.4 Uppaal

UPPAAL [8, 3] je nástroj pro modelování, simulaci a verifikaci systémů reálného času vyvinutý univerzitami v Uppsale a Aalborgu. Aplikace je dostupná na [20], kde lze nalézt i manuály a odkazy na články publikované v souvislosti s UPPAAL. Systém nabízí konzolovou aplikaci pro verifikaci modelů (`verifyta`) i grafické uživatelské prostředí, ve kterém lze grafickou formou modelovat systém, provádět jeho simulaci i spouštět verifikaci požadavků.

3.4.1 Modelování

Model systému v UPPAALu se skládá z jednoho či více procesů. Popis procesu vychází z časového automatu (def. 3.4), ale umožňuje ho rozšířit o další prvky, popis celého modelu je tedy rozšířením sítě časových automatů. Obrázek 3.10 ukazuje proces vytvořený jako model časového automatu bez využití dalších možností popisu.

Proměnné – Model v UPPAALu může pracovat i s jinými proměnnými než hodinami. Stejně jako může hrana resetovat hodiny, může přiřadit hodnotu i ostatním proměnným (konstantní nebo danou ohodnocením výrazu). Výrazy s proměnnými se mohou využít jako podmínky uvolnění hran (střežení) nebo invarianty míst. Rozlišujeme proměnné



Obrázek 3.11. Pohled na model složený z několika procesů

typu logická proměnná (`bool a;`), celé číslo (`int b1;`) v rozsahu $\langle -32768, 32767 \rangle$ nebo s menším rozsahem uvedeným v deklaraci(`int[0,10] b2;`) a pole (`bool c[4]; int d[2];`).

Reset hodin v UPPAALu nemusí znamenat jen vynulování jejich hodnoty, ale i přiřazení jiné konstantní hodnoty nebo dokonce hodnoty dané ohodnocením výrazu typu int. Střeření u hran také může obsahovat porovnání hodnoty hodin s celočíselnou proměnnou nebo výrazem.

Konstanty jsou prvky modelu, které mají stejně jako proměnné svoje jméno, ale jejich hodnota je konstantní. Definice konstanty může být např. `const e 1;`

Inicializace proměnných je určení jejich počáteční hodnoty. Inicializovat lze všechny proměnné kromě hodin (jejich počáteční hodnota je vždy 0). Počáteční hodnota se určí v deklaraci proměnné, např. `bool a:=true;` nebo `int[0,10] b:=3;`

Šablona (template) v UPPAALu obsahuje model procesu. Podle jedné šablony lze vytvořit více instancí procesu.

Binární synchronizační kanály se deklarují slovem `chan`, například '`chan k;`'. Hrana s návěštím '`k!`' je pak synchronizována s hranou '`k?`' v jiném procesu. K přechodu může dojít pouze v případě, že obě hrany splňují podmínky pro přechod, dojde k němu potom na obou hranách zároveň.

Broadcast kanály se deklarují jako '`broadcast chan k;`' Jedna hrana s návěštím '`k!`' se synchronizuje se všemi hranami s návěštím '`k?`', u kterých může dojít k přechodu.

Není-li žádná hrana s 'k?' volná, může přesto, narozdíl od binárního kanálu, dojít k přechodu u hrany s 'k!'.

Urgentní kanály způsobují, že hrany se jimi synchronizonizované přechody provedou ihned, jakmile jsou uvolněny podmínky přechodu, nedochází k žádnému zpoždění. Hrany synchronizované těmito kanály nemohou mít v podmínkách přechodu časová omezení (omezení na hodiny). Urgentní kanál se definuje s klíčovým slovem `urgent` před deklarací kanálu.

Urgentní místa jsou sémanticky ekvivalentní přidání hodin (např. `x`), které jsou nulovány na věech hranách přicházejících do místa, které má invariant `x<=0`. Když je systém v urgentním místě, nemůže uběhnout žádný čas.

Místa „committed“ mají nulové zpoždění jako urgentní místa. Navíc, pokud proces je v tomto místě, musí ho opustit dříve, než jiný proces změní místo neoznačené jako committed. Je-li v místě committed více procesů zároveň, opouští je v náhodném pořadí.

Obr. 3.11 ukazuje pohled na model zobrazený v simulátoru. Model se skládá z několika procesů synchronizovaných kanály typu broadcast. V procesu `proc_cycle` je místo označené jako committed (zobrazuje se jako kruh s písmenem C uprostřed). Kanál `chan_scan` se díky použití místa committed emituje ihned po kanálu `chan_timer_scan`, s nulovým zpožděním. Procesy `_proc_a` a `_proc_b` byly vytvořeny ze společné šablony, ale s různými parametry (`a` a `b`).

3.4.2 Verifikace

Verifikační nástroj UPPAAL umožňuje ověřovat několik typů formulí vycházejících z TCTL uvedené v tab. 3.1. V levém sloupci se nacházejí předpisy pro zápis formulí v UPPAALu, `výraz` je logický výraz definovaný nad proměnnými modelu včetně hodin. Pravý sloupec obsahuje formule zapsané způsobem dosud používaným v této kapitole. Poslední formule má význam „vede na“, je pravdivá tehdy, když po každém stavu, kdy platí první výraz, vždy někdy v budoucnu platí druhý výraz.

Verifikované formule mohou obsahovat speciální binární proměnné, které určují, zda je proces v určitém místě. Například `P.l` je pravdivá, jestliže se proces `P` nachází v místě `l`.

Mezi omezení formulí, které je možné verifikovat v UPPAALu, oproti TCTL patří skutečnost, že neobsaují operátor U a především v nich není možné do sebe libovolně vnořovat dvojice operátorů AG, EF, EG, AF. Oproti TCTL ale výraz ve formuli může obsahovat několik binárních operátorů porovnávajících hodiny a proměnné, ne pouze jedno srovnání mezi hodinami a konstantou (viz kap. 3.1.4).

`'A[]' výraz`	AG *výraz*
`'E<>' výraz`	EF *výraz*
`'E[]' výraz`	EG *výraz*
`'A<>' výraz`	AF *výraz*
`výraz1 '-->' výraz2`	AG (*výraz1* → AF *výraz2*)

Tabulka 3.1. Formule verifikovatelné v UPPAALu

Před spuštěním verifikace lze nastavit některé její parametry. Patří mezi ně nastavení způsobu reprezentace stavového prostoru nebo to, zda se má při nesplnění formule generovat protipříklad a jestli má jít o nejkratší možný protipříklad (ať už z časového hlediska nebo počtem stavů na cestě).

3.4.3 Souborové formáty

UPPAAL využívá k načítání modelů ze souborů knihovnu libutap (Uppaal Timed Automata Parser library, [21]), která dokáže zpracovat tři následující formáty.

TA formát je nejstarší. Je to čistý text vhodný pro manuální popis systémů. Nelze v něm uložit šablony a informace o grafické podobě modelu (souřadnice míst, návěští apod.).

XTA rozšiřuje TA formát o šablony a tedy o možnost vytvářet podle nich procesy s různými parametry. Informace o grafické podobě modelu nejsou součástí xta souboru, ale UPPAAL je může načíst ze souboru s příponou ugi, pokud byl vytvořen.

XML formát je XML (Extended Markup Language) verze XTA formátu. Elementům modelu, jako např. šablonám, místům, hranám, návěštím, odpovídají jednotlivé XML tagy. Grafické informace jsou přímou součástí souboru s popisem systému. Definice typu dokumentu (DTD) pro formát XML uvádí příloha A. DTD definice neříká, jaká je přesně syntaxe návěští, deklarací a dalších elementů, pouze udává strukturu modelu.

Uživatelské prostředí používá jako svůj přirozený formát XML. Soubory formátů TA a XTA v něm můžeme otevřít, ale od verze UPPAAL 3.4 je již nelze ukládat.

Kapitola 4

Převod automatu do SMV

Tato kapitola popisuje navržený postup tvorby SMV modelu, který modeluje automat popsaný logickými rovnicemi.

Soustava přiřazení `next` v SMV určuje hodnoty stavových proměnných v následujícím stavu, v následujícím okamžiku. Stejný význam má i množina přiřazení $(\widehat{\mathrm{co}_\mathrm{p}}(\widehat{C}) \cap \widehat{\mathrm{dom}_\mathrm{p}}(\widehat{C}))$ popisující přechodovou funkci automatu z def. 2.12, jehož jeden přechod odpovídá jednomu scan cyklu PLC. Zápis přiřazení dané trans-množiny jako přiřazení next v SMV vytváří model popisující stav automatu. Tato myšlenka je základem dále popsaných postupů převodu.

4.1 Převod Mealyho automatu

Výstupní funkci automatu dle def. 2.12 určují t-přiřazení z $\widehat{C} \widehat{\cap} \Omega$, kde $\widehat{C}$ je trans-množina získaná algoritmem APLCTRANS a Ω je množina výstupních proměnných. Přiřazení udává hodnotu výstupu v daném okamžiku ohodnocením výrazu momentálními hodnotami stavových a vstupních proměnných. V SMV takovému přiřazení neodpovídají přiřazení `next`, ale přiřazení okamžitým hodnotám. Pro výstup automatu není třeba v SMV zavádět stavovou proměnnou, protože neurčuje stav automatu, výstup je pak definován jako symbol `DEFINE` *výstup* `:=` *výraz*, kde *výstup* představuje jméno výstupní proměnné a *výraz* zápis výrazu t-přiřazení v syntaxi SMV.

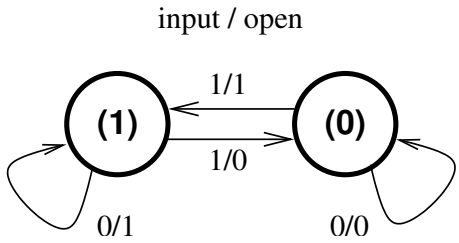
Na obr. 4.1 vlevo nahoře je trans-množina získaná převodem APLC programu z obr. 2.3, pod ní SMV model automatu generovaného APLC programem za podmínky, že množina vstupů $\Sigma = \{input\}$, množina vnitřních proměnných $V = \emptyset$ a množina výstupů $\Omega = \{open\}$. Hodnota proměnné `m_open` udává stav automatu (stav **(1)** $\sim$ `m_open`=1, stav **(0)** $\sim$ `m_open`=0).

Je-li některá výstupní proměnná obsažena v $\mathrm{co}_\mathrm{p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C})$, příslušné t-přiřazení určuje jak přechodovou, tak i výstupní funkci. Výstup automatu a stavová proměnná získaná ze stejného t-přiřazení si nejsou rovny, protože dané t-přiřazení určuje okamžitou hodnotu výstupu, ale až příští hodnotu stavové proměnné. V kódu SMV potom figuruje přiřazení next pro stavovou proměnnou i definice výstupu jako symbolu se stejnou pravou stranou. Jméno dané výstupní proměnné PLC nelze v tomto případě využít v SMV zároveň pro definici stavové proměnné i výstupu. Model na obr. 4.1 využívá pro pojmenování výstupu jméno výstupní proměnné APLC programu (`open`), stavová proměnná je pojmenovaná `m_open`.

Pravá strana přiřazení pro `m_open` na obr. 4.1 je shodná s výrazem definovaným pod

$\{open\ [\![(\neg input \wedge open) \vee (input \wedge \neg open)]\!]\}$

```
MODULE automat(input)
VAR
 m_open : boolean;
ASSIGN
 init(m_open) := 0;
 next(m_open) := !input&m_open|input&!m_open;
DEFINE
 open := !input&m_open | input&!m_open;
```



Obrázek 4.1. Trans-množina získaná APLCTRANSem, model automatu v SMV, graf přechodů automatu

symbolem `open`, můžeme ji tedy tímto symbolem nahradit, přiřazení pro `m_open` po nahrazení je `next(m_open) := open;`

Nyní popíšu postup převodu automatu generovaného APLC programem do SMV jako postup v několika bodech. Výsledkem je modul jazyka SMV modelující automat generovaný daným programem. Σ, Ω a V jsou vstupní, výstupní a vnitřní proměnné programu generujícího automat podle def. 2.12, $\widehat{C}$ je trans-množina získaná algoritmem APLCTRANS. Množina stavových proměnných pak je

$$Z = (\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C})).$$

Předpokládám, že jména proměnných splňují požadavky na identifikátory v SMV, tedy začínají písmenem nebo '_', nejsou shodné s žádným klíčovým slovem, atd. Zápisem $jm(x)$ v kódu SMV myslím jméno proměnné x, $x(0)$ je počáteční hodnota x. Počáteční hodnoty proměnných z množiny Z určují počáteční stav automatu.

Postup platí za předpokladu, že pro žádnou proměnnou $x \in ((V \cup \Omega) \cap \mathrm{dom}(\widehat{C} \widehat{\cap} (Z \cup \Omega)))$ není $\hat{x}\ [\![1]\!] \in \widehat{C}$ a zároveň $x(0) = 0$, ani $\hat{x}\ [\![0]\!] \in \widehat{C}$ a zároveň $x(0) = 1$. Všechny proměnné s konstantním přiřazením, které se vyskytují v přiřazeních pro stavové a vnitřní proměnné, tedy zachovávají počáteční hodnotu i ve všech následujících stavech a lze je nahradit konstantami. Rozšířením postupu o proměnné, u kterých tato podmínka neplatí, se zabývá kap. 4.2.

1. Hlavička modulu je „`MODULE` $jmenomodulu$`(`$jmenavstupu$`)`“, kde $jmenomodulu$ znamená zvolené jméno modulu a $jmenavstupu$ je výpisem jmen proměnných z množiny $\mathrm{dom}(\widehat{C}) \cap \Sigma$ oddělených čárkami.
2. Vytvoř pojmenování proměnných z množiny Z, $p : Z \to I$, kde I je množina všech řetězců použitelných jako identifikátory v SMV a různých od jmen proměnných v Σ a Ω. Pro $x_i, x_j \in Z, i \neq j$ platí $p(x_i) \neq p(x_j)$.
3. V sekci VAR definuj proměnné ze Z jako proměnné typu boolean se jmény danými přejmenováním p, tedy pro všechny $x \in Z$ zapiš „$p(x)$ `: boolean;`“
4. Do sekce ASSIGN uveď pro každou $x_i \in Z \cap V$ přiřazení „`next(`$p(x_i)$`) :=` $vyraz_i$`;`“, kde $vyraz_i$ je zápis výrazu z t-přiřazení $\hat{x}_i \in \widehat{C}$ v syntaxi SMV a s přejmenováním p stavových proměnných. Pro všechny $x \in Z \cap \Omega$ zapiš „`next(`$p(x)$`) :=` $jm(x)$`;`“
5. Pro všechny $x \in Z$ uveď přiřazení „`init(`$p(x)$`) :=` $x(0)$`;`“.

6. V dekci DEFINE uveď „$jm(x_i)$ `:=` $vyraz_i$`;`“ pro všechny $x_i \in \Omega$, $vyraz_i$ je zápis výrazu z t-přiřazení $\hat{x}_i \in \widehat{C}$ v syntaxi SMV a s přejmenováním p stavových proměnných.
7. Proměnné z $\{x : x \in ((V \cup \Omega) \cap \mathrm{dom}(\widehat{C} \widehat{\cap} (Z \cup \Omega))), x \notin Z\}$ jsou vnitřními a výstupními proměnnými APLC, které se vyskytují ve výrazech pro hodnoty stavových a výstupních proměnných, ale nebyly dosud definovány. Jejich t-přiřazení jsou ve tvaru $\hat{x}\,[\![x]\!]$, $\hat{x}\,[\![0]\!]$, nebo $\hat{x}\,[\![1]\!]$. Definuj je v sekci DEFINE jako symboly „$jm(x)$ `:=` $x(0)$`;`“.

V bodu 7 se definují konstanty pomocí symbolů. Doplňkem je dosazení těchto konstant přímo do výrazů pro hodnoty výstupů a příští hodnoty stavových proměnných. Tímto řešením a dalšími úpravami uvedeného postupu se zabývá kap. 4.2.

Automat může být určen pro verifikaci formulí obsahujících nejen vstupní a výstupní proměnné modelovaného APLC programu, ale i pro formule obsahující vnitřní proměnné. Verifikovanou vnitřní proměnnou můžeme definovat jako symbol stejně jako jsme dosud definovali výstupy. Pokud je taková vnitřní proměnná zároveň stavovou proměnnou, musíme ji definovat dvakrát – jednou jako symbol a jednou jako proměnnou, ta musí v tomto případě dostat nové jméno. Při vytváření modelu tedy můžeme některé vnitřní proměnné modelovat stejně jako výstupy, toho dosáhneme například přesunem proměnných z množiny V do množiny Ω před spuštěním algoritmu.

4.1.1 Příklad

Nyní demontruji převod automatu na příkladu. Mějme trans-množinu získanou překladem APLC programu

$$\widehat{C} = \{\widehat{m1}\,[\![i1 \vee i2 \wedge m1]\!]\,, \widehat{m2}\,[\![m2]\!]\,, \widehat{o1}\,[\![m1 \wedge \neg m2]\!]\,, \widehat{o2}\,[\![i3 \wedge o1]\!]\}.$$

Počáteční hodnota všech proměnných z $V \cup \Omega$ je 0. Množina vstupních proměnných programu je $\Sigma = \{i1, i2, i3\}$ vnitřní proměnné jsou $V = \{m1, m2\}$ a výstupní $\Omega = \{o1, o2\}$. Množinu stavových proměnných Mealyho automatu získáme podle def. 2.12

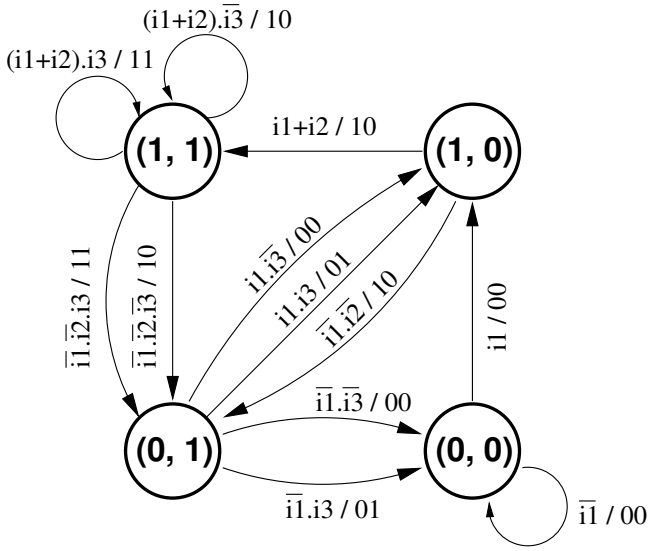
$$\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C}) = \{m1, o1, o2\} \cap \{i1, i2, i3, m1, m2, o1\} = \{m1, o1\}.$$

Následuje postup ze str. 36 aplikovaný na dané množiny $\widehat{C}, \Sigma, V, \Omega$.

1. Po volbě jména modulu „PLC“ je hlavička „`MODULE PLC(i1,i2,i3)`“.
2. Množina stavových proměnných $Z = \{m1, o1\}$. Pojmenování p pro proměnné ze Z zvolme například $p(m1) =$ `m1`, $p(o1) =$ `m_o1`.
3. V sekci VAR jsou definice `m1` a `m_o1`.
4. Do sekce ASSIGN zapíšeme „`next(m1) := i1 | i2 & m1; next(m_o1) := o1;`“.
5. Počáteční hodnoty stavových proměnných „`init(m1) := 0; init(m_o1) := 0;`“.
6. Definice symbolů výstupů je „`DEFINE o1 := m1 & !m2; o2 := i3 & m_o1;`“
7. Proměnná $m2$ se vyskytuje ve výrazu pro $o1$ (platí tedy $m2 \in \mathrm{dom}(\widehat{C} \widehat{\cap} (Z \cup \Omega))$). Hodnota $m2$ je 0, protože přiřazení $\widehat{m2}$ je kanonické a počáteční hodnota $m2$ je 0. Do sekce DEFINE umístíme definici konstanty „`m2 := 0;`“.

Automat a jeho SMV model ukazuje obr. 4.2. Stavy automatu jsou pojmenovány podle hodnot stavových proměnných, například stav **(1, 0)** určují hodnoty stavových proměnných `m1=1` a `m_o1=0`. Popisky u hran jsou ve formátu „podmínka přechodu / hodnoty výstupu“.

```
MODULE PLC(i1, i2, i3)
VAR
 m1 : boolean;
 m_o1 : boolean;
ASSIGN
 init(m1) := 0;
 init(m_o1) := 0;
 next(m1) := i1|m1&i2;
 next(m_o1) := o1;
DEFINE
 o1 := m1&!m2;
 o2 := m_o1&i3;
 m2 := 0;
```



Obrázek 4.2. Mealyho automat a jeho SMV model

4.2 Úpravy postupu

4.2.1 Konstantní přiřazení různé od počáteční hodnoty

Postup převodu na str. 36 platí pouze za předpokladu, že všechna konstantní přiřazení proměnným, na kterých přímo závisí stavové a vnitřní proměnné, přiřazují počáteční hodnoty, ne opačné.

Uvažujme příklad z kap. 4.1.1, ale nahraďme t-přiřazení $\widehat{m2}\,[\![m2]\!]$ v $\widehat{C}$ přiřazením $\widehat{m2}\,[\![0]\!]$ a počáteční hodnotu $m2(0) = 1$.

$$\widehat{C} = \{\widehat{m1}\,[\![i1 \vee i2 \wedge m1]\!]\,, \widehat{m2}\,[\![0]\!]\,, \widehat{o1}\,[\![m1 \wedge \neg m2]\!]\,, \widehat{o2}\,[\![i3 \wedge o1]\!]\}.$$

Proměnná $m2$ potom danou podmínku nesplňuje. Její hodnoty jsou ve všech stavech následujících po počátečním stavu různé od její počáteční hodnoty, není možné ji modelovat jako konstantu $m2 = m2(0) = 1$. Ani definice $m2 = 0$ není korektní, protože výraz $m1 \wedge \neg m2$ pro $o1$ by měl být v počátečním stavu ohodnocen počátečními hodnotami $m1$ a $m2$, tedy logickou jedničkou pro $m2$.

Proměnné x takové, že $(\hat{x}\,[\![1]\!] \in \widehat{C}) \wedge x(0) = 0$ nebo $(\hat{x}\,[\![0]\!] \in \widehat{C}) \wedge x(0) = 1$ modeluji jako další stavové proměnné v SMV, například uvedenou proměnnou $m2$ takto

```
VAR m2 : boolean;
...
ASSIGN
 init(m2) := 1;
 next(m2) := 0;
...
```

Jestliže program obsahuje více takovýchto proměnných, některé by měly přiřazení typu `init(`x`):=1; next(`x`):=0;`, ostatní obráceně (`init(`x`):=0; next(`x`):=1;`). Není proto nutné definovat každou jako stavovou proměnnou, ale postačí jedna z nich. Ostatní se mohou definovat jako symboly referující tuto proměnnou nebo její negaci.

`MODULE PLC(b)`
`VAR`
` m_a : boolean;`
`ASSIGN`
` init(c) := 0;`
` init(m_a) := 0;`
` next(c) := d;`
` next(m_a) := a;`
`DEFINE`
` d := 0;`
` a := b;`
` e := m_a | c;`
	`MODULE PLC(b)`
	`VAR`
	` m_a : boolean;`
	`ASSIGN`
	` init(m_a) := 0;`
	` next(m_a) := a;`
	`DEFINE`
	` a := b;`
	` e := m_a;`

Obrázek 4.3. Příklad dosazení a odebrání konstant

4.2.2 Dosazení konstant do výrazů

Uvedený postup převodu definuje proměnné s konstantní hodnotou jako symboly v SMV a výrazy obsahující takovou proměnnou vypisuje nezměněné. Dosazením hodnoty do výrazu před jeho výpisem do kódu SMV jej můžeme zjednodušit. Konstantu nemusíme po dosazení do všech výrazů definovat, pokud ji ovšem nechceme ponechat v modelu z jiného důvodu (např. když se vyskytuje ve verifikované formuli). Kód z obr. 4.2 po dosazení a zrušení definice symbolu pro konstantu je

```
...
ASSIGN
 init(m1) := 0;
 init(m_o1) := 0;
 next(m1) := i1|m1&i2;
 next(m_o1) := o1;
DEFINE
 o1 := m1;
 o2 := m_o1&i3;
```

Po dosazení se může stát, že některá proměnná získá konstantní či kanonické přiřazení. Jestliže se nejedná o konstantní přiřazení určující jinou než počáteční hodnotu (viz kap. 4.2.1), je proměnná další konstantou, jejíž hodnotu opět můžeme do zbývajících výrazů dosadit. Tento postup lze iteračně opakovat do okamžiku, kdy dosazení hodnot všech známých konstant nezpůsobí nalezení další konstanty.

Obr. 4.3 ukazuje modely automatu generovaného AProgramem s trans-množinou

$$\widehat{C} = \{\hat{a}\,[\![b]\!]\,, \hat{c}\,[\![d]\!]\,, \hat{e}\,[\![a+c]\!]\}.$$

Množiny vstupních, vnitřních a výstupních proměnných jsou

$$\Sigma = \{b\},\ V = \{c, d\},\ \Omega = \{a, e\}$$

a počáteční hodnoty všech proměnných se rovnají logické 0. Levý model byl vytvořen bez dosazení konstant, pravý až po provedení všech iterací odebírajících konstanty.

4.2.3 Neinicializované proměnné

Dosavadní postup předpokládá, že je dán jeden počáteční stav automatu. To znamená, že známe hodnoty proměnných PLC, jehož program modelujeme, při spuštění programu. Počáteční stav SMV modelu určují přiřazení init všem stavovým proměnným.

Jestliže některé stavové proměnné v SMV neinicializujeme, získáme model, jenž nemá určený jeden počáteční stav, ale celou množinu možných počátečních stavů. Můžeme tak modelovat a verifikovat PLC program, který nemá určené počáteční hodnoty příslušných proměnných.

Jestliže neznáme počáteční hodnotu proměnné s kanonickým či konstantním přiřazením, nemůžeme ji modelovat jednoduše jako konstantu. Pro neinicializovanou proměnnou x s kanonickým nebo konstantním přiřazením, která má vliv na stavové nebo výstupní proměnné $\Big(x \in (V \cup \Omega) \cap \mathrm{dom}(\widehat{C} \widehat{\cap} (Z \cup \Omega))\Big)$, zapíšeme `next(x):=1` nebo `next(x):=0` nebo `next(x):=x` a neuvedeme pro ni přiřazení init.

Neinicializované proměnné nemají hodnotu, kterou bychom místo nich mohli dosadit do výrazů, nevztahuje se na ně proto odebírání konstant popsané v kap. 4.2.2.

4.3 Automat typu Moore

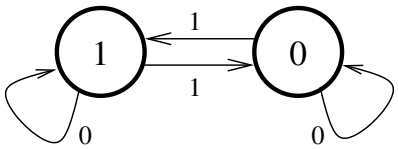
Výhodou SMV modelu automatu z def. 2.12 je fakt, že není nutné definovat všechny výstupy jako proměnné SMV. Stavová SMV proměnná existuje jen pro výstupní proměnné z množiny $(\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C}))$, tedy takové, že se vyskytují na pravé straně některého t-přiřazení. Samotné výstupy se definují pouze jako symboly a hodnota, kterou představují, je určená momentálním vstupem. Výstupy tedy okamžitě reagují na vstupy.

U skutečného PLC je doba trvání program scanu nenulová a tak dochází k určitému zpoždění výstupů za vstupy. Článek [11] popisuje metodu modelování PLC programu takovou, že přiřazení next určují hodnoty výstupů PLC stejně jako pro stavové proměnné. Model PLC programu autoři spojují v SMV s modelem řízeného procesu. Díky zpoždění výstupu za vstupem jejich model vyjadřuje souběh scan cyklu s činností řízeného procesu (moduly řídicího programu a procesu ale nesmějí být definovány jako asynchronní procesy – viz str.28).

Změnou popisu výstupů modelu, jehož tvorbou se zabývají podkapitoly 4.1 a 4.2, ze symbolů na proměnné s přiřazením `next` dosáhneme modelu, jehož výstupy jsou o jeden časový krok zpožděné za vstupy. Pro výstupy, které se vyskytují na pravé straně nějakého přiřazení z trans-množiny programu, existují v modelu Mealy automatu stavové proměnné.

$\{open\ [\![(\neg input \wedge open) \vee (input \wedge \neg open)]\!]\}$

```
MODULE automat(input)
VAR
 open : boolean;
ASSIGN
 init(open) := 0;
 next(open) := !input&open|input&!open;
```



Obrázek 4.4. Model automatu typu Moore v SMV

```
MODULE PLC(i1, i2, i3)
VAR
 m1 : boolean;
 m_o1 : boolean;
ASSIGN
 init(m1) := 0;
 init(m_o1) := 0;
 next(m1) := i1|m1&i2;
 next(m_o1) := o1;
DEFINE
 o1 := m1;
 o2 := m_o1&i3;
```

```
MODULE PLC(i1, i2, i3)
VAR
 m1 : boolean;
 o1 : boolean;
 o2 : boolean
ASSIGN
 init(m1) := 0;
 init(o1) := 0;
 init(o2) := 0;
 next(m1) := i1|m1&i2;
 next(o1) := m1;
 next(o2) := o1&i3;
```

Obrázek 4.5. Modely automatů typu Mealy a Moore získaných ze stejného programu

Po posunutí výstupu o jeden časový krok ho určuje stejné přiřazení `next` jako přiřazení pro příslušnou stavovou proměnnou, výstup a stavová proměnná tak splývají a není nutné je definovat zvlášť, postačí pouze jedna proměnná v SMV pro jednu výstupní proměnnou programu.

Posunutím výstupu vzniká model Moore automatu podobný tomu z def. 2.3. Jeho přechodovou funkci popisují t-přiřazení z množiny $(\widehat{\mathrm{co}_\mathrm{p}}(\widehat{C}) \cap \widehat{\mathrm{dom}}(\widehat{C})) \cup (\widehat{C} \widehat{\cap} \Omega)$. Oproti modelu Mealy automatu jsou mezi stavovými proměnnými i výstupy, na kterých žádná proměnná PLC nezávisí. Pokud bychom chtěli modelovat i proměnné, které se v uvedené množině nevyskytují, museli bychom pro ně uvést přiřazení `next` jako pro ostatní, pouze v případě konstanty můžeme použít definici symbolu. Výstupní funkce tohoto automatu pouze určuje, který výstup je určen kterou stavovou proměnnou. Na obr. 4.4 vidíme model Moore automatu generovaného stejným programem jako Mealy automat z obr. 4.1.

4.4 Vstupy automatu

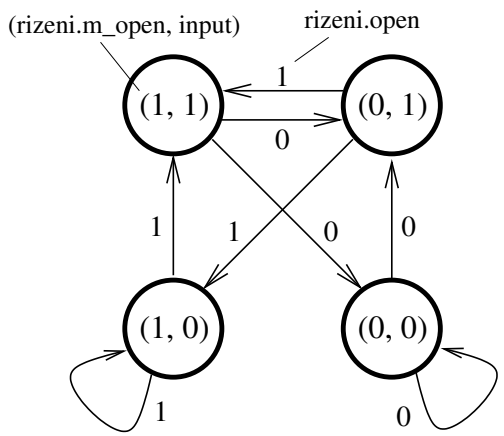
Model automatu se podle výše popsaných postupů vytváří jako SMV modul. Pro jeho verifikaci je nutné vytvořit povinný modul main a instanci modulu modelujícího automat a definovat jeho vstupy.

Pro jednoduchost předpokládejme, že neznáme popis chování vstupů nebo že chceme verifikovat proti všem, i s popisem se neshodujícím, vstupním sekvencím. Potom vstup automatu může nabývat jakékoliv hodnoty v každém okamžiku. Logická proměnná, která může kdykoliv mít jakoukoliv logickou hodnotu, se v SMV může definovat jako proměnná typu boolean s tím, že se neuvede žádné přiřazení `next` nebo `init` ani rovnici v sekcích TRANS nebo INIT, která by její hodnotu určovala. Nepopsané vstupy můžeme definovat společně s instancí modulu modelujícího automat v hlavním modulu SMV modelu.

Kompozicí modelu automatu se vstupy definovanými jako nedeterministické proměnné vzniká model nedeterministického automatu. Kompozici modelu z obr. 4.1 s nepopsaným vstupem ukazuje obr. 4.6.

```
MODULE automat(input)
VAR
 open : boolean;
ASSIGN
 init(m_open) := 0;
 next(m_open) := open;
DEFINE
 open := !input&open|input&!open;

MODULE main
VAR
 input : boolean;
 rizeni : automat(input);
```



Obrázek 4.6. Kompozice modelu automatu a vstupu

4.5 Inicializační procedury

Programovatelné logické automaty mívají zabudované prostřednky programové inicializace. Inicializace PLC se většinou využívá pouze pro nastavení proměnných na požadované počáteční hodnoty. V takovém případě můžeme ověřit inicializaci zvlášť a hodnoty jí nastavené použít při verifikaci automatu získaného z hlavního programu jako počáteční hodnoty stavových proměnných modelu. Pokud je inicializace obecnější nebo ji z nějakého důvodu chceme verifikovat zároveň s modelem hlavního programu, rozšíříme model automatu získaného překladem programu o popis inicializace.

Inicializační proceduru pomocí proměnné indikující první scan cyklus nebo pomocí inicializačního programu s jedním průchodem můžeme do SMV modelu zahrnout následujícím způsobem.

Indikační proměnná má v prvním scan cyklu logickou hodnotu rovnou 1 a ve všech dalších cyklech rovnou 0. V SMV modelu, kde jeden časový krok odpovídá jednomu scan cyklu, se taková proměnná popíše jednoduše, pojmenujme ji například `firstscan`. Její popis pak je

```
 init(firstscan) := 1;
 next(firstscan) := 0;
```

a přiřazení programově inicializovaných proměnných a proměnných na nich závislých mohou obsahovat proměnnou `firstscan`.

Inicializační program zapsaný jako AProgram lze převést do trans-množiny, nazvěme ji $\widehat{C}_I$. Potom inicializační přiřazení stavové proměnné x je $\widehat{C}_I \uparrow x$. Rozlišení, zda se má v určitém okamžiku proměnné přiřadit hodnota podle inicializačního přiřazení nebo podle přiřazení z trans-množiny hlavního programu, lze provést například zavedením proměnné indikující provádění inicializačního programu (pojmenujme ji například opět `firstscan`), ta je v prvním okamžiku rovna 1, poté 0, stejně jako proměnná použitá u inicializace pomocí proměnné indikující první průchod programu. Nazvěme *x-výraz* zápis výrazu z t-přiřazení $(\widehat{C} \uparrow x)$ a *x-initvýraz* z $(\widehat{C}_I \uparrow x)$. Proměnnou `x` pak můžeme modelovat

```
next(x) := case
  firstscan : x-initvýraz;
  1 : x-výraz;
esac;
```

nebo

```
next(x) := firstscan&x-initvýraz | !firstscan&x-výraz;
```

Inicializační procedury lze mezi sebou převádět, stejně tak jejich modely. Výše uvedený výraz pro proměnnou `x` by mohl být výrazem t-přiřazení z trans-množiny získané překladem hlavního programu využívajícího pro inicializaci indikační proměnnou `firstscan`.

Podobně výraz pro proměnnou z programu inicializovaného využitím indikační systémové proměnné můžeme rozdělit na inicializační výraz a výraz pro další přechody – inicializační výraz získáme dosazením 1 za indikační proměnnou do původního výrazu, výraz pro další přechody dosazením 0.

V kapitolách 4.2.1 a 4.2.2, které se zabývají konstantami v programu, se za konstantu považuje proměnná, která má určenou počáteční hodnotu a výraz pro následující hodnotu ji zachovává. Při modelování inicializační procedury je konstantou proměnná, jejíž počáteční hodnotu zachovává jak t-přiřazení hlavního programu, tak i inicializační přiřazení.

4.6 Specifikace verifikovaných formulí

Verifikací v SMV se zabývá kap. 3.3.2, zde uvádím jen příklad specifikace požadavků na konkrétním modelu. Vezměme SMV modely z obr. 4.5 na str. 41 a definujme jejich instance v hlavním SMV modulu.

```
MODULE main
VAR
 i1 : boolean;
 i2 : boolean;
 i3 : boolean;
 plc : PLC(i1, i2, i3);
```

Formuli vyjadřující, že při hodnotě vstupu `i3`=0 je vždy výstup `o2`=0, zapíšeme pro model Mealy automatu

```
AG (!i3 -> !plc.o2).
```

Formule je pravdivá.

Jestliže byl automat získán překladem PLC programu, můžeme požadavek interpretovat tak, že po načtení hodnoty 0 ze vstupu `i3` během input scanu bude výstup `i2`=0 po provedení prvního následujícího output scanu. V modelu Moore automatu jsou výstupy o jeden časový krok opožděné za vstupy, stejný požedavek by se potom zapsal

```
AG (!i3 -> AX !plc.o2).
```

Kapitola 5

Převod automatu do Uppaalu

Tato kapitola předkládá metody, které jsem navrhl pro modelování automatu popsaného logickými rovnicemi v nástroji UPPAAL. První části obsahují popis několika variant převodu samotného automatu. Část 5.2 ukazuje rozšíření o inicializaci automatu a část 5.3 rozšíření modelu o časovače.

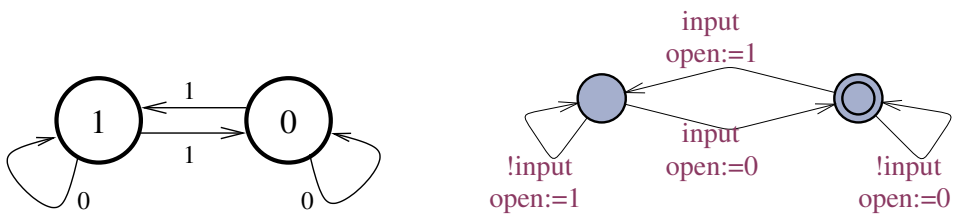
Mnoho prvků postupu je shodných nebo podobných s převodem do SMV, zde některé uvedu stručněji a odkážu se na související část kapitoly o převodu do SMV.

5.1 Model automatu

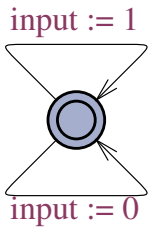
Časový automat, z něhož vychází UPPAAL, je zcela jiná struktura než klasické automaty typu Moore nebo Mealy. Vezměme automat typu Moore. Místa v jeho grafu přechodů bychom mohli interpretovat jako místa v modelu UPPAALu, jeho hrany jako hrany modelu. Takto převedený graf přechodů ukazuje obr. 5.1.

Místa modelu odpovídají hodnotám stavové proměnné `input` stejně jako stavy původního automatu – když je proces v levém místě, je vždy `open=1`, v pravém vždy `open=0`. Sémantika hran je ale různá. Automat reaguje na posloupnost vstupů, každému prvku ze vstupní posloupnosti odpovídá jeden přechod. Model v UPPAALu se interpretuje v reálném čase. Zobrazený model má v každém okamžiku některou z hran volnou k přechodu, dojít k němu ale může kdykoliv – ihned, s nějakým zpožděním a nebo také nikdy.

Uvedený model nepopisuje vstup automatu. Na rozdíl od SMV nemá v UPPAALu nepopsaná proměnná náhodně se měnící hodnotu, ale naopak, dokud ji žádná hrana nezmění,



Obrázek 5.1. Graf přechodů Moore automatu a jeho zobrazení v UPPAALu



Obrázek 5.2. Model náhodně se měnícího vstupu

zůstává stále stejná hodnota. Vstupní proměnnou, jejíž hodnota se náhodně mění, může popsat model na obr. 5.2.

Kompozicí modelu náhodné proměnné s modelem sestrojeným podle grafu přechodů automatu (obr. 5.1) vznikne model, kde průběh výstupu není přesně určen průběhem hodnot vstupu, protože oba procesy běží asynchronně. Může se stát, že na některou změnu vstupu druhý proces vůbec nezareaguje. Posloupnost výstupů odpovídající posloupnosti vstupů docílíme synchronizací obou procesů binárním kanálem (nazvěme ho třeba `kan`). Všem hranám v jednom procesu přiřadíme synchronizační návěští `kan!` a všem hranám ve druhém procesu `kan?`. Model se pak již podobá Moore automatu, ale v reálném čase – „následující“ hodnotou vstupu automatu je hodnota, kterou mu přiřadí přechod procesu vstupu, který se provede od daného okamžiku jako první, podobně je tomu i s hodnotami stavových proměnných.

Přestože posloupnosti vstupů v synchronizovaném modelu odpovídá posloupnost výstupů, není zaručeno, že vůbec existuje „následující vstup“, protože k přechodům dochází v libovolném čase – k dalšímu přechodu nemusí vůbec dojít. Tomuto jevu lze zabránit například následujícími způsoby.

- Nastavení nulové doby mezi přechody znemožní neprovedení přechodu, protože se musí provést okamžitě. Nulové zpoždění dosáhneme použitím urgentních míst v jednom ze synchronizovaných procesů.
- Nastavení pevné nenulové doby mezi přechody nebo intervalu, ve kterém se doba může pohybovat, také znemožní neprovedení přechodu. Navíc v takovém modelu na rozdíl od nulového zpoždění dochází k vývoji času, čehož lze využít k verifikaci časových charakteristik modelu získaného modelováním systému pracujícího v reálném čase. Například u modelu PLC programu doba mezi přechody odpovídá délce scan cyklu.

Zatím jsme uvažovali pouze model s jedním vstupem a jednou stavovou proměnnou. Více vstupních proměnných popíšeme více procesy, pro každý vstup jeden proces, který mu přiřazuje náhodnou hodnotu. Synchronizaci více procesů provádí kanál typu broadcast, jeden z procesů bude mít na hranách návěští `kan!`, všechny ostatní `kan?`. Synchronizace broadcast se vyznačuje tím, že s hranou emitující kanál (`kan!`) se synchronizují všechny právě volné přijímající hrany (`kan?`). Při časovém omezení aplikovaném v procesu, který přijímá broadcast kanál, může v ostatních procesech dojít k přechodu i pokud je časové omezení porušeno, při přechodu se pouze daný proces vynechá. Aby byla omezení na dobu mezi přechody v modelu automatu funkční, musí se všechna zavést do procesu emitujícího broadcast kanál.

Synchronizační kanál může emitovat některý z procesů pro vstup nebo stav automatu.

Můžeme ale také zavést další proces, který bude emitovat kanál a bude tedy také obsahovat případná časová omezení.

Proces modelující stav automatu můžeme zjednodušit použitím broadcast synchronizace kanálem emitovaným z jiného procesu: Hrany vedoucí do stejného místa, odkud vycházejí, a zároveň neměnící hodnotu žádné proměnné jsou zbytečné, protože nijak nemění stav modelu a nejsou již ani potřebné pro synchronizaci. Odstraněním takových hran (na obr. 5.1 hrany se střežením `!input`) se chování modelu z pohledu proměnných nijak nezmění. Model s jednou stavovou proměnnou bychom mohli dokonce zjednodušit na jedno místo a jednu hranu, která by neměla žádné střežení a přiřazovala by proměnné výraz určující její hodnotu. Pro automat popsaný logickými rovnicemi se jedná o výraz z pravé strany příslušné rovnice (t-přiřazení), jinak ho můžeme vypočítat jako $(!x \wedge splus \vee x \wedge !sminus)$, kde x je stavová proměnná, $splus$ střežení na původní hraně přiřazující hodnotu 1 a $sminus$ střežení hrany přiřazující 0.

Jak popsat automat s více než dvěma stavy, tedy s více než jednou binární stavovou proměnnou? Nabízí se několik variant,

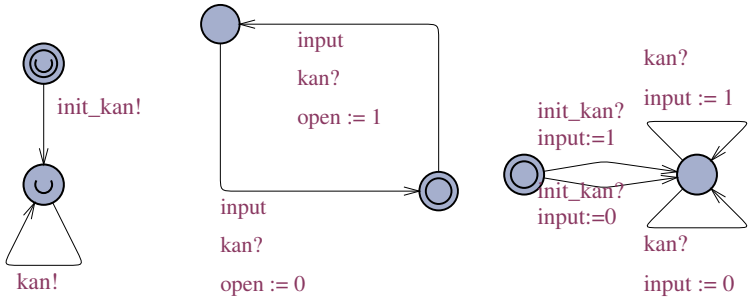
- modelovat automat jako jeden proces, kde každému místu odpovídá jeden stav automatu a hrany přiřazují proměnným hodnoty podle toho, do kterého místa vedou,
- rozdělit automat na síť dvoustavových automatů, každý pro jednu proměnnou, a ty modelovat jako procesy se dvěma stavy,
- rozdělit automat na síť dvoustavových automatů a ty pak modelovat jako procesy s jedním místem a jedním nestřeženým přechodem určujícím hodnotu podle výrazu pro proměnnou.

Zdánlivě tu chybí možnost modelovat celý automat jako jeden proces o jednom místu a jednom přechodu, který by přiřazoval všem proměnným výrazy. Tato varianta však není možná, protože hrana sice může přiřadit hodnotu více proměnným, ale tato přiřazení se neprovedou zároveň, ale sekvenčně. Závisela-li by některá proměnná x na proměnné y, jejíž přiřazení by bylo uvedeno před přiřazením pro x, neurčovala by se nová hodnota x podle hodnoty y před provedením přechodu, ale podle její nové hodnoty.

Vstup popsaný použitým procesem se sice náhodně mění, ale jeho počáteční hodnota je pevná – není-li v deklaraci uvedeno jinak, je nulová. Pro nastavení náhodného vstupu před prvním provedením přechodu v modelu automatu zavedeme do synchronizačního procesu oddělené počáteční místo. Hrana z něj vedoucí bude synchronizovat nastavení náhodných počátečních hodnot vstupů, viz obr. 5.3.

Na SMV modelu automatu typu Moore můžeme verifikovat formule vyjadřující požadavek na okamžitou reakci výstupu na určitý vstup, např. `AG (!i3 -> AX !plc.o2)` na modelu z obr. 4.5 vpravo. UPPAAL neumí takovou formuli vyjádřit. Případné splnění podobné formule `!i3 --> !plc.o2` nemusí zaručit požadavek, protože tato formule může platit i pokud je reakce delší než jeden přechod automatu. Verifikaci požadavku na reakci během jednoho stavového přechodu (scan cyklu v modelu PLC programu) můžeme provést následujícím postupem.

1. V procesu, který synchronizuje procesy automatu a vstupů rozdělíme synchronizační hranu na dvě hrany oddělené dalším místem.



Obrázek 5.3. Model automatu s náhodnou počáteční hodnotou vstupu a nulovou dobou mezi přechody

2. První hrana bude emitovat kanál synchronizující proces(y) automatu, druhá bude emitovat kanál synchronizující procesy vstupů. Kanál pro synchronizaci vstupů se může využít i pro nastavení jejich náhodné počáteční hodnoty.
3. Model má v nově vzniklém místu nové hodnoty výstupů a vnitřních proměnných, ale původní hodnoty vstupů, tedy hodnoty, ze kterých byly nové výstupy vypočítány. Místo pojmenujme, proměnnou symbolizující přítomnost synchronizačního procesu v tomto místě pak můžeme využít ve specifikaci verifikovaných formulí.

Pokud budeme za stav automatu brát stav proměnných modelu během přítomnosti v nově vytvořeném místu a sekvenci ⟨hrana synchronizující vstupy → původní místo synchronizačního procesu → hrana synchronizující automat⟩ budeme považovat za přechod automatu, získáme tak z pohledu vstupů a výstupů model Mealyho automatu.

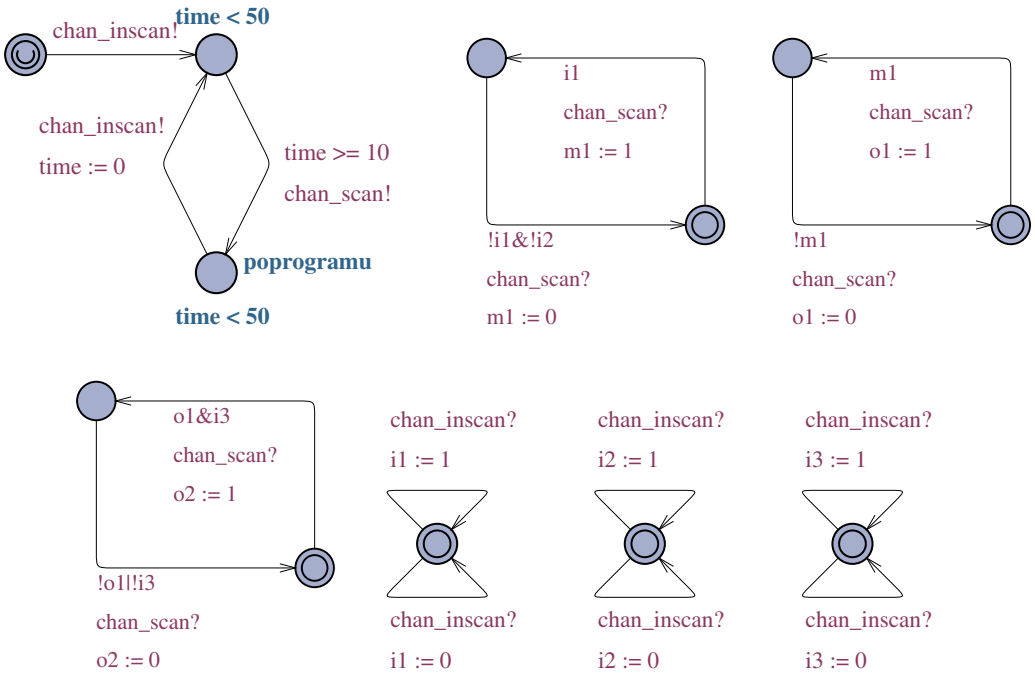
Obr. 5.4 ukazuje model automatu rozloženého na procesy jednotlivých proměnných a synchronizovaného oddělenými kanály pro vstupy a automat. Automat vychází z logických rovnic příkladu v části 4.1.1, jeho SMV modely jako Mealy i Moore automat ukazuje obr. 4.5. Cyklické provádění synchronizace je v tomto UPPAAL modelu zajištěno omezeními na hodiny, konkrétně doba mezi dvěma načteními vstupů může být 10 až 50 časových jednotek. Místo mezi hranou pro synchronizaci automatu a hranou pro synchronizaci vstupů se jmenuje `poprogramu`, formuli vyjadřující okamžitou reakci výstupu `o2` zapíšeme

```
A[] (proc_cycle.poprogramu && !i3 imply !o2).
```

Popis modelování automatu v UPPAALu se zatím nezabýval skutečností, které proměnné vlastně modeluje. Při výběru proměnných můžeme vyjít z pravidel pro popis automatu typu Moore v kap. 4.3 a modelovat proměnné z množiny

$$(\mathrm{co_p}(\widehat{C}) \cap \mathrm{dom}(\widehat{C})) \cup \Omega,$$

kde $\widehat{C}$ je trans-množina získaná překladem APLC programu a Ω množina výstupů. Množinu proměnných můžeme omezit hledáním konstant a dosazováním jejich hodnot do výrazů pro ostatní proměnné podle kap. 4.2.2. Konstantu v UPPAALu definujeme klíčovým slovem `const` a odstranění proměnné znamená odstranění její deklarace a buď odstranění jejího procesu nebo zjednodušení procesu, který kromě ní popisuje i jiné proměnné.



Obrázek 5.4. Model s nenulovou dobou mezi přechody

5.2 Inicializace

Inicializací můžeme myslet nastavení počáteční hodnoty proměnné nebo programovou inicializaci, tj. situaci, kdy rovnice pro první přechod automatu přiřazují proměnným jiné hodnoty než další přechody.

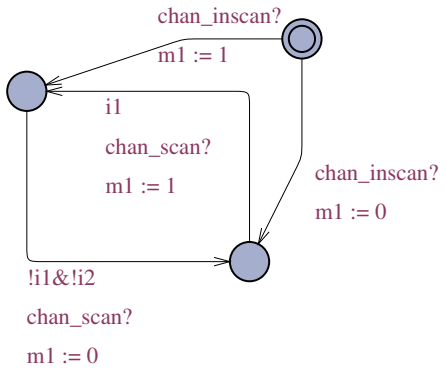
Počáteční hodnoty proměnných

Modely zobrazené v kap. 5.1 mají nulovou počáteční hodnotu stavových proměnných, procesy automatu mají jako počáteční místa označená místa pro hodnotu 0. Opačnou počáteční hodnotu proměnné lze nastavit volbou počátečního místa pro 1 a deklarací proměnné s odpovídající počáteční hodnotou, např. `bool x:=true;`

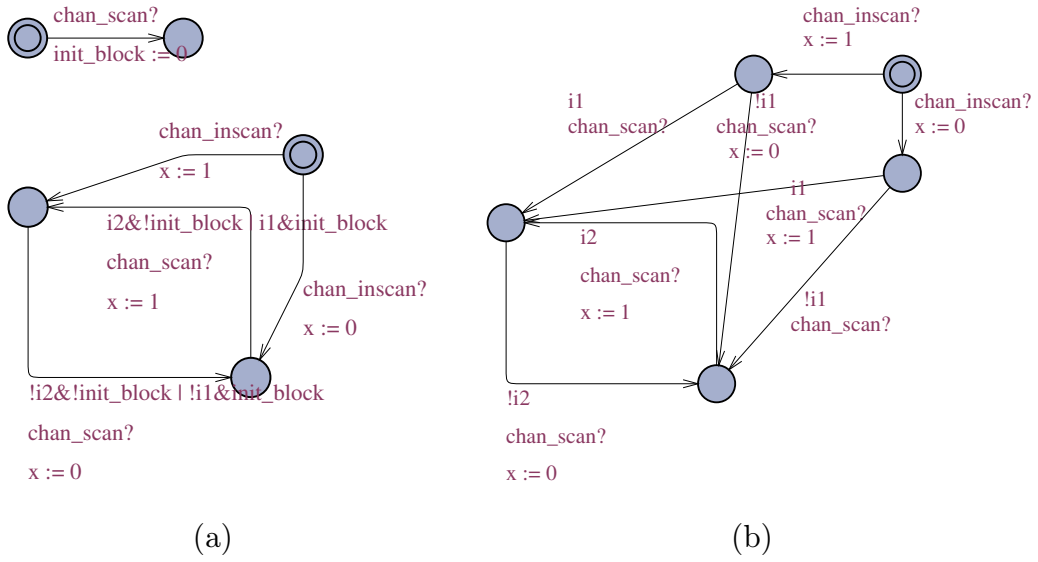
Pokud chceme modelovat automat s neurčeným počátečním stavem (některé proměnné modelovat jako neinicializované), můžeme nastavení náhodných počátečních hodnot proměnných provést podobně jako u vstupů. Do procesu(ů) automatu přidáme nové počáteční místo, ze kterého povedou hrany do míst pro možné počáteční stavy automatu. Inicializační hrany budou proměnným přiřazovat odpovídající hodnoty. Model automatu s takovou neinicializovanou proměnnou ukazuje obr. 5.5.

Programová inicializace

Programová inicializace PLC je prostředek pro počáteční nastavení programu. Prostředky inicializace jsou různé, např. v předchozích kapitolách zmíněné použití speciální proměnné, jejíž hodnota indikuje první scan cyklus, nebo použití inicializačního programu, který proběhne jednou před prováděním hlavního programu. Jak ukazuje kap. 4.5, lze oba způsoby



Obrázek 5.5. Proces neinicializované proměnné



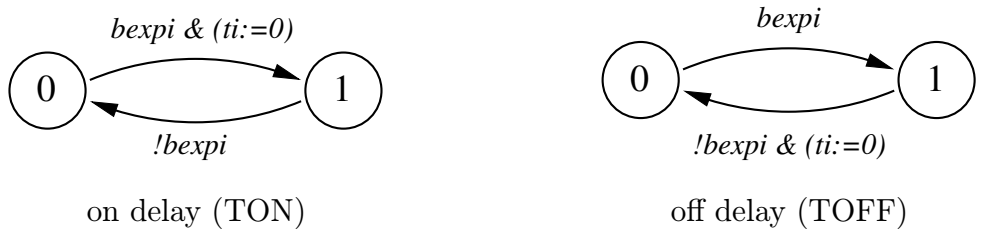
Obrázek 5.6. Dva způsoby modelování programové inicializace

popsat společnými rovnicemi pro první přechod automatu a následující přechody nebo dvěma rovnicemi pro první a následující změny každé proměnné.

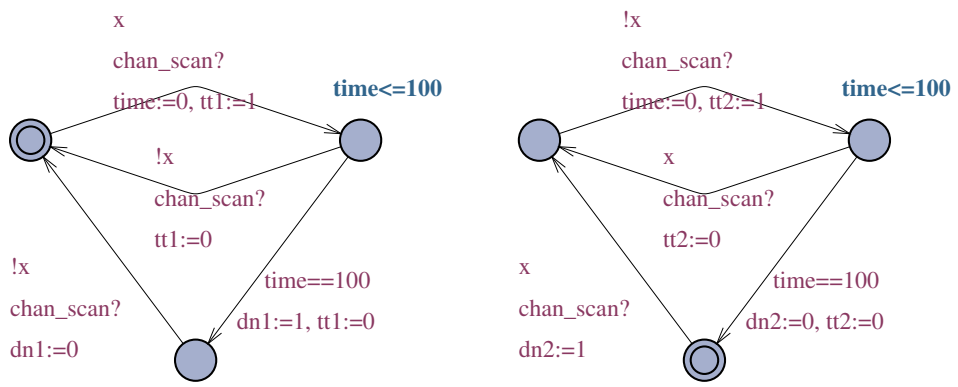
Inicializace společnými rovnicemi nemění strukturu modelu, pouze musíme navíc modelovat proměnnou indikující první přechod automatu. Počáteční hodnota této proměnné je 1, s prvním přechodem se změní na 0.

Inicializaci oddělenými rovnicemi můžeme provést například přidáním dalších míst a hran do modelu automatu. Nové hrany budou provádět přiřazení podle inicializačních rovnic, původní hrany podle rovnic pro další změny hodnot proměnných.

Mějme automat s jednou stavovou proměnnou x a vstupy i_1 a i_2. Jeho inicializační t-přiřazení je $\hat{x}\,[\![i_1]\!]$ a přiřazení hlavního programu $\hat{x}\,[\![i_2]\!]$. Společná rovnice při použití indikační proměnné $init\_block$ je $\hat{x}\,[\![init\_block \cdot i_1 + !init\_block \cdot i_2]\!]$. Oba popisy modelované v UPPAALu jsou na obr. 5.6 (počáteční hodnota x je navíc náhodná).



Obrázek 5.7. Časové automaty pro časovače



Obrázek 5.8. Modely časovačů TON (vlevo) a TOFF (vpravo) v UPPAALu

5.3 Časovače

Práce [14] popisuje rozšíření automatu modelujícího PLC program o časovače typu TON a TOFF. Vstup časovače se v APLC programu popíše jako nově vytvořená proměnná, zápisem do této proměnné nahrazujeme instrukci spouštění časovače. Výstupy časovače se jeví jako vstupy automatu popsaného logickými rovnicemi. Popisovány jsou i resetovací vstupy časovačů, ale těmi se zde nebudu zabývat. Časové automaty časovačů převzaté z [14] jsou na obr. 5.7, $bexp_i$ je výraz z t-přiřazení pro proměnnou symbolizující vstup časovače, t_i jsou hodiny pro měření času v automatu.

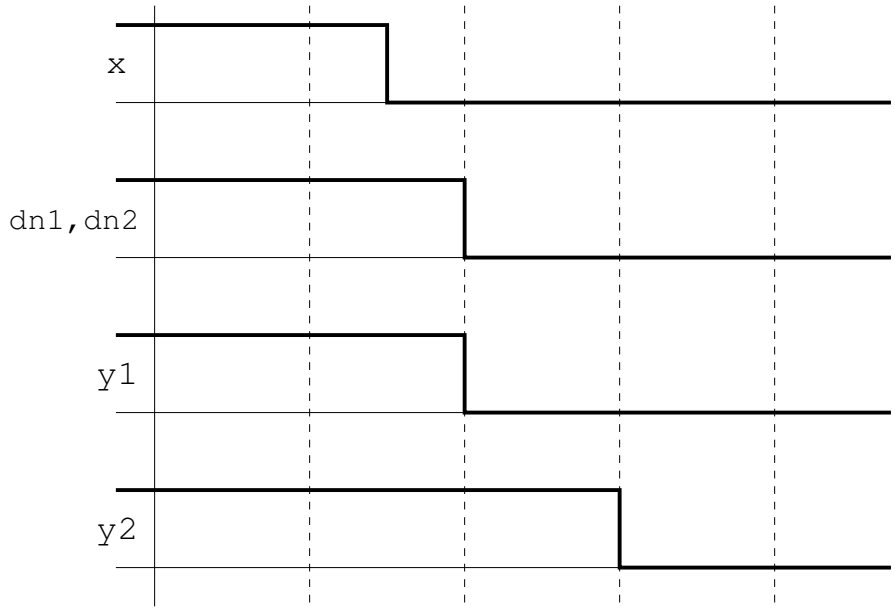
Výstupy automatu definují výrazy

TON	DN	$bexp_i \wedge (t_i \geq c_i)$
	TT	$bexp_i \wedge (t_i \leq c_i)$
TOFF	DN	$bexp_i \vee \neg(t_i \geq c_i)$
	TT	$\neg bexp_i \wedge (t_i \leq c_i)$

c_i označuje časové zpoždění časovače.

Výstupy časovačů můžeme v UPPAALu modelovat jako další proměnné, jejichž hodnoty se mění při změně místa automatu podle obr. 5.7 a při překročení zpoždění časovače. Místo, které symbolizuje běh časovače (1 v automatu pro TON, 0 v automatu pro TOFF), v UPPAALu rozdělíme na dvě místa – jedno pro čas menší než je hodnota zpoždění, druhé pro čas větší. Obr. 5.8 znázorňuje modely časovačů typu TON a TOFF, zpoždění obou časovačů je 100 jednotek, proměnné symbolizující jejich vstupy ($vstup_1$, $vstup_2$) mají přiřazení $vstup_i$ $[\![x]\!]$.

Uvedené modely v UPPAALu nemodelují časové automaty přesně. Změny výstupů modelů způsobené změnou jejich vstupů se provedou při přechodu hrany synchronizující model



Obrázek 5.9. Průběhy po změně vstupu časovačů

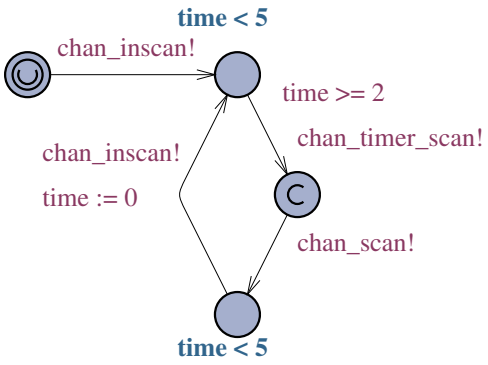
automatu, proměnné na nich závislé tedy na tuto změnu mohou reagovat až při další synchronizaci (dalším přechodu). Naproti tomu v popisu automatu se pro výstupy nezavádějí další proměnné, ale nahrazují se výrazy určujícími jejich hodnotu – proměnné na nich závislé reagují ihned. Rozdíl demonstruje příklad.

Mějme následující programy. Oba pracují s časovačem typu TON a oba popisuje transmnožina $\{input_i \, [\![x]\!] \, , y_i \, [\![dn_i]\!]\}$.

```
load x                  load dn2
store input1            store y2
load dn1                load x
store y1                store input2
```

Představme si, že se nacházíme v okamžiku, kdy je výstup `dn` obou časovačů rovný 1, a vstup `x` změníme z 1 na 0. V prvním programu se nejdříve načte vstup `x`, vzhledem k jeho hodnotě se následující instrukcí časovač zastaví – jeho výstup `dn1` se změní na 0. Nová hodnota výstupu se zapíše dalšími dvěma instrukcemi do `y1`. Druhý program nejdříve načte vstup stále ještě běžícího časovače a jeho hodnotu 1 uloží do `y2`, poté se teprve načte nová hodnota vstupu a časovač se zastaví. Průběhy hodnot ukazuje obr. 5.9, čárkované čáry představují okamžiky proběhnutí programu při abstrakci nulové doby provedení program scanu. Popis pomocí automatu modeluje první program. Popis modelem v UPPAALu, kde proces časovače je synchronizován stejným signálem jako proces(y) automatu, modeluje druhý program.

Reakci proměnné přímo závislé na výstupu časovače na změnu vstupu časovače takovou, jako představuje první z uvedených programů, dosáhneme dvěma způsoby: buď dosazením výrazu pro výstup časovače za proměnnou po vzoru popisu pomocí automatů z obr. 5.7, nebo spuštěním přechodu v procesu časovače před spuštěním přechodu v procesu automatu popsaného ligickými rovnicemi: Hranu synchronizačního procesu, která spouští procesy automatu a časovačů rozdělíme na dvě hrany (obr. 5.10). První z nich bude synchronizovat časovače, jejichž výstupy se využívají k určení hodnot ostatních proměnných až po zápisu proměnné symbolizující vstup časovače. Druhá hrana bude synchronizovat procesy automatu popsaného



Obrázek 5.10. Synchronizační proces pro časovače spouštěné před čtením jejich výstupu

logickými rovnicemi a procesy časovačů, jejichž výstupy se naopak čtou pouze před zápisem do jejich vstupů.

Modelování časovačů při abstrakci nulové doby proběhnutí program scanu nemůže postihnout některé jevy, například to, že ve skutečném programu může mezi dvěma čteními výstupu časovače v rámci jednoho cyklu uplynout jeho čas a změnit výstup (pokud použité PLC mění výstupy časovače během program scanu podle uběhnutého času). V modelu časovačů je časové zpoždění konstantou, PLC ho umožňují v průběhu programu měnit. Program, který nemá pevné zpoždění všech časovačů, nelze popsaným způsobem modelovat.

Kapitola 6

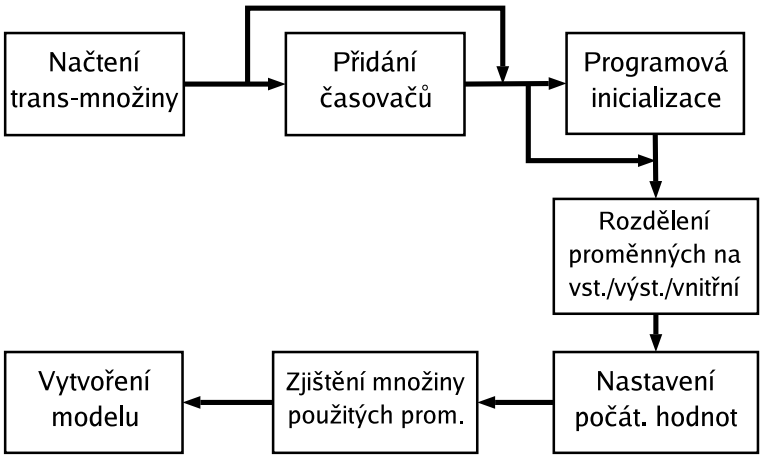
Implementace

Postupy převodu jsem naprogramoval jako aplikaci s uživatelským prostředím založeným na zadávání parametrů převodu v jednotlivých krocích. Po zadání potřebných parametrů se vygeneruje soubor s modelem automatu. Soubor pro SMV obsahuje textový popis systému tak, jak byl ukázán v kap. 3.3 a 4, tedy pouze s využitím základní syntaxe podle [10]. Jako formát výstupu jsem zvolil variantu XML, protože umožňuje ukládat grafickou informaci a při práci v grafickém prostředí UPPAAL nemusí provádět konverzi, případné změny modelu přímo v UPPAALu se mohou uložit do stejného souboru jako model vygenerovaný transformačním programem.

Převodní aplikace je naprogramována v jazyku Java [22]. Tento programovací jazyk jsem zvolil především proto, že umožňuje vývoj aplikací spustitelných na mnoha hardwarových platformách a pod různými operačními systémy. Jedinou podmínkou spuštění aplikace je přítomnost dostatečně aktuálního prostředí *Java Runtime Environment* (JRE) na použitém systému. (Aplikaci jsem vyvíjel pro verzi JRE 1.4) S využitím knihoven *Java Foundation Classes* (JFC) *Swing* jsem mohl snadno realizovat grafické uživatelské prostředí. Pro vytvoření nápovědy jako okna aplikace jsem použil software *JavaHelp* přístupný stejně jako JRE z [22].

Vstupem programu je trans-množina získaná překladem APLC programu algoritmem APLCTrans, jako jediný vstup překladu ale nestačí, protože neobsahuje informace o tom, která proměnná je vstupem, která výstupem, jeké jsou jejich počáteční hodnoty atd. Aplikace umí vytvářet modely programové inicializace a časovačů, které vyžadují další doplňkové informace. Vkládání většiny informací se děje manuálním zásahem uživatele. Pro dosažení jednoznačnosti je postup modelování rozdělen na několik částí, které umožňují jednotlivé informace zadávat zcela odděleně. V každé části se zadávají parametry, jejichž případná kolize s parametry zadanými dříve se kontroluje při přechodu na další část. Výběr většiny parametrů je dokonce omezen ovládacími prvky tak, aby ke kolizi nemohlo dojít. Postup zadávání uvedený na obr. 6.1 je v aplikaci rozdělen na jednotlivé „karty“ – okna, která obsahují vstupní a ovládací prvky příslušné většinou právě jednomu kroku z obrázku.

Podkapitola 6.1 popisuje sekvenční zadávání parametrů převodu a některé rysy samotného generování modelu. Zmiňuje se i o obsahu odpovídajících karet převodní aplikace, většina z nich však není v této kapitole zobrazena. Pro vyobrazení všech karet viz přílohu C ukazující využití aplikace pro převod dvou jednoduchých programů krok za krokem. Část 6.2 obsahuje přehled některých dalších parametrů modelu, na které není uživatel přímo dotázán, a nastavení



Obrázek 6.1. Kroky nastavení parametrů pro překlad automatu

ovlivňující běh aplikace.

6.1 Jednotlivé kroky převodu

Jednotlivé kroky postupu z obr. 6.1 většinou odpovídají jednotlivým kartám aplikace. Je tomu tak pro kroky od načtení trans-množiny až po nastavení počátečních hodnot. Určení parametrů, které omezují množinu použitých proměnných se děje z části na kartě obsahující i prvky pro volbu výstupního systému (SMV×UPPAAL).

Mezi kartami se přechází stiskem tlačítka 'Next' pro posun na následující kartu a tlačítkem 'Back' pro přesun na předchozí kartu. Po zadání informací náležejících svým významem příslušnému kroku tedy stiskneme tlačítko 'Next'. Schéma postupu naznačuje, že časovače a inicializaci není třeba zadávat, v takovém případě na odpovídajících kartách nevyplňujeme žádné údaje, ale ihned můžeme přejít na další krok. Na obr. 6.2 je zobrazené okno aplikace s úvodní kartou pro vložení trans-množiny (jako první karta neobsahuje tlačítko 'Back').

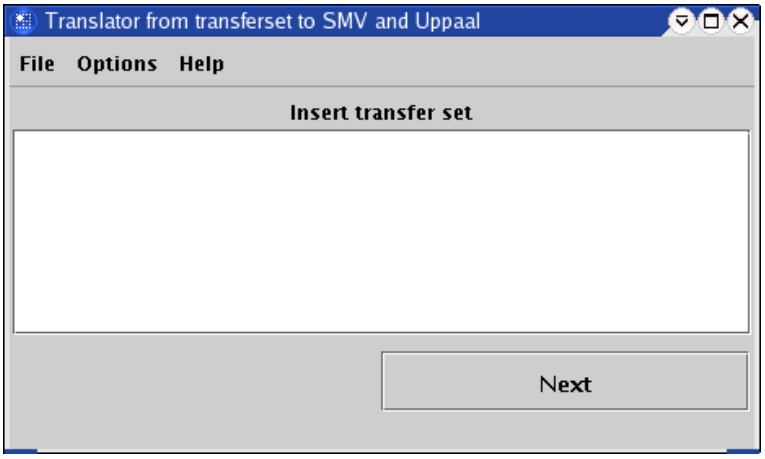
6.1.1 Vložení trans-množiny

Hlavním vstupem překladu je trans-množina, ze které vychází popis automatu. Trans-množinu můžeme získat překladem APLC programu algoritmem APLCTRANS, proto je vstupní formát zvolen tak, aby bylo možné použít přímo výstup programu APLCTrans implementujícího daný algoritmus. Příklad výstupu programu je na obr. 2.3. Výstup obsahuje informativní údaje, samotná trans-množina je uvedená je složených závorkách za 'Result='.

Aplikace načítá trans-množinu ze vstupního textového pole od prvního výskytu závorky '{' do výskytu '}'. Do textového pole se text zadává manuálním zápisem, kopírováním z okna jiné aplikace pomocí schránky nebo načtením souboru (položka menu File>Open). Trans-množina se předpokládá v následující syntaxi.

```
trans-množina  :: t-přiřazení
               |  trans-množina1 ',' t-přiřazení

t-přiřazení    :: identifikátor '[' výraz ']'
```



Obrázek 6.2. Okno aplikace s kartou pro vložení trans-množiny

```
výraz          :: literál
               |  '!' výraz1
               |  výraz1 '.' výraz2
               |  výraz1 '+' výraz2
               |  '(' výraz1 ')'
```

kde literál je identifikátor proměnné, '0' nebo '1'.

Samotné načtení se provede po stisku tlačítka 'Next'. Jména proměnných musí splňovat některá pravidla běžná pro identifikátory v programovacích jazycích, například nesmějí začínat číslicí a nesmějí obsahovat většinu zvláštních znaků. Výjimku tvoří znak '@', kterým začínají identifikátory registrů APLC (např. @f). T-přiřazení pro registry se z trans-množiny odstraní, protože se jedná o pomocné proměnné, které se v každém cyklu APLC nastavují na pevně danou hodnotu (@f=1) instrukcemi `end` a `init` a nemohou tedy popisovat přechodovou funkci automatu. Identifikátory nesmějí být shodné s žádným rezervovaným slovem nástrojů SMV a UPPAAL, navíc nesmějí kolidovat se slovy a prefixy rezervovanými pro tvorbu modelu (např. se jménem SMV modulu modelujícího automat).

Pokud bylo načtení trans-množiny úspěšné, program přejde na další kartu. Při neúspěchu se objeví okno s chybovou hláškou a kurzor zpravidla přejde na místo detekce chyby.

6.1.2 Přidání časovačů

Kapitola 5.3 popisuje rozšíření modelu v UPPAALu o časovače. Přidáním časovačů před kroky nastavení typů proměnných a počátečních hodnot můžeme proměnné představující vstupy a výstupy časovačů z těchto kroků vyjmout.

Na příslušné kartě lze přidat časovač stiskem tlačítka 'Add' a vyplněním parametrů časovače na novém okně, které se otevře pro tento účel. Pro výběr vstupu a výstupů časovače okno obsahuje ovládací prvky, které umožňují vybrat pouze proměnné, které mohou symbolizovat vstupy nebo výstupy. Jako vstup lze vybrat pouze proměnnou, která se vyskytuje v codomain trans-množiny a není v její domain, protože zápis do ní pouze zastupuje instrukci spouštění časovače, její hodnota není nikde v APLC programu čtena. Pro výstup lze naopak využít

proměnnou z domain a nepřítomnou v codomain, protože její hodnotu určuje stav časovače, ne instrukce APLC programu, lze ji tedy jenom číst.

Seznamy proměnných v ovládacích prvcích se postupně aktualizují tak, aby například nebylo možné použít jednu proměnnou jako výstup dvou časovačů.

Volba 'Call before output use' způsobí modelování časovače spouštěného před prvním čtením jeho výstupu pro určení jiné proměnné než vlastního vstupu (kap. 5.3, str. 51).

6.1.3 Určení programové inicializační metody

V dalším kroku se určuje, za se bude modelovat některá inicializační procedura podle kapitol 4.5 nebo 5.2 a pokud ano, zadají se její parametry. Nabízejí se tři možnosti,

- nemodelovat žádnou inicializační proceduru, pak postačí na příslušné kartě ponechat volbu přepínače 'No initiation' a přejít na další krok,
- modelovat inicializaci pomocí proměnné indikující první scan cyklus volbou druhého přepínače a výběrem indikační proměnné ze seznamu proměnných, které jsou v domain trans-množiny, nejsou v její codomain a nebyly vybrány jako vstup nebo výstup některého časovače,
- inicializační program přeložený do trans-množiny a vložený do textového pole stejným způsobem jako trans-množina hlavního programu. V inicializačním programu (v jeho domain a codomain) se nesmějí vyskytovat proměnné použité jako vstupy nebo výstupy časovačů, modelování předpokládá, že k nim inicializační program nepřistupuje.

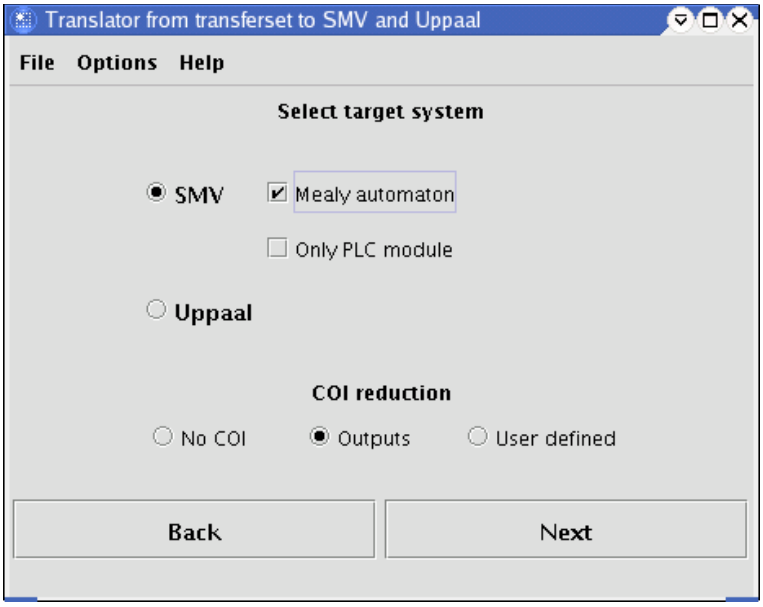
6.1.4 Korekce určení typu proměnných

Další karta aplikace obsahuje seznam proměnných, které se vyskytují v trans-množině hlavního programu nebo v trans-množině inicializačního programu (samozřejmě po odstranění t-přiřazení pro registry APLC – viz kap. 6.1.1). Ze seznamu jsou odstraněny proměnné symbolizující vstupy a výstupy časovačů a proměnná indikující první scan cyklus, pokud byla na předchozí kartě vybrána. Každá proměnná v seznamu má přepínač určující, zda se jedná o vstupní, výstupní nebo vnitřní proměnnou, uživatel u každé vybere jednu variantu.

Programy zpravidla nezapisují do obrazů vstupních proměnných a naopak použití vnitřní proměnné, do které se nezapisuje může znamenat snad jen použití konstant uložených v paměti. Lze proto předpokládat, že proměnné, které se vyskytují v domain trans-množiny programu a nejsou v její codomain, jsou vstupy, ostatní proměnné výstupy či vnitřními proměnnými. Pro předpokládané vstupy se automaticky nastaví přepínač do polohy pro vstupy, pro ostatní proměnné do polohy pro vnitřní proměnné. Uživatel tedy

- provede korekci výběru u vstupů, do jejichž obrazů program zapisuje,
- provede korekci u vnitřních proměnných a výstupů, do nichž program nezapisuje,
- vybere výstupy.

Pro velkou část proměnných nemusí odhadnuté nastavení měnit.



Obrázek 6.3. karta pro určení výstupního formátu a nastavení parametrů COI redukce

6.1.5 Nastavení počátečních hodnot

Každá vnitřní proměnná a obraz výstupu může mít nastavenou počáteční hodnotu. Nastavení může vzniknout například v případě, kdy použité PLC, jehož program ověřujeme, automaticky nuluje před spuštěním programu celý obsah paměti proměnných – jako počáteční hodnota všech binárních proměnných se nastaví 0. Počáteční hodnoty mohou také pocházet z inicializační procedury, která byla ověřena zvlášť a v modelu ji nepopisujeme.

Pro vnitřní a výstupní proměnné jsou na kartě nastavení počátečních hodnot připraveny přepínače určující jejich počáteční hodnoty. Každý přepínač nabízí tři varianty: 0, X, 1, kde X znamená, že proměnná může při spuštění programu nabývat libovolné hodnoty.

6.1.6 Zjištění množiny použitých proměnných

Model automatu může obsahovat všechny vnitřní i výstupní proměnné APLC programu, ze kterého vychází. Některé proměnné mohou být v modelu definovány jako skutečné proměnné použitého verifikačního nástroje, proměnné s konstantní hodnotou se mohou definovat jako konstanty v UPPAALu nebo symboly v SMV, při modelování Mealy automatu v SMVmůžeme dokonce i některé proměnné definovat pouze symboly. Přes uvedená zjednodušení může být výsledný model stále zbytečně veliký. Pokud chceme verifikovat formuli, ve které figuruje jen část z použitých proměnných, postačí modelovat proměnné a časovače, které zbudou po aplikaci COI redukce (str. 20).

Pro výběr, zda použít COI redukci a na které proměnné, leží na kartě zobrazené na obr. 6.3 přepínač s volbami pro

- žádnou redukci,
- redukci na výstupní proměnné a všechny proměnné a časovače, které na ně mají vliv,

- redukci na uživatelsky zadané proměnné a časovače a všechny elementy, které na ně mají vliv.

Při zvolení třetí možnosti následuje po stisku 'Next' ještě karta se seznamem všech proměnných a časovačů. Výběrem z tohoto seznamu se určí elementy, které musí výsledný model obsahovat.

Tato práce se nezabývá spojováním modelu automatu, s modelem řízeného procesu a jejich případnou společnou verifikací. Lze ale prohlásit, že model řízeného procesu přidá závislosti výstup→vstup, které použitá redukce neuvažuje. K použití COI redukce je v takovém případě třeba přistupovat obezřetně – buď ji provést jinými prostředky nad celým modelem nebo redukovat pouze model automatu, ale do nutných proměnných zahrnout všechny vstupy a výstupy vedoucí z/do modelované části řízeného procesu.

6.1.7 Vytvoření modelu

Předchozí kroky nastavily většinu parametrů modelu. Zbylé se mohou měnit v okně pro nastavení modelovacích voleb (menu Options>Modelling options) popsaných v kap. 6.2.1, jde o časové parametry scan cyklu v UPPAAL modelu a způsob modelování inicializačních procedur. Stiskem tlačítka 'Next' na kartě pro výběr výstupního formátu, případně na kartě pro výběr proměnných COI redukce, se vytvoří samotný model.

Zda bude výsledný model v SMV nebo v UPPAALu, určuje výběr příslušného přepínače. Při modelování v SMV lze navíc zvolit model automatu typu Mealy či Moore a to, jestli se ve výstupu vytvoří pouze model automatu jako jeden SMV modul nebo i modul main obsahující definice vstupních proměnných. Jestliže byly zadány časovače, není volba SMV přístupná, lze vytvořit pouze model v UPPAALu.

6.2 Možnosti nastavení

V aplikaci je možné nastavit některé parametry modelu a nastavení ovlivňující její běh položkami menu Options.

6.2.1 Modelovací volby

Menu Options>Modelling options otevře okno pro nastavení modelovacích parametrů, které nejsou dodávány žádným z výše zmíněných kroků. Jedná se o určení způsobu modelování inicializační procedury a nastavení parametrů synchronizačního procesu v UPPAALu.

Kapitoly 4.5 a 5.2 popisují modelování dvou inicializačních procedur, obě lze převést na popis přechodové funkce oddělenými výrazy i na popis jedním výrazem a proměnnou indikující první stav automatu. Na okně modelovacích voleb lze určit, která varianta se použije. Existuje i volba automatického použití přirozeného popisu inicializační metody – oddělenými výrazy pro inicializační program a společným výrazem pro inicializaci indikační proměnnou.

Dalšími parametry, které se nevolí na kartách okna aplikace, jsou vlastnosti synchronizačního procesu v UPPAAL modelu. Jsou to

- minimální a maximální délka cyklu,
- volba, zda má být nulová doba od synchronizace modelu automatu po synchronizaci změn vstupů (od output scanu po input scan),
- volba, zda se má model neobsahující časovače synchronizovat procesem se všemi místy jako urgentními (nulová doba celého cyklu).

6.2.2 Ostatní nastavení

Měnit lze i některé parametry neovlivňující strukturu výsledného modelu ale především běh samotné aplikace. Menu Options>Settings otevře okno s nastavením některých časových limitů pro náročnější operace a s volbou úrovně minimalizace výrazů.

Převod automatu pracuje s logickými výrazy, načítá je z textového formátu, v některých případech nad nimi provádí logické operace. Nad načítanými výrazy i výsledky operací se může provádět minimalizace výrazů. Automaticky se provádí úpravy typu $0 \cdot x = 0$, $1 \cdot x = x$, $!0 = 1$ apod. Kromě těchto úprav je implementována minimalizace založená na metodě Quine-McCluskey, která vrací minimalizovaný výraz v disjunktní normální formě (DNF). Tato minimalizace je velmi časově náročná, proto je umožněno nevyužívat ji. Tři volby na okně s nastavením mají význam

- při minimalizaci složeného výrazu nejprve minimalizovat jeho vnitřní části,
- minimalizovat DNF až po jejím sestrojení z celého výrazu,
- neprovádět minimalizaci.

Některé činnosti převodní aplikace mohou být časově náročné, například otevírání příliš velkého souboru nebo již zmíněná minimalizace logických výrazů. Aby během nich nedošlo k dlouhému zamrznutí aplikace, bylo jejich provedení umístěno do oddělených výpočetních vláken. Díky oddělení těchto operací hlavní okno nezamrzá a je například možné během nich aplikaci regulérně ukončit nebo v některých případech sledovat výpisy referující o průběhu operace. Kromě toho vlákna po případném vypršení nastaveného časového limitu směřují svou činnost k urychlenému ukončení. V nastavovacím okně je možné zvlášť měnit časové limity pro jednotlivé operace.

Poslední položka menu Options (Reserved words) zpřístupní panel pro změny slov a prefixů využívaných při tvorbě modelu. Změna slova nebo prefixu se může hodit například při jeho kolizi se jménem proměnné, pak místo změny jména proměnné ve vstupní trans-množině můžeme změnit kolidující rezervované slovo nebo prefix. Toto okno je přístupné pouze z první karty, tedy před načtením trans-množiny.

Kapitola 7

Příklady převodu a verifikace

Vytvořený program pro převod automatu do verifikačních nástrojů jsem použil k verifikaci několika PLC programů z důvodu ověření jeho funkčnosti a kvůli srovnání modelů vytvořených různými postupy.

Převod dvou jednoduchých programů obsahuje příloha C, která na nich především předvádí způsob použití převodní aplikace. Zde jsou uvedeny dva příklady, na kterých je sledována hlavně samotná verifikace a její výpočetní nároky.

První příklad ukazuje modelování a verifikaci programu řízení garážových vrat použitého jako příklad v [12], kde se modeluje a verifikuje jako Moore automat, ukazuje možnost připojení modelu řízeného procesu a srovnává se rychlost verifikace s jiným řešením. Zde, v kap. 7.1, se verifikuje model vytvořený podle automatu Mealy. Pro stručnost se tu nachází verifikace pouze jednoho požadavku.

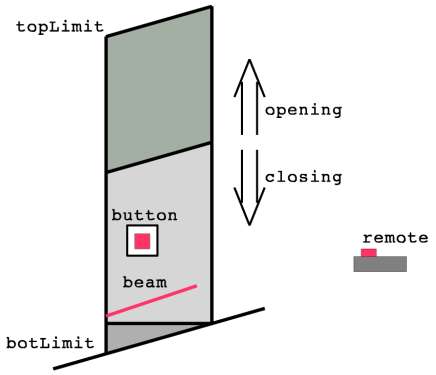
Druhý příklad se zabývá verifikací programu řízení čerpací jednotky. Program obsahuje časovače, proto je modelován v UPPAALu. Zadání příkladu a řídicí program byl převzat z článku [16], kde se k jeho verifikaci přistupuje jiným způsobem a využívá se různých abstrakcí původního programu. Příklad obsahuje srovnání rychlosti verifikace celého programu s verifikací modelů vzniklých abstrakcí na základě inspekce struktury programu.

7.1 Řízení garážových vrat

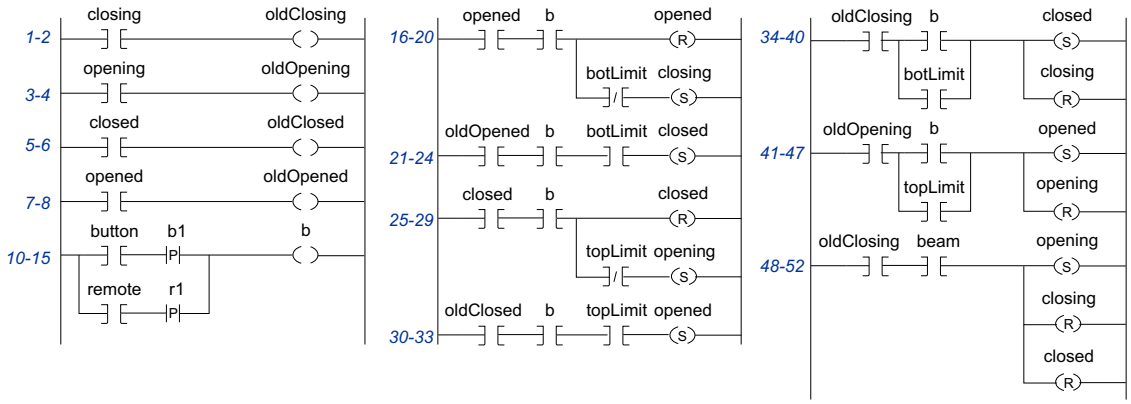
Navrhněme řízení garážových vrat podle následující specifikace.

- V garáži je jedno tlačítko a jedno tlačítko je na dálkovém ovládání,
- po stisku tlačítka se vrata 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,
- dolní částí prostoru vrat probíhá světelný paprsek, který detekuje přítomnost předmětu, je-li přerušen během zavírání dveří, pohyb se změní na druhý směr.

Situaci znázorňuje obr. 7.1.



Obrázek 7.1. Garážová vrata



Obrázek 7.2. LD program pro řízení garážových vrat

Pro zápis programu i jeho verifikaci zavedeme následující pojmenování vstupních a výstupních 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.

Na obr. 7.2 se nachází řídicí program v podobě žebříčkového diagramu, na obr. 7.3 potom jeho zápis v APLC instrukcích. Čísla u příček žebříčkového diagramu odpovídají příslušným řádkům IL programu.

Při použití správně navrženého programu by nikdy nemělo dojít k uváznutí (deadlocku), při běhu programu se vždy jeho stav může vyvíjet. Tento požadavek můžeme vztáhnout na stavové proměnné. Požadavek možnosti změny proměnné *closing* kdykoliv v budoucnosti vyjadřuje CTL formule

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

čili kdykoliv existuje způsob, jak může *closing* nabýt log. hodnoty 1 i 0.

SMV model programu se zápisem formule je v příloze B. Uvedená formule není pravdivá. Jak ukazuje následující výpis, k uváznutí $closing$ dojde po načtení vstupů $topLimit = 1$ a

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

Obrázek 7.3. IL program pro řízení garážových vrat

$botLimit = 1$ (současném sepnutí obou koncových spínačů), ke kterému by nemělo nikdy dojít. Ve stavu 1.2 ve výpisu jsou $closed = closing = opened = opening = 0$, při pohledu na program v LD můžeme vidět, že pak všechny příčky diagramu kromě „10–15“ musejí mít nulovou hodnotu, nebude se moci provést žádná z instrukcí nastavujících hodnoty uvedených proměnných.

```
-- specification AG (EF plc.closing & EF (!plc.closing)) is false
-- as demonstrated by the following execution sequence
```

```
-> State: 1.1 <-
  beam = 0
  botLimit = 0
  button = 0
  remote = 0
  topLimit = 0
  plc.M_b1 = 0
  plc.M_closed = 0
  plc.M_closing = 0
  plc.M_opened = 1
  plc.M_opening = 0
  plc.M_r1 = 0
  plc.r1 = 0
  plc.opening = 0
  plc.opened = 1
  plc.oldOpening = 0
  plc.oldOpened = 1
  plc.oldClosing = 0
  plc.oldClosed = 0
  plc.closing = 0
  plc.closed = 0
  plc.b1 = 0
  plc.b = 0
-> State: 1.2 <-
  botLimit = 1
  button = 1
  topLimit = 1
  plc.opened = 0
  plc.b1 = 1
  plc.b = 1
```

Při omezení možných hodnot vstupů zápisem `TRANS !(topLimit & botLimit)` již formule byla pravdivá:

```
-- specification AG (EF plc.closing & EF (!plc.closing)) is true
```

	NuSMV	-dynamic[1]
log. rovnice	0,13	0,13
instrukce	210	2,9

Tabulka 7.1. Srovnání rychlosti verifikace modelů programu řízení vrat

Program by byl z hlediska verifikované formule správný, pokud bychom se spolehli na předpokládanou funkci koncových spínačů. Nebude odolný proti rušivým vlivům, např. náhodnému krátkému sepnutí jednoho ze spínačů cizím předmětem, který se dostane do jeho prostoru.

Program můžeme modelovat i jinými způsoby, nejen jeho převodem na soustavu logických rovnic. Tab. 7.1 obsahuje srovnání rychlostí verifikace modelu získaného z logických rovnic s modelem vytvořeným přímým modelováním programu podle popisu v [6].

Výhodou modelu automatu popsaného logickými rovnicemi je vyšší rychlost verifikace a přehledný výpis stavů v protipříkladu formule. Model získaný přímým modelováním instrukcí je složitější, obsahuje pomocné proměnné pro zásobník a pro čítač instrukcí. Verifikace tohoto modelu je náročnější a výpis protipříkladu obsahuje posloupnost stavů programu po jednotlivých instrukcích, je tedy mnohem delší, ale na druhou stranu poskytuje více informací. Výhodou přímého modelování oproti implementovanému modelování logických rovnic je možnost využít širších programovacích instrukcí jako práce s číselnými proměnnými nebo použití zpětných skoků. Model získaný přímým modelováním a příslušný výpis protipříkladu zde není pro svou délku uveden. Model se nachází na přiloženém CD.

7.2 Řízení čerpací jednotky

7.2.1 Verifikovaný program

Pro další příklad převodu a verifikace použiji program řízení čerpací jednotky se strukturou podle obr. 7.4 převzatý z [16]. Jednotka dodává vodu z nádrže pomocí dvou čerpadel. Ke každému čerpadlu je připojen přívodní a odtokový ventil, po jednom ventilu mají výtoková větev potrubí a větev zpětného toku.

Na řídicí systém jsou kladeny následující bezpečnostní požadavky.

P1 Přítokový ventil čerpadla musí být otevřen alespoň 5 s před spuštěním čerpadla.

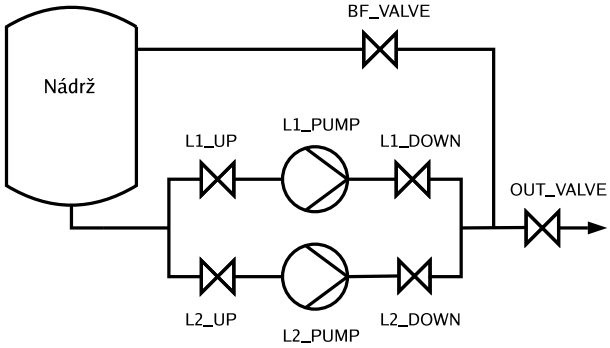
P2 Obě čerpací linky nesmí být nikdy spuštěné zároveň.

P3 Když je zastaven provoz, musí být všechny akční členy vypnuty (čerpadla zastavená, ventily zavřené).

P4 V případě poruchy čerpadla budou všechny související akční členy vypnuty.

Program přistupuje k řízenému procesu pomocí vstupů a výstupů PLC. V obr. 7.4 jsou u jednotlivých akčních členů zaznamenána jména proměnných zastupující příslušné výstupy. Logická hodnota 1 na výstupu pro ventil znamená otevření ventilu, jednička na výstupu pro

[1] Při verifikaci se ukázalo výhodné zapnout dynamické přeuspořádávání OBDD



Obrázek 7.4. Schéma zapojení čerpadel

čerpadlo jeho spuštění. Program využívá řadu vstupů, patří k nim např. signalizace velikosti požadavku na odběr (L_FLOW, H_FLOW), signalizace poruchy čerpadla (L*i*_FAIL) nebo zastavení provozu (SP_FAIL).

Článek [16] ukazuje verifikaci řídicího programu zapsaného v žebříčkovém diagramu. Zde uvádím pouze jeho přepis do APLC programu. Bloky kódu na obr. 7.5 označené čísly odpovídají jednotlivým částem žebříčkového diagramu. Vstupy časovačů jsou nahrazeny proměnnými TON_L*i*_in.

7.2.2 Modelování programu a zápis požadavků

Program můžeme přeložit pomocí APLCTRANSu a poté ho modelovat postupem popsaným v předchozích kapitolách jako model v nástroji UPPAAL. Formule *P2–4* lze v UPPAALu zapsat

```
0  Load line_swap
   REdge osr1
   Push
   And !L1_PRIO
   Store ACT_L1
   Pop
   And L1_PRIO
   Store ACT_L2

1  Load ACT_L1
   Or L1_PRIO
   And !ACT_L2
   Store L1_PRIO

2  Load L1_PRIO
   And !L2_PROD
   Or L2_FAILURE
   Store RS_L1_S

3  Load !H_FLOW
   And !L_FLOW
   Or L1_FAIL
   Or SP_FAIL
   Store RS_L1_R1

4  Load RS_L1_S
   Or L1_PROD
   And !RS_L1_R1
   Store L1_PROD

5  Load !L1_PRIO
   Or L1_FAILURE
   Store RS_L2_S

6  Load !H_FLOW
   And !L_FLOW
   Or L2_FAIL
   Or SP_FAIL
   Or L1_PROD
   Store RS_L2_R1

7  Load RS_L2_S
   Or L2_PROD
   And !RS_L2_R1
   Store L2_PROD

8  Load L1_PROD
   Store L1_UP

9  Load L1_PROD
   Store TON_L1_in

10 Load TON_L1_dn
   Store L1_PUMP
   Store L1_DOWN

11 Load L2_PROD
   Store L2_UP

12 Load L2_PROD
   Store TON_L2_in

13 Load TON_L2_dn
   Store L2_PUMP
   Store L2_DOWN

14 Load L1_PROD
   Or L2_PROD
   Store OUT_VALVE

15 Load L1_PUMP
   Or L2_PUMP
   And !H_FLOW
   Store BF_VALVE
   End
```

Obrázek 7.5. Program řízení čerpací jednotky zapsaný v instrukcích APLC

P1	`A[] !(proc_cycle.testing and (_proc_L1_UP.time<5000 or !L1_UP) and L1_PUMP)`
P2	`A[] !(proc_cycle.testing and L1_UP and L1_PUMP and L1_DOWN and L2_UP and L2_PUMP and L2_DOWN)`
P3	`A[] (proc_cycle.testing and SP_FAIL) imply (!L1_PUMP and !L1_UP and !L1_DOWN and !L2_PUMP and !L2_UP and !L2_DOWN and !OUT_VALVE and !BF_VALVE)`
P4	`A[] (proc_cycle.testing and L1_FAIL) imply (!L1_PUMP and !L1_UP and !L1_DOWN)`

Tabulka 7.2. Požadavky jako formule pro UPPAAL

přímo – kromě proměnných vázajících se k požadavku se ve formuli bude vyskytovat pouze proměnná indikující přítomnost synchronizačního procesu v místě po provedení programu (viz kap. 5.1).

Požadavek *P1* nelze zapsat přímo. K jeho ověření je třeba přidat do modelu hodiny, které budou měřit čas od otevření přítokového ventilu. Nové hodiny mohou být nulovány v každém cyklu při L*i*_UP=0 a do formule se pak vloží požadavek, že L*i*_PUMP nesmí být roven 1 pokud je hodnota hodin menší než 5000 – tuto variantu používá [16]. Jinou možností, která nevyžaduje nulování hodin v každém cyklu, je provedení jejich resetu pouze na hraně vedoucí z místa symbolizujícího logickou 0 proměnné L*i*_UP do místa pro 1 v procesu příslušejícím této proměnné. Požadavek potom pro čerpadlo L1 vyjadřuje formule pro *P1* z tab. 7.2. V tabulce se nacházejí specifikace dalších požadavků – *P4* také pouze pro L1.

V [16] autoři konstatují, že se jim z důvodů vysoké paměťové náročnosti nepodařilo provést verifikaci celého programu modelovaného jejich nástrojem. Proto používají abstrakci založenou na odstranění některých příček z programu před jeho modelováním.

Požadavek *P1* závisí na příčkách 8–10. Pokud z programu ostatní příčky odstraníme a budeme proměnnou L1_PROD uvažovat jako vstup s náhodně se měnící hodnotou, ověříme modelovanou část programu proti všem možným sekvencím této proměnné, tedy i proti těm, které mohou nastat v celém programu.

Požadavky *P2–4* nezávisejí přímo na čase, [16] ukazuje postup, jak je lze ověřit při odebrání časovačů: Příčky 9–10 a 12–13 nahradíme přímým zápisem hodnoty proměnné L1_PROD do výstupů L1_PUMP a L1_DOWN resp. zápisem L2_PROD do výstupů L2_PUMP a L2_DOWN.

7.2.3 Výsledky verifikace

Použitý program splňuje všechny formule z tab. 7.2. K tomuto výsledku dospěla verifikace popsaná v [16], tak i verifikace, kterou jsem provedl na modelech získaných postupy z předchozích kapitol.

Časová náročnost verifikace jednotlivých formulí na různých modelech se výrazně liší, jak ukazuje tab. 7.3. Její sloupce odpovídají různým postupům modelování. Pravý, výrazněji oddělený, sloupec obsahuje údaje předložené článkem [16] jako doby verifikací na mo-

delech získaných abstrakcemi, které odebírají některé příčky žebříčkového diagramu (viz předch. podkapitolu). Tato verifikace proběhla na jiném počítači (Athlon $1,6$ GHz, 2 GB RAM) než verifikace modelů, které jsem vytvořil a verifikoval (K6-III 450 MHz, 196 MB RAM). Všechny modely, které popisují časové chování cyklu, předpokládají jeho pevnou délku 20 ms.

M1 obsahuje časy verifikace modelů získaných z celého programu jeho překladem na transmnožinu. Modely byly redukovány COI redukcí pro proměnné obsažené ve verifikované formuli.

M2 představuje modely využívající abstrakcí podle [16] bez použití COI redukce.

M3 $\sim$ M2, ale s COI redukcí.

M4 $\sim$ M3, ale modely nepopisují časové trvání cyklu, doba celého jejich cyklu je nulová.

SMV obsahuje časy verifikace formulí na modelech získaných použitím stejných abstrakcí jako v M2. Tentokrát ale modely byly vytvořené pro SMV jako automaty typu Mealy a verifikovány pomocí NuSMV. Pro modely automatů Moore byly časy průměrně o $0,03$–$0,04$ sekundy delší, rozdíl se však snížil při použití COI redukce.

Časy v tabulce jsou ve formátu „minuty ':' sekundy ',' neceločíselná část sekundy".

Tabulka ukazuje, že při pouhém použití modelovacího postupu bez abstrakce provedené uživatelem trvá verifikace mnohonásobně delší dobu než při zjednodušení programu před jeho modelováním. Přesto je možné tento výsledek považovat za úspěch, protože se narozdíl od [16] vůbec podařilo dosáhnout výsledku bez těchto abstrakcí.

Při použití převzatých abstrakcí programu se čas verifikace snížil. Modely získané po jejich použití a bez COI redukce (sloupec M2) by měly být z pohledu proměnných ekvivalentní s modely verifikovanými autory [16]. Podle hodnot ve sloupci a s vědomím, že jsem použil méně výkonný počítač, by se zdálo, že modely vytvořené převodem soustavy logických rovnic kladou nižší výpočetní nároky. Měl jsem ale k dispozici určitě novější a možná i efektivnější verzi UPPAALu (3.4.7), ke srovnání by bylo vhodnější provést verifikaci obou modelů za stejných podmínek.

Sloupec M3 ukazuje vylepšení časů při použití COI redukce. Rychlost verifikace se dále zvýšila u programů bez použití časovačů a bez potřeby čas sledovat tím, že se vytvořil model s nulovou délkou cyklu (M4).

Výrazně nejmenších časových nároků verifikace dosáhly nečasované modely pro SMV.

	M1	M2	M3	M4	SMV	ZRK
P1	43:55	0,32	0,3	–	–	0:01
P2	20:11	37,2	11,4	4,8	0,09	1:08
P3	22:36	37,3	13,6	5,75	0,1	1:32
P4	8:14	37,2	9,0	4,16	0,09	1:16

Tabulka 7.3. Srovnání dob verifikace na různých modelech

Kapitola 8

Závěr

Navrhnul jsem a implementoval převod automatu popsaného logickými rovnicemi do nástrojů SMV a UPPAAL. Vytvořený program usnadňuje využití dříve navrženého a implementovaného algoritmu APLCTRANS [14] k verifikaci PLC programů. Docílený postup verifikace se alespoň zčásti přibližuje ideálnímu stavu, kdy by se verifikace stala plně zautomatizovanou činností vyžadující minimum zásahů uživatele.

Postup převodu byl úspěšně použit k převodu a verifikaci několika krátkých PLC programů většinou uměle vytvořených jako příklady nebo vzniklých zjednodušením skutečných programů. Výpočetní nároky verifikace získaných modelů se značně lišily podle použitých parametrů převodu a většinou byly podobné či výrazně menší než nároky verifikace modelů získaných použitím postupů modelujících program přímo bez jeho abstrakce jako soustavy rovnic. Mezi výhody použitého řešení patří možnost modelovat program téměř stejným postupem ve dvou odlišných nástrojích. UPPAAL dokáže narozdíl od SMV modelovat a verifikovat časové charakteristiky systémů, v SMV je zase možné vyjádřit širší spektrum nečasových požadavků plnou CTL logikou.

Srovnání rychlostí verifikace v příkladu 7.1 vyznívá pro model získaný překladem soustavy logických rovnic lépe než pro přímé modelování programu. Srovnání časů verifikace vytvořených UPPAAL modelů z příkladu 7.2 s časy uváděnými autory [16] neukazuje průkazně dosažení lepších vlastností modelu, k důkladnému srovnání by bylo zapotřebí spustit verifikaci za zcela stejných podmínek. Z naměřených časů ale jasně vyplývá výhoda možnosti modelovat nečasový automat v SMV, s nímž bylo dosaženo výrazně rychlejší verifikace než v UPPAALu.

Výsledkem práce tedy je použitelná převodní aplikace s některými dobrými vlastnostmi, ale její kvality a využitelnost by se mohly rozšířit realizací dalších možností.

Implementovaný převod do UPPAALu neumožňuje využití všech třech popsaných způsobů modelování automatu (str. 46). Použil jsem pouze variantu využívající rozdělení automatu na procesy pro jednotlivé proměnné, kde každý proces obsahuje alespoň dvě místa odpovídající binárním hodnotám proměnné. Vytváření společného procesu jsem neimplementoval, protože jednotlivé rovnice pro stavové proměnné popisují přímo jednotlivé procesy, vytvoření společného procesu by vyžadovalo jejich slučování. Oddělená místa pro různé hodnoty proměnných jsem zvolil kvůli vizuálnímu sledování hodnot proměnných a jejich změn v okně simulátoru UPPAALu. Možná by bylo užitečné provést důkladné srovnání všech tří

způsobů, především z hlediska rychlosti jejich verifikace.

Aplikace vytvořená pro převod automatu vyžaduje od uživatele postupné zadávání parametrů. Při opakovaném modelování stejného programu nebo jeho variant by bylo vhodné uložit například informace o typu jednotlivých proměnných, jejich počátečních hodnotách, či časovačích a při překladu je načíst a nemuset je znovu zadávat. Nynější implementace usnadňuje pouze vytvoření několika modelů se stejnou trans-množinou, kdy je možné se vracet k předchozím krokům pomocí tlačítka 'Back'.

Využití modelování automatu popsaného logickými rovnicemi by usnadnila jeho alespoň částečná integrace či propojení s implementací algoritmu APLCTRANS, aby nebylo nutné mezi oběma programy manuálně kopírovat trans-množinu v textové formě.

Literatura

[1] ALUR, R. – COURCOUBETIS, C. – DILL, D. L. Model-Checking in Dense Real-time. In *Information and Computation*, vol. 104, No. 1, pp. 2–34. 1993.

[2] BEHRMANN, G. et al. UPPAAL Implementation Secrets. In *Proc. of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*. 2002.

[3] BEHRMANN, G. – DAVID, A. – LARSEN, K. G. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, pp. 200–236. Springer–Verlag, 2004.

[4] BEREZIN, S. *Model Checking and Theorem Proving: a Unified Framework*. PhD Thesis, Carnegie Mellon University, 2002.

[5] BIERE, A. et al. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Proceedings of Tools and Algorithms for the 107 Construction and Analysis of Systems*, LNCS 1579. Springer-Verlag, 1999.

[6] 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.

[7] HUTH, M. – RYAN M. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000. ISBN 0-521-65602-8.

[8] LARSEN, K. G. – PETTERSSON, P. – YI, W. UPPAAL in a Nutshell. In *Int. Journal on Software Tools for Technology Transfer*, vol. 1, No. 1–2, pp. 134–152. Springer–Verlag, 1997.

[9] MADER, A. A classification of PLC models and applications. In *WODES 2000: 5th Int. Workshop on Discrete Event Systems*, pp. 239–247. Gent, Belgium: 2000. ISBN 0-7923-7897-0.

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

[11] RAUSCH, M. – KROGH, B. H. Formal Verification of PLC Programs. In *Proc. of American Control Conference*. PA, USA: June 1998.

[12] Šprdlík, O. – Šusta, R. SMV Verification of PLC Programs. In *Proceedings of the 6th International Scientific-Technical Conference on Process Control* (ŘÍP 2004). Pardubice: University of Pardubice, 2004, p. 187. ISBN 80-7194-662-1.

[13] Šusta, R. Paralel Abstraction of PLC Program. In *Proceedings of the 5th International Scientific-Technical Conference on Process Control* (ŘÍP 2002). Pardubice: University of Pardubice, 2002, pp. 9–12. ISBN 80-7149-452-1.

[14] Šusta, R. *Verification of PLC programs*. PhD thesis, Revised edition. Czech Technical University in Prague, Department of Cybernetics, 2003.

[15] Willems, H. X. *Compact timed automata for PLC programs*. Tech. report CSI–R9925, University of Nijmegen, 1999.

[16] Zoubek, B. – Roussel, J.-M. – Kwiatkowska, M. Towards automatic verification of ladder logic programs. In *IMACS Multiconference on Computational Engineering in Systems Applications (CESA)*. 2003. ISBN 2-9512309-5-8.

Internetové zdroje

[17] *The SMV System* [online]. Model Checking Group at Carnegie Mellon University. ⟨`http://www-2.cs.cmu.edu/~modelcheck/smv.html`⟩

[18] *NuSMV home page* [online]. Trentino Cultural Institute, Center for Scientific and Technological Research. ⟨`http://nusmv.irst.itc.it/`⟩

[19] McMillan, K. *The SMV Model Checker*. [online]. ⟨`http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`⟩

[20] *UPPAAL* [online]. ⟨`http://www.uppaal.com`⟩, [cit. 17. 12. 2004].

[21] *Uppaal Timed Automata Parser Library* [online]. ⟨`http://www.cs.aau.dk/~behrmann/utap/`⟩, [cit. 18. 12. 2004].

[22] Sun Microsystems *Java Technology* [online]. ⟨`http://java.sun.com/`⟩

Použité symboly a zkratky

$\hat{x}$	t-přiřazení
$\widehat{C}$	trans-množina
$\uparrow$	rozšíření
$\widehat{\in}$	relace přítomnosti t-přiřazení pro proměnou v trans-množině
$\widehat{\cap}$	průnik trans-množiny s množinou proměnných
$\models$	relace splnění formule na modelu a stavu
$\alpha(X)$	abeceda generovaná množinou X
APLC	Abstraktní PLC
BMC	Bounded Model Checking
$BPLC$	Binární PLC
CDD	Clock Difference Diagram
co	codomain
$\mathrm{co_p}$	PLC codomain
COI	Cone Of Influence (reduction)
CTL	Computation Tree Logic
dom	domain
$\mathrm{dom_p}$	PLC domain
DBM	Diference Bounded Matrix
DNF	Disjunktní normální forma
DTD	Document Type Definition, definice typu dokumentu
IL	Instruction List
JRE	Java Runtime Environment
$\mathrm{map_p}$	mapování
$\mathbb{N}$	množina přirozených čísel
OBDD	Ordered Binary Decision Diagram
PLC	Programmable Logic Controller, programovatelný logický automat
$\mathbb{R}$	množina reálných čísel
SMV	Symbolic Model Verifier – verifikační nástroj
TCTL	Timed CTL
TON	časovač 'on delay'
TOFF	časovač 'off delay'
UPPAAL	verifikační nástroj vyvinutý univerzitami v Uppsale a Aalborgu
XML	eXtended Markup Language

Seznam obrázků

1.1 Kroky verifikace PLC programu pomocí převodu na logické rovnice 6

2.1 Scan cyklus PLC 8
2.2 Odezva časovačů on delay a off delay 9
2.3 Program APLC, výsledek překladu, graf přechodů automatu 15

3.1 Různé OBDD reprezentující stejnou funkci 21
3.2 Jednoduchý model a reprezentace jeho stavů vektory binárních hodnot 23
3.3 Algoritmus splnitelnosti v časovém automatu 25
3.4 Množina časových omezení vyjádřená různými prostředky 26
3.5 Ukázka modelu v SMV 27
3.6 Inicializace, nedeterministické výrazy, definice symbolu 27
3.7 Model rozložený na více modulů 28
3.8 Souběžný a prokládaný režim 29
3.9 Výsledek verifikace modelu 30
3.10 Časový automat v UPPAALu 31
3.11 Pohled na model složený z několika procesů 32

4.1 Model dvoustavového Mealy automatu 36
4.2 Mealyho automat a jeho SMV model 38
4.3 Příklad dosazení a odebrání konstant 39
4.4 Model automatu typu Moore v SMV 40
4.5 Modely automatů typu Mealy a Moore získaných ze stejného programu . . . 41
4.6 Kompozice modelu automatu a vstupu 42

5.1 Graf přechodů Moore automatu a jeho zobrazení v UPPAALu 44
5.2 Model náhodně se měnícího vstupu 45
5.3 Náhodná počáteční hodnota, nulová doba mezi přechody 47
5.4 Model s nenulovou dobou mezi přechody 48
5.5 Proces neinicializované proměnné 49
5.6 Dva způsoby modelování programové inicializace 49
5.7 Časové automaty pro časovače 50
5.8 Modely časovačů TON (vlevo) a TOFF (vpravo) v UPPAALu 50
5.9 Průběhy po změně vstupu časovačů 51
5.10 Synchronizační proces pro časovače spouštěné před čtením jejich výstupu . . 52

6.1 Kroky nastavení parametrů pro překlad automatu 54
6.2 Okno aplikace s kartou pro vložení trans-množiny 55
6.3 karta pro určení výstupního formátu a nastavení parametrů COI redukce . . . 57

7.1 Garážová vrata 61
7.2 LD program pro řízení garážových vrat 61
7.3 IL program pro řízení garážových vrat 62
7.4 Schéma zapojení čerpadel 64
7.5 Program řízení čerpací jednotky zapsaný v instrukcích APLC 64

Seznam tabulek

3.1 Formule verifikovatelné v UPPAALu 33

7.1 Srovnání rychlosti verifikace modelů programu řízení vrat 63
7.2 Požadavky jako formule pro UPPAAL 65
7.3 Srovnání dob verifikace na různých modelech 66

Příloha A

DTD souborů XML pro Uppaal

```
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.0//EN"
"http://www.docs.uu.se/docs/rtmv/uppaal/xml/flat-1_0.dtd">
```

Obsah definičního souboru flat-1_0.dtd je

```
<!ELEMENT nta (imports?, declaration?, template+, instantiation?, system)>
<!ELEMENT imports (#PCDATA)>
<!ELEMENT declaration (#PCDATA)>
<!ELEMENT template (name, parameter?, declaration?, location*, init?, transition*)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name x   CDATA #IMPLIED
               y   CDATA #IMPLIED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter x   CDATA #IMPLIED
                    y   CDATA #IMPLIED>
<!ELEMENT location (name?, label*, urgent?, committed?)>
<!ATTLIST location id ID #REQUIRED
   x  CDATA #IMPLIED
   y  CDATA #IMPLIED>
<!ELEMENT init EMPTY>
<!ATTLIST init ref IDREF #IMPLIED>
<!ELEMENT urgent EMPTY>
<!ELEMENT committed EMPTY>
<!ELEMENT transition (source, target, label*, nail*)>
<!ATTLIST transition x   CDATA #IMPLIED
                     y   CDATA #IMPLIED>
<!ELEMENT source EMPTY>
<!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target ref IDREF #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ATTLIST label kind CDATA #REQUIRED
x    CDATA #IMPLIED
y    CDATA #IMPLIED>
<!ELEMENT nail EMPTY>
<!ATTLIST nail x   CDATA #REQUIRED
               y   CDATA #REQUIRED>
<!ELEMENT instantiation (#PCDATA)>
<!ELEMENT system (#PCDATA)>
```

Příloha B

SMV model programu řízení vrat

```
MODULE PLC(beam, botLimit, button, remote, topLimit)
VAR
 M_b1 : boolean;
 M_closed : boolean;
 M_closing : boolean;
 M_opened : boolean;
 M_opening : boolean;
 M_r1 : boolean;
ASSIGN
 init(M_b1) := 0;
 init(M_closed) := 0;
 init(M_closing) := 0;
 init(M_opened) := 1;
 init(M_opening) := 0;
 init(M_r1) := 0;
 next(M_b1) := b1;
 next(M_closed) := closed;
 next(M_closing) := closing;
 next(M_opened) := opened;
 next(M_opening) := opening;
 next(M_r1) := r1;
DEFINE
 b :=  remote&!M_r1 | button&!M_b1;
 b1 :=  button;
 closed :=  M_closing&!beam&botLimit | !M_closing&M_r1&M_b1&M_closed
  | !M_closing&!remote&M_b1&M_closed | !M_closing&M_r1&!button&M_closed
  | !M_closing&!remote&!button&M_closed | M_closing&!beam&remote&!M_r1
  | M_closing&!beam&button&!M_b1 | M_closing&!beam&M_closed;
 closing :=  !M_closing&M_opened&remote&!M_r1&!botLimit
  | !M_closing&M_opened&!botLimit&button&!M_b1
  | M_closing&!beam&!remote&!botLimit&M_b1 | M_closing&!beam&M_r1&!botLimit&M_b1
  | M_closing&!beam&!remote&!botLimit&!button
  | M_closing&!beam&M_r1&!botLimit&!button;
 oldClosed :=  M_closed;
 oldClosing :=  M_closing;
```

```
 oldOpened :=  M_opened;
 oldOpening :=  M_opening;
 opened :=  M_opening&topLimit | remote&!M_r1&M_opening | button&!M_b1&M_opening
  | M_opened&!remote&M_b1 | M_opened&M_r1&M_b1 | M_opened&!remote&!button
  | M_opened&M_r1&!button | remote&!M_r1&topLimit&M_closed
  | button&!M_b1&topLimit&M_closed;
 opening :=  M_closing&beam | !remote&M_b1&M_opening&!topLimit
  | M_r1&M_b1&M_opening&!topLimit | !remote&!button&M_opening&!topLimit
  | M_r1&!button&M_opening&!topLimit | remote&!M_r1&!M_opening&!topLimit&M_closed
  | button&!M_b1&!M_opening&!topLimit&M_closed
  | M_opened&remote&!M_r1&botLimit&!M_opening&!topLimit
  | M_opened&botLimit&button&!M_b1&!M_opening&!topLimit;
 r1 :=  remote;


MODULE main
VAR
 beam : boolean;
 botLimit : boolean;
 button : boolean;
 remote : boolean;
 topLimit : boolean;

 plc: PLC(beam, botLimit, button, remote, topLimit);


SPEC
 AG ((EF plc.closing) & (EF !plc.closing))
```

Příloha C

Tutorial

Příloha ukazuje na dvou jednoduchých příkladech postup převodu trans-množiny řídicího PLC programu do vstupních formátů verifikačních nástrojů pomocí transformační aplikace. Příklady nepostihují všechny možnosti převodu, kladou si za cíl názorně ukázat převod v několika variantách.

První příklad převádí program řízení přídavné nádrže do SMV. Druhý příklad ukazuje modelování programu využívajícího časovače do nástroje UPPAAL.

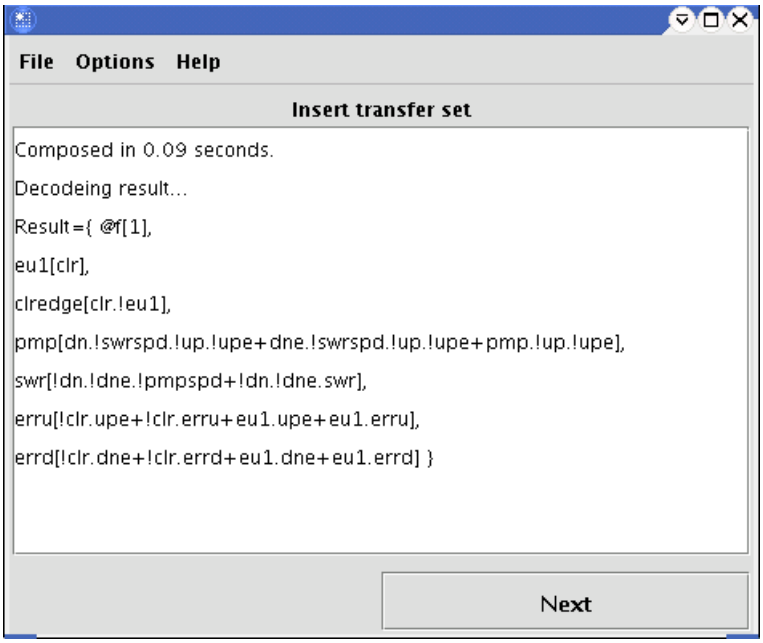
C.1 Přípravná nádrž

Program pro řízení přípravné nádrže převzatý z [14] byl přeložen programem APLCTRANS. Samotný program zde není uveden, vstupem transformačního programu je získaná trans-množina.

```
Composed in 0.09 seconds.
Decodeing result...
Result={ @f[1],
eu1[clr],
clredge[clr.!eu1],
pmp[dn.!swrspd.!up.!upe+dne.!swrspd.!up.!upe+pmp.!up.!upe],
swr[!dn.!dne.!pmpspd+!dn.!dne.swr],
erru[!clr.upe+!clr.erru+eu1.upe+eu1.erru],
errd[!clr.dne+!clr.errd+eu1.dne+eu1.errd] }
```

Vstupy programu jsou `clr`, `dn`, `dne`, `up`, `upe`, `pmpspd` a `swrspd`. Výstupy jsou `pmp`, `swr`, `erru` a `errd`.

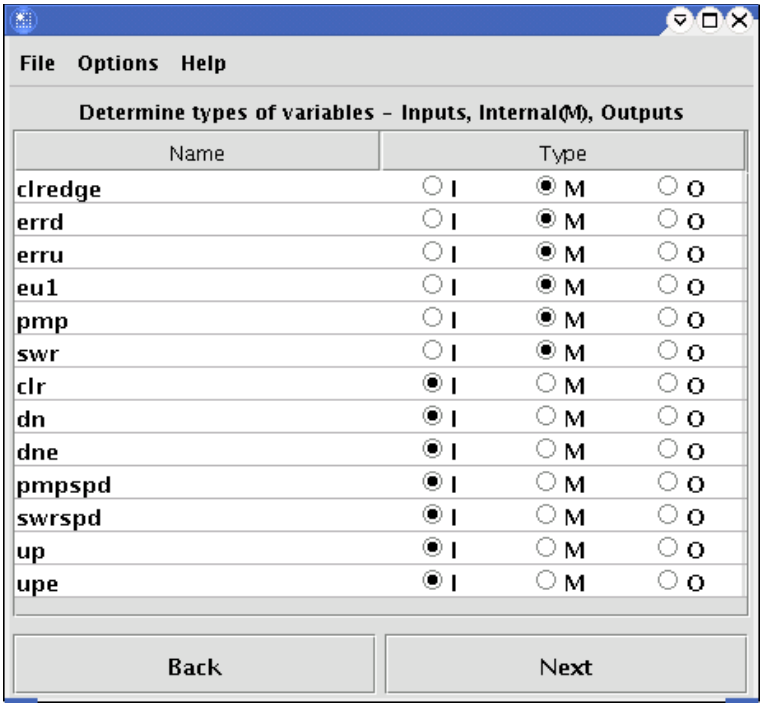
Uvedený kód získaný překladem pomocí APLCTRANS můžete zkopírovat do textové oblasti na první kartě programu.



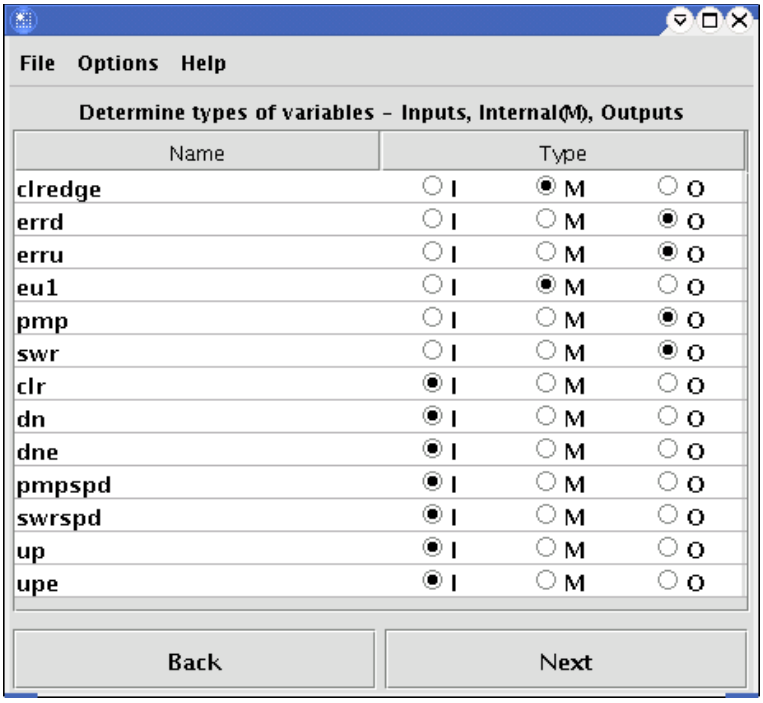
Vstup je připraven, stiskněte 'Next'. Objeví se karta pro zadávání časovačů. Program řízení nádrže nepoužívá žádný časovač, proto opět stiskněte 'Next'.

Následující karta slouží k určení inicializace, kterou používá řídicí program. Převáděný program sám žádnou inicializaci neprovádí, proto ponechejte 'No initiation' a stiskněte 'Next'.

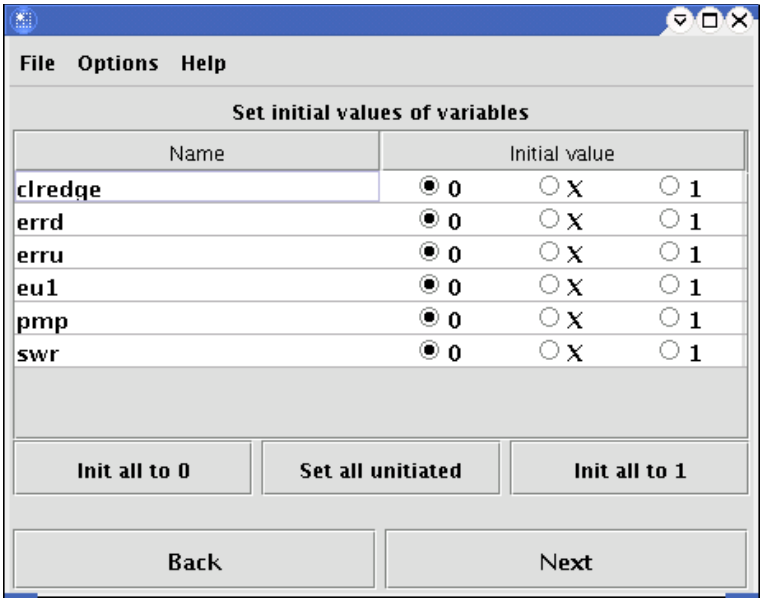
Nyní je třeba určit, které proměnné jsou vstupy, které výstupy a vnitřními proměnnými. Pro každou použitou proměnnou obsahuje tabulka na této kartě jeden řádek s přepínačem s volbami **I**, **M** a **O**. Volba **I** označuje vstupní, **M** vnitřní a **O** výstupní proměnnou. Proměnné, které se nevyskytují na pravé straně žádného přiřazení ze vstupní trans-množiny, jsou označené jako vstupní, ostatní jako vnitřní.



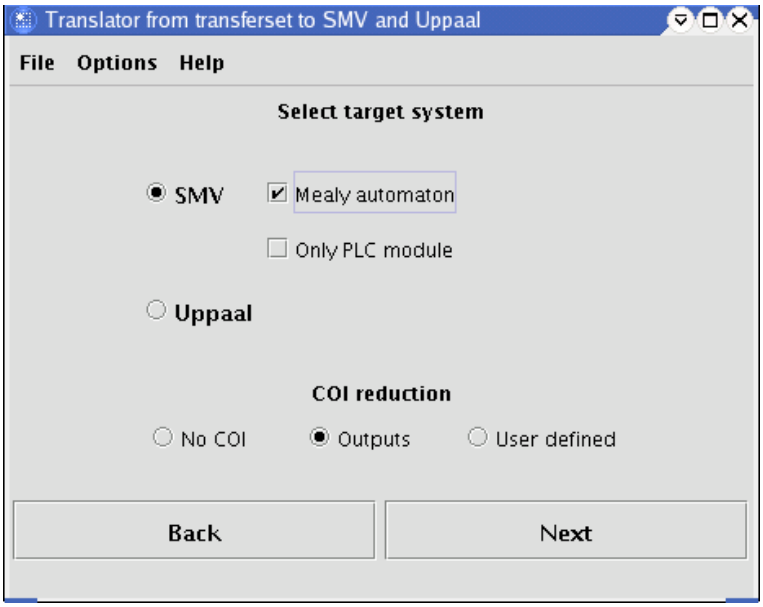
Přepněte proměnné `pmp`, `swr`, `erru` a `errd` na **O**, aby volby odpovídaly zadání a tlačítkem 'Next' přepněte na další fázi.



Nyní určete počáteční hodnoty vnitřních a výstupních proměnných. Volby **0** a **1** označují logickou 0, resp. 1. Proměnná s volbou **X** nemá určenou počáteční hodnotu. Za předpokladu, že PLC nastavuje před spuštěním programu celou oblast paměti proměnných na 0, zvolte **0** u všech proměnných, pro zjednodušení k tomu lze využít tlačítko 'Init all to 0'. Pokračujte na další kartu.



Zbývá zvolit, zda chcete vytvořit model v jazyku SMV či model pro nástroj Uppaal a nastavit některé parametry převodu.



Přepínač 'Mealy automaton' určuje, zda se má v SMV vytvořit model automatu typu Mealy. Bez výběru přepínače se vytvoří model automatu typu Moore.

Volba 'Only PLC module' způsobí, že se vytvoří pouze modul jazyka SMV popisující automat, tento modul bude nutné spojit s popisem vstupů PLC. Jinak bude výstupem programu soubor i s definicemi vstupních proměnných.

Pod popiskem 'COI reduction' se nachází přepínač, kterým můžete určit, zda se mají z modelu odstranit proměnné, které neovlivňují žádnou z proměnných, jejichž popis chcete modelovat.

Pro vytvoření modelu Mealy automatu, který popisuje chování výstupních proměnných a neobsahuje proměnné, které na ně nemají vliv, vyberte přepínače podle obrázku.

SMV model získaný uvedeným postupem se zobrazí na poslední kartě. Můžete ho uložit položkou menu File > Save.

```
MODULE PLC(clr, dn, dne, pmpspd, swrspd, up, upe)
VAR
 M_eu1 : boolean;
 M_errd : boolean;
 M_erru : boolean;
 M_pmp : boolean;
 M_swr : boolean;
ASSIGN
 init(M_eu1) := 0;
 init(M_errd) := 0;
 init(M_erru) := 0;
 init(M_pmp) := 0;
 init(M_swr) := 0;
 next(M_eu1) := eu1;
 next(M_errd) := errd;
 next(M_erru) := erru;
 next(M_pmp) := pmp;
 next(M_swr) := swr;
DEFINE
 eu1 := clr;
 errd :=  !clr&dne|!clr&M_errd|M_eu1&dne|M_eu1&M_errd;
 erru :=  !clr&upe|!clr&M_erru|M_eu1&upe|M_eu1&M_erru;
 pmp :=  M_pmp&!up&!upe|dn&!swrspd&!up&!upe|!swrspd&!up&!upe&dne;
 swr :=  !dn&!dne&!pmpspd|!dn&!dne&M_swr;

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

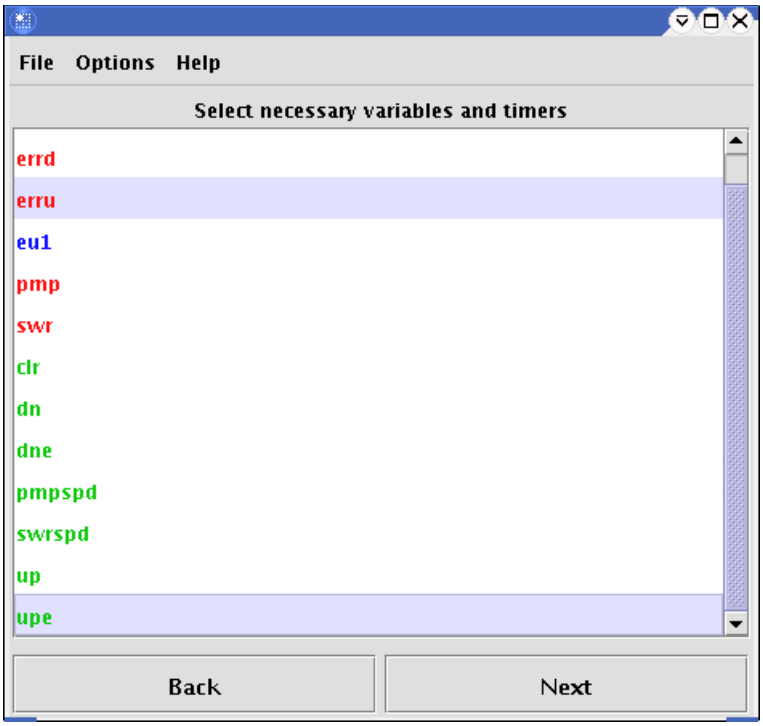
Na modelu můžeme verifikovat například formule.

```
SPEC
 AG (upe -> plc.erru)
SPEC
 AG (upe&!clr -> plc.erru)
SPEC
 AG !(plc.pmp&plc.swr)
```

První z nich vyjadřuje požadavek, aby výstup erru byl nastaven vždy po nastavení vstupu `upe` na 1, druhý, aby se tak stalo pokud je `upe`=1 a zároveň není `clr`=1. Poslední požadavek znamená, že výstupy `pmp` a `swr` nejsou nikdy zároveň nastavené na 1. Pro jejich ověření

přidejte uvedené řádky do souboru obsahujícího výsledný model a poté na něj aplikujte SMV.

Chcete-li vytvořit i jiný model stejného programu, stiskněte tlačítko 'Back' Pro vytvoření modelu automatu typu Moore definovaného pouze nad proměnnými, které se vyskytují v první formuli, vypňete přepínač 'Mealy automaton' a vyberte možnost 'User defined' u nastavení COI redukce. Jako další krok tentokrát následuje výběr potřebných proměnných, v tomto případě `erru` a `upe`:



SMV model vypadá následovně.

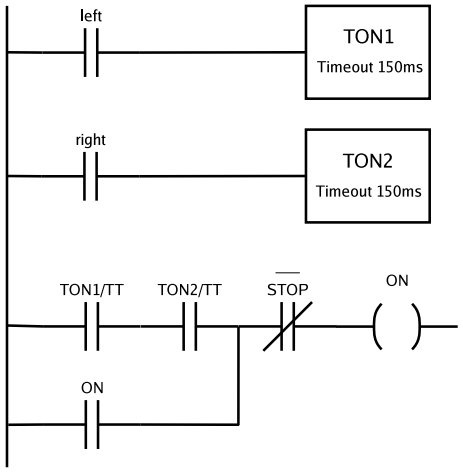
```
MODULE PLC(clr, upe)
VAR
 erru : boolean;
 eu1 : boolean;
ASSIGN
 init(erru) := 0;
 init(eu1) := 0;
 next(erru) := !clr&upe|!clr&erru|eu1&upe|eu1&erru;
 next(eu1) := clr;

MODULE main
VAR
 clr : boolean;
 upe : boolean;
 plc: PLC(clr, upe);
```

Jedná se o model Moore automatu, okamžité hodnoty proměnných ovlivňují až následující výstup, proto se i některé formule musí vyjádřit jinak. Například verifikovanou formuli `AG (upe -> plc.erru)` je třeba změnit na `AG (upe -> AX plc.erru)`.

C.2 Spouštění současným stiskem dvou tlačítek

V nebezpečných provozech je běžné používat pro spuštění procesu dvě tlačítka. Oparátor musí k jejich současnému stisku použít obě ruce, žádnou pak nemá ve chvíli spuštění v nebezpečném prostoru. Funkci sledování současnécho stisku tlačítek vykonává program zakreslený jako žebříčkový diagram. Napravo je jeho přepis do APLC programu. Spouštění časovače nahradil v APLC programu zápis do proměnné představující vstup časovače.



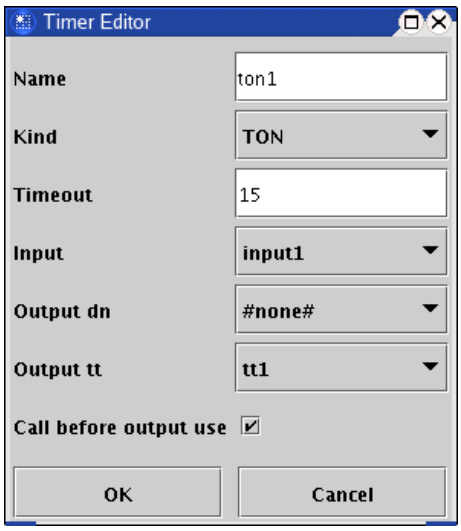

```
load left
store input1
load right
store input2
load tt1
and tt2
or on
and !stop
store on
end
```

Výsledek překladu APLC programu je

```
Result={ @f[1],
input1[left],
input2[right],
on[tt1.tt2.!stop+on.!stop] }
```

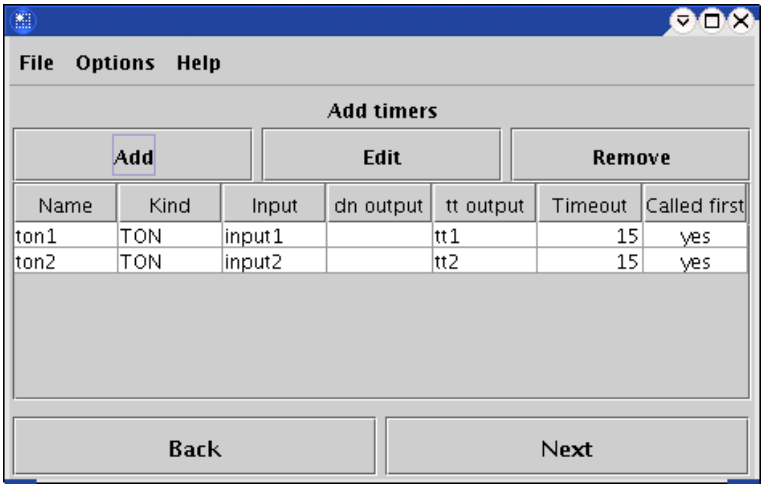
Vložíme ho do textového pole pro trans-množinu a stiskneme 'Next'.

Druhá karta slouží k přidání časovačů a určení jejich parametrů. Pro přidání časovače ton1 stiskněte tlačítko 'Add timer' vlevo nahoře. Objeví se okno 'Timer Editor'. Vyplňte vlastnosti podle obrázku. Nastavená doba časovače je 150 ms, jí odpovídající položka Timeout má hodnotu 15, jedna časová jednotka tak odpovídá 10 ms. Můžete zvolit i jiné měřítko, ale dodržte ho i u nastavení ostatních časů.

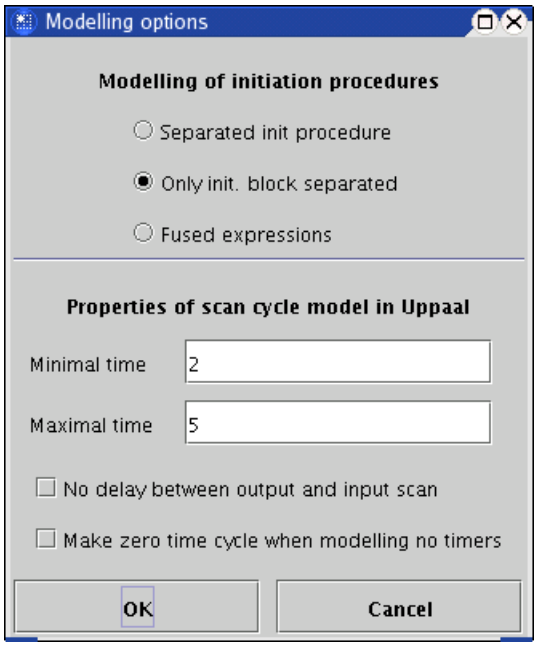


Přepínač ve spodní části okna určuje, zda časovač bude na svůj vstup reagovat už před vyhodnocením zbytku programu. V modelovaném programu se časovač ton1 nachází v prvním žebříčku, před prvním čtením hodnoty svého výstupu. Ve třetím žebříčku figuruje výstup `ton1/tt`, který je v okamžiku čtení již ovlivněn hodnotou vstupu časovače v daném cyklu. Přepínač tedy ponechejte zapnutý.

Druhý časovač nastavte stejným způsobem.



Položka menu Options > Modelling options otevře okno pro nastavení některých parametrů modelu. Jeho horní část se týká způsobu, jakým se modelují inicializační procedury, jelikož v programu žádné inicializační procedury nejsou, je tato část v tomto případě irelevantní. Spodní část nastavuje parametry scan cyklu v Uppaalu. Minimal time a Maximal time jsou dolní resp. horní mez doby scan cyklu.

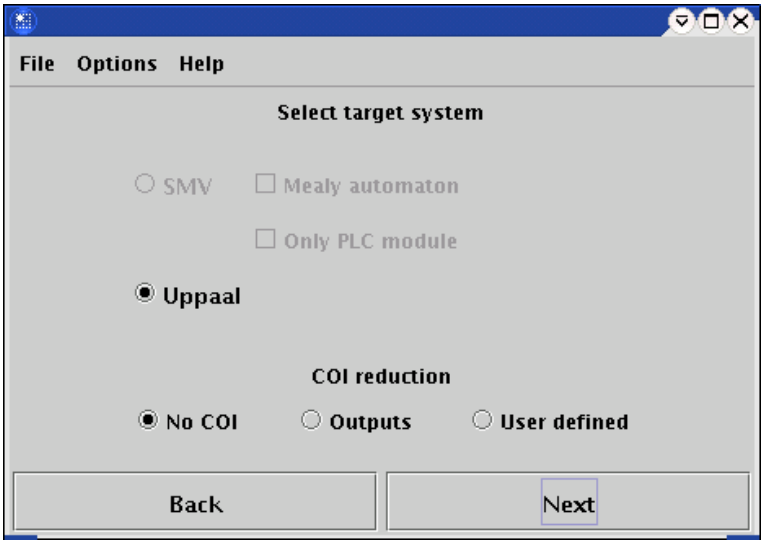


Výběrem přepínače 'No delay between output and input scan' můžete zvolit, že v modelu scan cyklu se ihned po provedení output scanu (společného s program scanem) načtou další vstupy. Veškeré zpoždění modelu scan cyklu se pak koncentruje mezi input scan a zápis výsledků

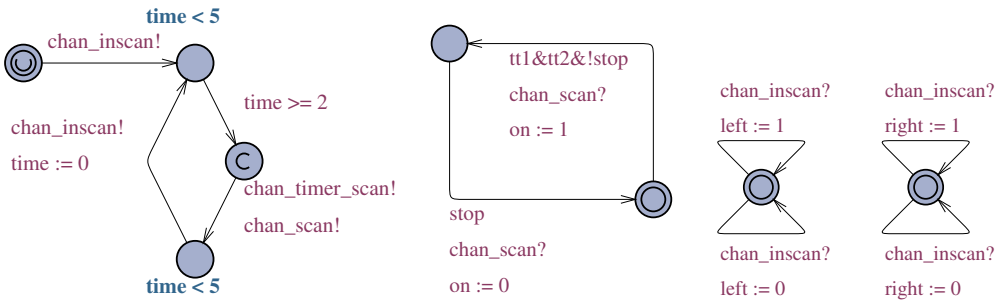
program scanu. Jestliže přepínač nevzvolíte, model umožní zpoždění i mezi zápisem výsledků program scanu a načtením dalších vstupů.

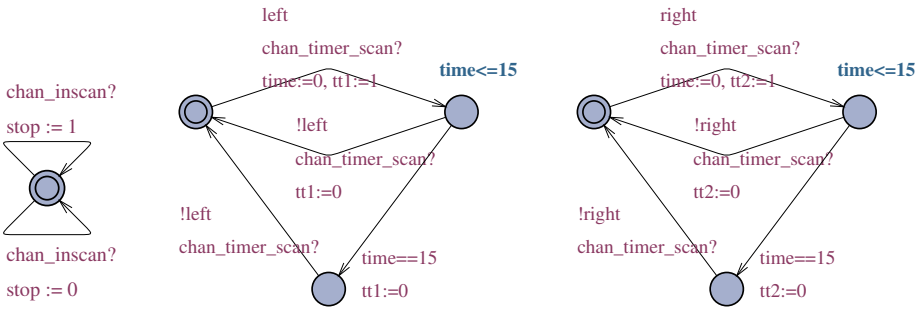
Přepínačem 'Make zero time cycle . . . ' lze pro modely bez časovačů nastavit cyklus bez časového zpoždění, který může vést na rychlejší verifikaci než cyklus jehož trvání určují časové meze. Pokud model obsahuje časovače nebo je vytvářen pro verifikaci časových požadavků, nelze cyklus nulové délky použít.

Na kartě časovačů zvolte stiskněte 'Next', poté zvolte 'No initiation', na další kartě můžete přepnout proměnnou on na výstup volbou **O**, ostatní proměnné jsou vstupy. Po přepnutí na další kartu zvolte počáteční hodnotu on rovnou **0**. Předposlední karta slouží k volbě verifikačního nástroje, ve kterém budete program verifikovat. Model obsahuje časovače, proto zbývá pouze varianta Uppaal, který je narozdíl od SMV určen pro verifikaci časovaných automatů. Pro modelování všech proměnných zvolte 'No COI', nebude se provádět redukce modelu odstraňováním proměnných.



Po stisku 'Next' se vytvoří model v XML formátu nástroje Uppaal. Po jeho uložení (File > Save) a otevření v Uppaalu můžete v okně simulátoru vidět model rozložený na několik procesů a podobný tomu na obrázku.



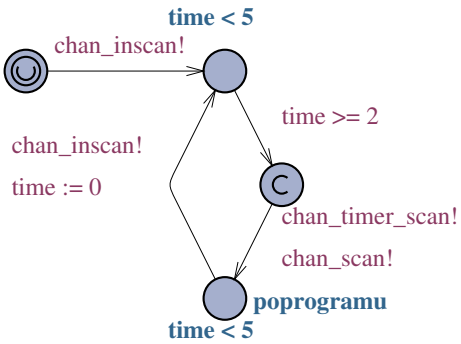


Procesy určující hodnoty vstupních proměných se provádějí pouze při emitování signálu chan_inscan procesem scan cyklu a přiřazují hodnotu náhodně podle toho, který ze dvou přechodů se provede. Modelují tak obrazy náhodně se měnících vstupů.

Na modelu můžeme verifikovat například formuli `stop --> !on` ... je-li vstup stop rovný logické 1, pak někdy v budoucnu musí být proměnná `on` nastavená na 0, čili program vždy zareaguje na stisk tlačítka `stop` vypnutím zařízení. Pravdivost této formule neříká, jestli k vypnutí dojde ihned. Pojmenování místa symbolizujícího stav po program scanu a před input scanem podle následujícího obrázku umožní verifikaci formule

`A[] stop&&proc_cycle.poprogramu imply !on`,

která říká, že proměnná `on` bude mít hodnotu 0 už v tom cyklu, kdy se ze vstupů přečte `stop`=1. Identifikátor „`proc_cycle`" je název procesu modelujícího scan cyklus.



V Uppaalu lze verifikovat i složitější formule obsahující časové podmínky jako

```
A[] proc_cycle.poprogramu && left && right && !stop && (proc_ton1.time<15)
&& (proc_ton2.time<15) imply on
```

Příloha D

Obsah přiloženého CD

K diplomové práci je přiložen CD-ROM s převodní aplikací a některými doplňkovými materiály. Obsah je rozdělen do tří adresářů.

doc obsahuje text diplomové práce v různých formátech a samostatný soubor s ukázkovými příklady uvedenými jako tutorial v příloze C.

java je adresář s převodní aplikací ve zdrojových kódech, přeloženou do bajtkódu a v .jar souboru. Nachází se zde i soubor s definicí tříd JavaHelpu potřebných pro spuštění nápovědy jako okna aplikace.

models obsahuje modely programů z kap. 7 a přílohy C.

Rejstřík

APLCTRANS, 6, 11
automat
  generovaný APLC programem, 15
  generovaný BPLC, 12
  časový, 24

Binární PLC, 11
Bounded Model Checking, 24, 31

codomain
  PLC codomain, 14
  t-přiřazení, 13
  trans-množiny, 14
COI redukce, 20, 57
Computation Tree Logic, *viz* CTL
cone of influence, *viz* COI redukce
CTL, 16, 17

domain
  PLC domain, 14
  t-přiřazení, 13
  trans-množiny, 14

Java Runtime Environment, 53
JFC Swing, 53

komprese, 14
Kripkeho model, 17

OBDD, 20, 21

rozšíření, 14

seřazený binární rozhodovací diagram, *viz* OBDD
SMV, 7, 26
  přiřazení
    init, 27
    next, 26
  sekce
    ASSIGN, 27
    DEFINE, 28
    FAIRNESS, 30
    SPEC, 29
    TRANS, 26
    VAR, 26

t-přiřazení, 13
  kanonické, 13
TCTL, 19, 33
Timed Computation Tree Logic, *viz* TCTL
trans-množina, 13

Uppaal, 7, 31

časovač, 9
  model, 50, 55