Temporální logika pro verifikaci konkurenčních systému

Patrik Halfar

Tématická práce k předmětu Teorie programovacích jazyků
Fakulta informačních technologií, Vysoké učení technické v Brně

Poslední úprava: 23. prosince 2010

Temporální logika pro verifikaci konkurenčních systému

Patrik Halfar[1]

FIT, Vysoké učení technické v Brně, email: ihalfar@fit.vutbr.cz

1 Úvod

Programy a systémy mohou být principiálně rozděleny do dvou charakterizujících kategorií. **Transformační programy** tvoří první kategorií, pro kterou je typické sekvenční zpracování: načtení vstupu, provedení transformace, zápis na výstup. Druhou skupinu programů lze nazvat **reaktivní programy**. Jejich chování obvykle spočívá v neustále kontrole vstupů a na základě změn na vstupech se provádí změna hodnot na výstupech. Prvním rozdílem, který lze pozorovat je fakt, že reaktivní program nekončí po zápisu hodnot na výstup, ale jeho činnost je nekončící. V důsledku tohoto je velmi často reaktivní program také programem konkurenčním, neboli paralelním. Tato práce se zabývá výhradně reaktivními programy.

V dnešní době většina důležitých programů spadá právě do kategorie reaktivních programů. Nejběžnějším příkladem takovéhoto systémů může být samotný operační systém. Běžně dnes musí operační systém zpracovávat několik vstupů najednou (klávesnice, myš, síťová karta, kamera, apod.) a pro každý z těchto vstupu vytvářet výstup, který může vycházet z kombinace několika vstupů. Od takovýchto systému se očekává, že se za každých okolností budou chovat korektně, a někdy chyba v programu může mít katastrofální následky.

Aby se takovýmto chybám předešlo provádí se verifikace programu. Verifikací se ověřuje, že program splňuje definované požadavky. Mezi obvyklé požadavky patří:

- program nedosáhne chybového stavu,
- v kritické sekci není víc než povolený počet procesů,
- nedochází ke ztrátě dat/událostí,
- atd.

Podmínkou pro provedení verifikace je zápis programů pomocí formálního prostředku, který verifikaci umožňuje. Současně je také nezbytné, mít stejný či jiný formální prostředek, který umožní požadované vlastnosti na program zřetelně definovat, protože nelze tvořit program bez jasného zadání, které by bylo následně verifikované.

Mimo jiné formalismus musí být obecný, neboť různé programovací jazyky používají pro komunikaci mezi procesy různé mechanismy (např. zasílání zpráv, sdílená proměnná, vzdálené volání procedur, ...). Stejně tak existují i různé mechanismy pro zajištění výlučného přístupu. Všechny tyto možnosti by měly být formálním prostředkem vyjádřitelné a ověřitelné.

Všechny výše popsané možnosti mohou být zapouzdřeny tzv. *generickým modelem* reaktivních programů, který umožňuje jednotný pohled na všechny výše popsané mechanismy. Jinými slovy tento model vytváří abstrakci nad většinou běžně užívaných jazyků a jejich konstrukcemi. Je vysoce pravděpodobné, že i jazyky, které vzniknou

později, budou konkretizací tohoto modelu (generický model bude možné využít k popisu programu popsaných v těchto jazycích). Existují i jiné prostředky pro vyjádření paralelismu jako jsou např. Petriho sítě.

Generický model je formalismus, který dovoluje popsat libovolný program a mechanismus jednotným způsobem. Ovšem není v možnostech tohoto modelu definovat požadavky na cílový program, nebo provést nad programem verifikaci jeho chování. K tomuto účelu se používá jiných nástrojů. Jedním z formálních prostředků, který dovoluje definování požadavků a jejich zpětnou verifikaci je jazyk temporální logiky. V této práci budou dále rozebrány možnosti temporální logiky pro specifikaci požadavků vlastností programů.

2 Základní modely

V úvodu bylo uvedeno, že generický model umožňuje provedení formální specifikace požadavků a verifikace výsledného programu. To ovšem vyžaduje možnost vzájemného mapování mezi generickým modelem a programovacím jazykem. Tato kapitola se zaměří na stručný popis generického modelu a další konkrétní modely, které představují mapování mezi programovacím jazykem generickým modelem. Pro toto mapování budou použity dva programovací jazyky – grafický (diagramový) a textový, ten bude využívat strukturovaného pseudojazyka podobného jazyku *Pascal*.

2.1 Generický model

Jednou ze složek generického modelu je základní transakční systém, který poskytuje mechanismus pro zachycení stavů systému a přechodů mezi těmito stavy, které modifikují proměnné systému. Úplný model transakčního systému rozšiřuje základní model o spravedlivost.

Základní transakční systém

Základní transakční systém pro modelování reaktivních systémů je čtveřice

$$(\Pi, \Sigma, \mathcal{T}, \Theta),$$

kde

- $\Pi = \{u_1, u_2, \ldots, u_n\}$ je konečnou množinou proměnných. Tyto proměnné jsou typované, a typ definuje doménu a rozsah hodnot, které mohou proměnné nabývat. Některé z těchto proměnných reprezentují data – deklarují se explicitně a program čte anebo modifikuje jejich hodnotu. Jiné proměnné jsou řídící – může to být aktuální pozice v programu, počet hodnot na zásobníku, apod.
- Σ představuje množinu stavů. Každý stav $s \in \Sigma$ je interpretací nad Π, která každé proměnné $u \in \Pi$ přiřazuje konkrétní hodnotu odpovídajícího typu.
- T je konečnou množinou přechodů. Každý přechod $\tau \in \mathcal{T}$ je transformací stavu systému a definován jako zobrazení $\tau : \Sigma \to 2^{\Sigma}$. Jinými slovy každému stavu s přiřazuje množinu stavů, do kterých se lze ze stavu s dostat. Každý stav s', do kterého se lze dostat ze stavu s se nazývá s-následníkem. Je-li s-následníkem opět stav s, pak se přechod nazývá *nečinným* a značí se τ_I.
- Θ se nazývá inicializační podmínkou. Stav, který splňuje tuto podmínku se nazývá *inicializačním stavem*.

Relace přechodu je výraz, který popisuje změny, provedené přechodem.

$$\rho(\Pi, \Pi') : C_r(\Pi) \wedge (y'_1 = e_1) \wedge (y'_2 = e_2) \wedge \cdots \wedge (y'_n = e_n)$$

C_r je tzv. *povolující podmínka*, ta musí být splněna, aby bylo možno realizovat daný přechod. Druhou část tvoří konjunkce *modifikačních příkazů*, které přiřazují nové hodnoty proměnným z Pi.

Povolené přechody jsou také přechody, o kterých platí, že pro stav $s \in \Sigma$ a přechod $\tau \in \mathcal{T}$ existuje τ-následník, čili $\tau(s) \neq \varnothing$.

Činné přechody představují množinu τ_D všech přechodů, které nejsou nečinné.

$$\mathcal{T}_D = \mathcal{T} \setminus \{\tau_I\}$$

Výpočet je nekonečná posloupnost stavů

$$\sigma : s_0, s_1, \ldots s_n,$$

kde s_0 se nazývá inicializační stav a platí $s \vDash \Theta$. A o každém stavu s_n platí, že je s_m-následníkem, kde $n = m+1$. Taková dvojce stavů se pak nazývá τ-krok. Každý výpočet může obsahovat buď nekonečný počet činných τ-kroků, nebo koncový stav. Koncovým stavem se rozumí takový stav, ze kterého vedou pouze nečinné přechody.

V dalších částech budou popsány konkrétní modely odvozené z generického modelu. Každý konkrétní model charakterizuje nějaký programovací jazyk a mapuje jej na základní transakční systém. Pro popis modelu existují dva hlavní programovací jazyky – grafický (diagramy) a textový. Popis pomocí diagramů představuje jistou variaci vývojových diagramů upravenou pro reprezentaci konkurentních programů. Textový popis představuje standardní strukturovaný programovací jazyk, používající syntaxi z několika existujících programovacích jazyků.

2.2 Model přechodových diagramů

Tento model používá pro popis grafického jazyka. Na rozdíl od vývojového diagramu nejsou v tomto modelu akce asociovány s uzly, ale s hranami vedoucí z uzlů, které reprezentují pozici v programu. Program P v přechodovém diagramu má tvar

$$P :: [\text{deklarace}][P_1 \parallel \cdots \parallel P_n],$$

kde $P_1, \ldots P_n, n \geq 1$ jsou procesy.

Deklarace je vždy na začátku programu a v této části se definují proměnné, které se dále používají v procesech, zápis má podobu:

mód proměnná, . . . , proměnná: typ **where** φ_i.

Mód může nabývat jednu z hodnot **in**, **out**, **local**, podle toho, zda se jedná o proměnnou vstupní, výstupní, nebo lokální. Lokální proměnné jsou takové, které nejsou přístupné vně programu. U vstupních proměnných je omezení, že program nemůže měnit jejich hodnotu.

Seznam proměnných lze použít, pokud se v programu vyskytuje více proměnných stejného módu a typu.

Typem proměnných mohou být základní typy – celočíselný, znak, apod, nebo strukturované jako pole, seznam, množina.

Výraz φ_i nabývá různého významu v závislosti na módu. Použití u módu **local** a **out** je povinné a definuje inicializační hodnoty proměnných. U módu **in** může být vynechán, a pokud je použit, má význam vymezení rozsahu vstupních hodnot, pro které se program bude chovat korektně. Formule φ se pak nazývá počáteční podmínkou a je dána konjunkcí všech formulí φ_i.

Procesy $P_1, \ldots P_n$ jsou každý reprezentován přechodovým diagramem (grafem). Uzly diagramu představují pozici v programu, to se obvykle značí jako sekvence

$$P_i :: \ell_0^i, \ell_1^i, \ldots, \ell_{t_i}^i$$

Pozice mohou být vstupní a výstupní. Ze vstupní pozice vedou hrany představující akci, kdežto z výstupních pozic žádné hrany nevedou. Množiny pozic dvou různých programů jsou disjunktní. Akce jsou uváděny jako popisky hran a mají formát střeženého přiřazení

$$c \to [\overline{y} := \overline{e}],$$

kde c je booleovský výraz střežící instrukci, $\overline{y}$ je seznam proměnných z Π a $\overline{e}$ je seznam výrazů. Délka obou seznamů je stejná a proměnné na k-tém místě je přiřazena hodnota k-tého výrazu. Všechny proměnné vyskytující se ve formuli popisku hrany musí být deklarovány na začátku programu.

Aby bylo možné zapsat aktuální stav programu, je nutné doplnit množinu datových proměnných o proměnné řídící. Ty se obvykle označují jako $\pi_1, \ldots, \pi_n$ a ukazují na aktuální pozici v procesů $P_1, \ldots, P_n$. Rozsah hodnot každé řídící proměnné je dán množinou pozic příslušného procesu.

Základní přechodový systém popsán pomocí diagramu je i v tomto případě definován čtveřicí tvořenou *stavovými proměnnými* $\Pi = \{\pi_1, \ldots, \pi_n, y_1, \ldots, y_m\}$, *množinou stavů*, která je tvořena permutací přes všechny možné hodnoty proměnných (omezené jejích doménou – typem), je zřejmé, že tato množina může být nekonečná, neboť doména hodnot může být také nekonečná, příkladem je doména celých čísel. Dále *množinou přechodů*, která je tvořen množinou hran, kterou lze popsat pomocí pozice, ze které vystupuje (ℓ) a do které vstupuje ($\tilde{\ell}$), přičemž obě pozice musí být z téhož procesu, a instrukcí tvaru $c \to [\overline{y} := \overline{e}]$. Relace přechodu pak má tvar

$$\rho_\tau : (\pi_i = \ell) \wedge c \wedge (\pi_i' = \tilde{\ell}) \wedge (\overline{y}' = \overline{e}),$$

kde $\tau \in \mathcal{T}$. Jinými slovy relace ρ_τ znamená, že proces P_i je na pozici ℓ, přechod τ je proveditelný a po jeho provedení je proces na pozici $\tilde{\ell}$ a modifikované proměnné obsahují nové hodnoty. Nakonec je tu inicializační podmínka, která nastavuje datovým

proměnným počáteční hodnoty a řídícím proměnným počáteční pozice jednotlivých procesů.

2.3 Textový model sdílené proměnné

Předchozí model využíval zápisu pomocí grafické notace pro popis programu. Takto popsaný program využívá pro vzájemnou synchronizaci mezi vlákny sdílenou proměnnou. Ekvivalentně může být program popsán i pomocí strukturovaného programovacího jazyka. Uvažovaný jazyk je tvořen příkazy, které mohou být rozděleny do dvou skupin – atomické a složené.

Atomické příkazy jsou takové příkazy, které jsou provedeny vždy ,,na jednou". Tedy takovéto příkazy nemohou být během svého vykonávání přerušeny a zachovávají tedy konzistentní stav. Mezi takovéto příkazy patří:

- **skip** je jednoduchým příkazem, který znamená *nedělat nic*.
- $\overline{y} := \overline{e}$ znamená přiřazení. Proměnným v seznamu $\overline{y}$ jsou přiřazeny hodnoty výrazů v seznamu $\overline{e}$. Délka obou seznamů je samozřejmě shodná.
- **await** c je příkazem vyhodnocujícím booleovskou podmínku c. Je-li tato podmínka splněna, pak se příkaz zachová stejně jako příkaz **skip** a přejde se na další příkaz. Ovšem v případě nepravdivosti této podmínky setrvává proces na tomto příkazu, dokud není vyhodnocovaná podmínka pravdivá.

Složené příkazy tvoří skupinu dvou a více příkazů. Během provádění těchto příkazů může dojít k jejích přerušení a může být započato provádění příkazů jiného procesu. V případě, že dva procesy pracují se stejnou proměnnou, může nastat situace, že tyto příkazy nezachovávají konzistenci proměnné. Mezi tyto příkazy patří:

- **if** c **then** S_1 **else** S_2 jako podmíněný příkaz. Tedy v závislosti na vyhodnocení booleovského výrazu c je proveden příkaz S_1 pro pravdivé ohodnocení nebo příkaz S_2 pro nepravdivé vyhodnocení podmínky. Může existovat i zkrácená varianta příkazu, kdy je vynechána část pro případ nesplněné podmínky, pak je podoba příkazu **if** c **then** S_1**.**
- S_1**;**S_2 představuje sekvenční provedení příkazů, je tedy proveden příkaz S_1 a po jeho dokončení je proveden příkaz S_2. Sekvenčně seřazených příkazů může být více než dva. Pro konkrétní případ, kdy je prvním příkazem příkaz **await** c, tedy **await** c**;**S používá zápis **when** c **do** S.
- S_1 **or** S_2 slouží pro selekci. Vykonání tohoto příkazu je dvou fázové – nejprve se vybere příkaz, který je možné provést a v druhé fázi se tento příkaz provede. Je-li možné provést oba příkazy, pak výběr příkazu, který se provede je nedeterministický. Selekce může být prováděna i mezi více než dvěma příkazy.
- $S_1 \parallel S_2$ představuje paralelní provedení příkazů. Tedy zahájení vykonání příkazu S_2 nečeká na dokončení příkazu S_1. I v tomto případě může být definováno více příkazů pro paralelní provedení.
- **while** c **do** S definuje cyklus. Tělo cyklu S se provádí dokud je splněna podmínka c. Není-li podmínka c splněna při započetí tohoto příkazu, nemusí být tělo příkazu provedeno ani jednou.

– **[lokální deklarace;** S**]** definuje blok příkazů, které používají lokální proměnné (nejsou viditelné mimo tento blok příkazů).

Každý příkaz S nebo S_i použitý při definici složeného příkazů může být tvořen atomickým nebo jiným složeným příkazem.

Programy v tomto jazyce mají tvar

$$P :: \big[deklarace; [P_1 :: S_1 \parallel \cdots \parallel P_n :: S_n]\big],$$

kde část deklarace slouží k definici proměnných a její tvar má podobu stejnou jako deklarace v jazyce přechodových diagramů

mód proměnná, ..., proměnná: typ **where** φ_i

se stejnou sémantikou jako v předchozím případě. Definice $P_i :: S_i$ pro $0 < i \leq n$ se nazývá pojmenovaný proces, kde P_i je jménem procesu a S_i tvoří tělo procesu, které je složeno z atomických nebo složených příkazů. Všechny procesy pak tvoří tělo programu.

Popisky textových příkazů U grafické definice programů byl každý příkaz asociován s hranou a vycházel z nějakého uzlu a v jiném uzlu končil. Množina názvů uzlů byla pro každý proces disjunktní a bylo tedy možné použít těchto uzlů pro určení pozice v programu. Protože textový program nic takového neobsahuje, zavádí se popisky, které mají stejné vlastnosti jako uzly v diagramu. Tedy každý příkaz může být označen úvodním a koncovým popiskem a pak lze pozici v programu určit pomocí popisků. Zavedenou notací pro označení koncového popisku je použití úvodního popisku se „stříškou" (např. je-li úvodní popisek ℓ_i, pak koncový popisek má tvar $\widehat{\ell_i}$).

2.4 Vztah generického modelu a textového modelu se sdílenou proměnnou

Generický model je tvořen základním přechodovým systémem a je tedy potřeba způsobu pro mapování textového jazyka na generický model a naopak.

Stavové proměnné jsou tvořeny datovými a řídícími proměnnými. Datové proměnné jsou všechny globálně a lokálně deklarované proměnné. Pro jednoduchost lze uvažovat, že každá proměnná má globálně unikátní jméno, to lze díky skutečnosti, že každou proměnnou lze přejmenovat bez vlivu na chování programu. Druhou skupinou jsou řídící proměnné, nejčastěji se uvažuje proměnná udávající pozici v programů, tedy proměnné $\pi_1, \ldots, \pi_n$ pro n procesů. Doména těchto proměnných je dána počtem příkazů každého procesu.

Stavy představují všechny možné kombinace hodnot proměnných, kterých mohou proměnné nabýt. Stejně jako v předchozím případě platí, že tato množina může být nekonečná.

Přechody mají definovanou relaci pro každý příkaz. Nečinný přechod je definován ekvivalentně jako v předchozím případě, tedy ρ_I : T. S každým činným příkazem může být asociován jeden či více přechodů. Množina přechodů asociovaných s příkazem S se označuje *trans(S)*.

Pro zkrácení zápisu relací přechodů pro definované příkazy se zavádí následující notace:

$$A \dot{-} a = A \setminus \{a\} \qquad \text{pokud } a \notin A \text{ pak } A \dot{-} a = A$$

$$A + b = A \cup \{b\} \qquad \text{pokud } b \in A \text{ pak } A + b = A$$

Pro každý atomický příkaz se přidá přechod τ_ℓ a definuje relace tvaru:

– Pro příkaz ℓ : **skip** : $\widehat{\ell}$ se definuje relace

$$\rho_\ell : ([\ell] \in \pi) \wedge (\pi' = (\pi \dot{-} [\ell] + \widehat{\ell})),$$

která říká, že je-li aktuálně program na úvodní pozici ℓ příkazu pak po provedení tohoto příkazu je úvodní pozice odstraněna z množiny aktuálních pozic v programu a nahrazena koncovou pozicí příkazů.

– Ekvivalentně pro příkaz ℓ : **await** c : $\widehat{\ell}$ má podobu

$$\rho_\ell : ([\ell] \in \pi) \wedge c \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}]),$$

což se podobá relaci příkazu **skip** s tím rozdílem, že vyžaduje pravdivě ohodnocenou podmínku c.

– Pro přiřazení $\ell : \overline{y} := \overline{e} : \widehat{\ell}$ se zavádí relace

$$\rho_\ell : ([\ell] \in \pi) \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}]) \wedge (\overline{y}' = \overline{e}),$$

což značí že proměnným v seznamu $\overline{y}$ se přiřadí hodnoty výrazů ze seznamu $\overline{e}$.

Pro složené příkazy lze rozlišit, zda se jedná o příkazy s podmínkou a příkazy spojující více příkazů. Pokud jde o příkazy s podmínkou, zavádí se pro každý příkaz dva přechody τ_ℓ^{T} a τ_ℓ^{F} pro splněnou a nesplněnou podmínku příkazu. Stejně tak jsou definovány i dvě relace. S výjimkou příkazu **when** c **do** S neboť při nesplnění podmínky c se pozice v programu nemění, neboť se čeká na splnění podmínky.

– Podmíněný příkaz ℓ : **if** c **then** $[\ell_1 : S_1]$ **else** $[\ell_2 : S_2]$ má relace

$$\rho_\ell^{\mathrm{T}} : ([\ell] \in \pi) \wedge c \wedge (\pi' = \pi \dot{-} [\ell] + [\ell_1])$$

$$\rho_\ell^{\mathrm{T}} : ([\ell] \in \pi) \wedge \not c \wedge (\pi' = \pi \dot{-} [\ell] + [\ell_2])$$

Protože část s příkazem pro nesplněný příkaz může být vynechána má pak příkaz tvar $\ell : \left[\textbf{ if } c \textbf{ then } [\ell : S_1] \right] : \widehat{\ell}$ zavádí se relace i pro tento tvar. V případě, že podmínka c je splnitelná, je relace ρ_ℓ^{T} totožná s tvarem pro plnou podobu příkazu a pro nesplnitelnou podmínku má relace podobu

$$\rho_\ell^{\mathrm{T}} : ([\ell] \in \pi) \wedge \not c \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}])$$

– Pro cyklus ℓ : **while** c **do** $\left[\tilde{l} : \tilde{S}\right] : \widehat{\ell}$ se definují relace

$$\rho_\ell^{\mathrm{T}} : ([\ell] \in \pi) \wedge c \wedge (\pi' = \pi \dot{-} [\ell] + [\tilde{\ell}])$$

$$\rho_\ell^{\mathrm{F}} : ([\ell] \in \pi) \wedge \not{c} \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}])$$

– Příkaz čekající na splnění podmínky c ℓ : **when** c **do** $[\tilde{\ell} : \tilde{S}]$ jak bylo uvedeno výše je jen jeden přechod a tedy jen jedna relace

$$\rho_\ell : ([\ell] \in \pi) \wedge c \wedge (\pi' = \pi \dot{-} [\ell] + [\tilde{\ell}])$$

Pro případ paralelního vykonání příkazů $\ell : \left[[\ell_1 : S_1 : \widehat{\ell_1}] \parallel \cdots \parallel [\ell_n : S_n : \widehat{\ell_n}]\right]$ se rovněž definují dva přechody – vstupní přechod τ_ℓ^{E} a výstupní přechod τ_ℓ^{X}. K těmto přechodům jsou pak definovány vstupní a výstupní relace

$$\rho_\ell^{\mathrm{E}} : ([\ell] \in \pi) \wedge \left(\pi' = (\pi \dot{-} [\ell]) \cup \{\ell_1, \ldots, \ell_m\}\right)$$

$$\rho_\ell^{\mathrm{X}} : \left(\{\widehat{\ell}, \ldots, \widehat{\ell_n}\} \subseteq \pi\right) \wedge \left(\pi' = (\pi \setminus \{\widehat{\ell_1}, \ldots, \widehat{\ell_n}\}) + [\widehat{\ell}]\right)$$

Zbývá definovat přechody a jejích relace pro příkazy zřetězení, výběru a bloku:

– Přechody a jejich relace pro zřetězení přechodu $S = [S_1, \ldots, S_n]$ jsou dány přechody prvního příkazu

$$\mathrm{trans}(S) = \mathrm{trans}(S_1)$$

– Naproti tomu přechody pro příkaz výběru $S = [S_1 \mathrm{or} \ldots \mathrm{or} S_n]$ jsou dány jako sjednocení všech přechodů

$$\mathrm{trans}(S) = \mathrm{trans}(S_1) \cup \cdots \cup \mathrm{trans}(S_n)$$

– Pro uzavřený blok příkazů $S = [deklarace; \tilde{S}]$ se použijí všechny přechody příkazů bloku

$$\mathrm{trans}(S) = \mathrm{trans}(\tilde{S})$$

Počáteční podmínka je poslední ze čtveřice tvořící základní přechodový systém. Tato podmínka se sestaví jako konjunkce všech formulí φ_i definující počáteční hodnoty při deklaracích globálních i lokálních proměnných a přiřazení vstupních pozic každého z procesů do řídících proměnných:

$$\Theta : \varphi_1 \wedge \cdots \wedge \varphi_m \wedge (\pi = \{\ell_1, \ldots, \ell_n\})$$

pro m datových proměnných a n procesů.

2.5 Synchronizační nástroje

Pro model pracující se sdílenou proměnnou je nezbytná přítomnost nějakého synchronizačního nástroje, který umožní zajistit konzistenci dat pro strukturované příkazy. Jedním z těchto nástrojů je *semafor*, je zároveň jediným, který je v této publikaci zmíněn. Semafor je tvořen dvojicí příkazů **request**(r) a **release**(r). Pro tyto příkazy jsou definovaný dva přechody s následujícími relacemi:

– $\ell : \text{request}(r) : \widehat{\ell}$

$$\rho_\ell : ([\ell] \in \pi) \wedge (r > 0) \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}]) \wedge (r' = r - 1)$$

– $\ell : \text{release}(r) : \widehat{\ell}$

$$\rho_\ell : ([\ell] \in \pi) \wedge (\pi' = \pi \dot{-} [\ell] + [\widehat{\ell}]) \wedge (r' = r + 1)$$

Protože grafický, který je jazyk uveden výše, rovněž používá pro vzájemnou komunikaci sdílené proměnné, může být semafor definován i v tomto jazyce.

2.6 Textový model zasílání zpráv

Jinou možností vzájemné komunikace mezi procesy je zasílaní zpráv. Tento model může být odvozen od textového modelu se sdílenou proměnnou provedením dvou úprav:

1. Zakázání sdílené proměnné jako komunikačního prostředku včetně vyšších synchronizačních nástrojů (např. semafor).
2. Přidání dvou nových příkazů pro zasílaní zprav a čtení zpráv. Také je nezbytné zavést komunikační kanál do/z kterého se zprávy posílají/čtou.

Jelikož tento model vychází z textového modelu pro sdílenou proměnnou, tak při mapování na základní přechodový systém využívá stejných relací pro všechny převzaté prvky. V důsledku to znamená vypuštění přechodů a relací pro semafor, neboť relace pro přiřazení hodnoty proměnné nerozlišuje, zda se jedná o proměnnou lokální či sdílenou. Dále je nezbytné doplnit přechody a přechodové relace pro operace s komunikačním kanálem.

Komunikační kanál se definuje v deklarační části obdobně jako proměnná

$$\text{mode}\alpha_1, \alpha_2, \ldots, \alpha_n : \textbf{channel of } \text{type } \textbf{where}\varphi$$

je vidět, že rozdíl od deklarace proměnné je pouze v přidání klíčového slova **channel of**. Druhý rozdíl, který není z tohoto zápisu zřejmý je v možnosti vynechat inicializační část s **where**. Pokud není inicializační formule definovaná, je kanál inicializován prázdný, jinou možností je naplnit kanál zprávami ještě před spuštěním programu.

Existují různé varianty komunikačních kanálu – *s pamětí* a *bez pamětí* (ten může být uvažován jako speciální typ kanálu s pamětí kapacity 1). U kanálu s pamětí se rozlišují dvě možností – omezená, resp. neomezená kapacita. Podle druhu komunikačního kanálu se mění i jeho vlastnosti a především je rozdíl znatelný v relacích operací používající kanál.

Zasílání zpráv je první ze dvou operací, které zprostředkovávají komunikaci skrze komunikační kanál a má tvar

$$\ell_s : \alpha \Leftarrow e : \widehat{\ell_s}$$

s významem, zapsat hodnotu výrazu e do kanálu α. Podmínkou je, že hodnota výrazu a kanál musejí být typově stejné.

Pro kanál s pamětí se pro operaci zasílaní zpráv zavádějí dvě relace podle toho, zda se jedná o paměť

– omezenou

$$\rho_{\ell_s} : ([\ell_s] \in \pi) \wedge (\pi' = \pi \dot{-} [\ell_s] + [\widehat{\ell_s}]) \wedge (\alpha' = \alpha \bullet e$$

– či neomezenou

$$\rho_{\ell_s} : ([\ell_s] \in \pi) \wedge (|\alpha| < N) \wedge (\pi' = \pi \dot{-} [\ell_s] + [\widehat{\ell_s}]) \wedge (\alpha' = \alpha \bullet e$$

Pozn. Operace $X \bullet a$ znamená připojení hodnoty a na konec seznamu X.

Příjem zpráv je inverzní operací k zasílaní zpráv. Tato operace přečte zprávu z kanálu α a zapíše do proměnné y

$$\ell_r : \alpha \Rightarrow y : \widehat{\ell_r},$$

kde se opět vyžaduje typová shoda kanálu a proměnné. Přečtením zprávy se zpráva z kanálu odstraní.

Relace pro operaci příjmu zprávy nerozlišuje, zda se jedná o kanál s omezenou či neomezenou pamětí a má vždy stejný tvar

$$\rho_{\ell_r} \left[\begin{array}{l} ([\ell_r] \in \pi) \wedge (|\alpha| > 0) \\ \wedge \\ (\pi' = \pi \dot{-} [\ell_r] + [\widehat{\ell_r}]) \wedge (y' = \mathit{hd}(\alpha)) \wedge (\alpha' = \mathit{tl}(\alpha)) \end{array} \right]$$

Pozn. Funkce *hd()* vrací první prvek seznamu a funkce *tl()* vrací seznam bez prvního prvku.

Kanál bez paměti je zvláštním případem, tento kanál se též nazývá *synchronním*, neboť vyžaduje, aby zpráva do něj zapsána byla současně přečtená. Tedy pro tento kanál se přidává jen jeden přechod $\tau_{\langle \ell_s, \ell_r \rangle}$ jehož relace má tvar

$$\rho_{\langle \ell_s, \ell_r \rangle} : \left[(\{\ell_s, \ell_r\} \subset \pi) \wedge (\pi' = \pi \setminus \{\ell_s, \ell_r\} \cup \{\widehat{\ell_s}, \widehat{\ell_r}\}) \wedge (y' = e) \right]$$

2.7 Spravedlivý přechodový systém

Rozšířením základního přechodového systému o dvě komponenty

– $\mathcal{J} \subset \mathcal{T}$ – množina správných přechodů
– $\mathcal{C} \subset \mathcal{T}$ – množina soucitných přechodů

vznikne spravedlivý přechodový systém $(\Pi, \Sigma, \mathcal{T}, \Theta, \mathcal{J}, \mathcal{C})$.

Množina správných přechodů je množinou takových přechodů, o kterých platí, že stanou-li se od nějakého stavu výpočtů trvale povolenými, pak během výpočtu budou provedeny nekonečně mnohokrát.

Množina soucitných přechodů je množinou přechodů, o kterých platí, že pokud jsou nekonečně mnohokrát povolené, pak budou i nekonečně mnohokrát provedené.

Výpočet spravedlivého přechodového systému $P = (\Pi, \Sigma, \mathcal{T}, \Theta, \mathcal{J}, \mathcal{C})$ je definován jako nekonečná posloupnost stavů $\sigma : s_0, s_1, \ldots$, splňující těchto pět podmínek:

1. **Inicializace** $s_0 \vDash \Theta$, první stav je inicializační.
2. **Postupnost**. Pro každé dva stavy s_i, s_{i+1} existuje přechod $\tau \in \mathcal{T}$ takový, že s_{i+1} je τ-následníkem stavu s_i.
3. **Činnost**. Posloupnost obsahuje buď nekonečně mnoho činných přechodů, nebo koncový stav.
4. **Správnost**. Pro žádný přechod $\tau \in \mathcal{J}$ nenastane situace, že by byl nepřetržitě povolený od nějakého stavu výpočtů a nebyl proveden nekonečněkrát.
5. **Soucitnost**. Pro žádný přechod $\tau \in \mathcal{C}$ nenastane situace, že by byl nekonečně mnohokrát povolený, ale byl proveden jen konečným počtem opakování.

V této kapitole byly popsány formální modely – grafický i textové, které mohou na vyšší úrovni abstrakce popsat reaktivní program. Výhoda těchto modelů spočívá v generickém přístupu popisu chování programů a tedy i možnosti definice formálních požadavků a jejích verifikace. Jazyk, který je v této publikaci zvolen pro popis formálních požadavků na program je jazyk temporální logiky.

3 Jazyk temporální logiky

Jazyk temporální logiky vychází z jazyka predikátové logiky prvního řádu (FOL, z angl. *First-Order Logic*) a rozšiřuje množinu operátorů, které lze aplikovat na formule jazyka FOL, o temporální operátory. Ze začátku budou tedy připomenuty základní vlastnosti jazyka FOL, které budou následně rozšířeny o temporální operace.

3.1 Jazyk predikátové logiky prvního řádu

Každý jazyk je postaven nad abecedou. Abecedu tvoří: proměnné, funkce, predikáty. Nad abecedou jsou sestavená slova, která se spojují pomocí spojek a kvantifikátoru a vznikají formule.

Každá proměnná může být ohodnocena. Ohodnocení znamená přiřazení hodnot proměnným, množina ohodnocení všech proměnných je stav. Každá formule se vyhodnocuje nad stavem. Tedy je splnitelnou nebo nesplnitelnou v daném stavu.

Nejprve nechť jsou tedy definovány stavební prvky FOL. Základním prvkem každého jazyka je abeceda (značeno V), nad kterou je jazyk vystavěn. V případě FOL je abeceda tvořena

- spočetnou množinou typovaných proměnných,
- logickými spojkami ($\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$),
- obecným ($\forall$) a existenčním ($\exists$) kvantifikátorem,
- neprázdnou množinou predikátových operátorů $\mathcal{P}$,
- množinou funkčních operátorů $\mathcal{F}$,
- pomocných symbolů (např. závorky, čárky, apod).

Pro každý prvek z množiny $\mathcal{P}$ a $\mathcal{F}$ existuje přirozené číslo $n \geq 0$, které je označováno jako *arita* operátoru, resp. funkce, a vyjadřuje kolika **termů** se tento operátor, resp. funkce, týká. Je-li arita nulová, pak u predikátu se hovoří o tzv. nestrukturovaném

výroku, tedy takovém výroku, který se netýká žádného **termu**, a u funkce hovoříme o konstantě.

Arita predikátu i funkce byla definována nad **termy**. Term lze definovat následujícími pravidly, které lze aplikovat rekurzivně:

1. Každá proměnná z abecedy V je term,
2. Nechť $x_1, x_2, \ldots, x_n$ jsou termy a $f \in \mathcal{F}$ je funkcí arity $n \geq 0$, pak také $f(x_1, x_2, \ldots, x_n)$ je term.

Obdobným způsobem aplikaci rekurze lze dále definovat *atomickou formuli*, *booleovskou formuli* a obecnou *formuli* jazyka FOL:

Atomická formule

1. Každá proměnná booleovského typu z abecedy V je atomickou formuli.
2. Nechť $x_1, x_2, \ldots, x_n$ jsou termy a $p \in \mathcal{P}$ je predikátem arity $n \geq 0$, pak $p(x_1, x_2, \ldots, x_n)$ je atomickou formulí.

Booleovská formule

1. Každá atomická formule z abecedy V je booleovskou formulí.
2. Aplikací logických spojek na booleovské formule vznikne opět booleovská formule.

Formule

1. Formulí je každá booleovská formule.
2. Každá formule, nad kterou je aplikován kvantifikátor je opět formulí.

3.2 Vyhodnocování formulí jazyka predikátové logiky prvního řádu

Každá proměnná jazyka FOL je typována, tedy může nabývat pouze hodnot z domény daného typu. Pro proměnné booleovského typu jde tedy o hodnoty *true* a *false*, pro proměnné celočíselného typu o hodnoty $\ldots, -1, 0, 1, \ldots$, apod. **Ohodnocením** proměnné se rozumí situace, kdy je proměnné přiřazena konkrétní hodnota příslušného typu. Množinou ohodnocení všech proměnných z abecedy označujeme **stav**. Stav je tedy realizací všech proměnných abecedy, pro stav s a proměnnou $u \in V$ značíme

$$s[u].$$

Z definice formule lze odvodit, že ohodnocení predikátu, resp. funkce, je dáno vyhodnocením funkce nad ohodnocenýma proměnnýma a lze tedy pro formuli p psát $s[p]$ pro ohodnocení formule ve stavu s.

Dva stavy s a s' jsou nazývány **x-odlišné** jestliže platí

$$s[y] = s'[y] \qquad \text{pro každé} y \in V \setminus \{x\}$$

Jinými slovy se oba stavy liší právě v ohodnocení proměnné x.

Ohodnocení formule jazyka FOL nad stavem s znamená rozhodnout, zda je či není splnitelná pro dané ohodnocení proměnných. Pro libovolný stav s a formule p, q jazyka FOL říkáme, že *formule p je splnitelná ve stavu s*, značíme

$$s \models p$$

a definujeme

$$s \Vdash p \qquad \text{právě když } s[p] = T$$

pro libovolnou booleovskou formuli p, nebo

$$s \Vdash \exists u : p, \text{ resp. } s \Vdash \forall u : p,$$

pro formuli p a proměnnou $u \in V$, právě tehdy, když

$$s' \Vdash p$$

pro nějaké, resp. každé, s', které je u-odlišné od s. Pro formule s logickými spojkami pak definujeme splnitelnost pro spojky $\neg$ a $\vee$:

$$s \Vdash \neg p \text{právě tehdy, když } s \nVdash p$$

$$s \Vdash p \vee q \qquad \text{právě tehdy, když } s \Vdash p \text{ nebo } s \Vdash q$$

3.3 Rozšíření FOL o temporální vlastnosti

Aby bylo možné rozšířit jazyk FOL a získat tak temporální logiku, je nutné zavést do jazyka FOL *časovou složku*, pomocí které lze pak definovat **model**, nad kterým se provádí vyhodnocování formulí jazyka temporální logiky, a **temporální operátory**, pomocí nichž lze upravovat platnost temporálních formulí v čase.

Čas je z fyzikálního pohledu množinou reálných čísel s relací ostrého uspořádání a definovaným minimálním prvkem symbolizujícím počátek. Pro práci s časem v počítačových systémech se čas diskretizuje a může být tedy definován jako množina přirozených čísel včetně nuly. Tato množina rovněž splňuje vlastnosti existence minimálního prvku a relace ostrého uspořádání.

Existence minimálního prvku znamená exaktní určení počátku, v použité doméně pro vyjádření času se počátek označuje časem nula, tedy hodnoty času, které nemůže nic předcházet.

Model bude definován následovně: Nechť existuje spočetná množina $\mathcal{S}$ všech možných stavů nad proměnnými jazyka FOL a surjektivní zobrazení $f : \mathbb{N}_0 \leftarrow \mathcal{S}$, z časové množiny na množinu stavů. Jinak řečeno ke každé hodnotě času je přiřazen právě jeden stav, přičemž každý stav se může v čase vyskytovat opakovaně, nebo se nemusí vyskytovat vůbec. Označením s_n pak značíme stav s v čase n a množinu všech těchto stavů označujeme model σ. Tento model přebírá z časové domény relaci ostrého uspořádání, v důsledku lze tedy každý stav s_n označit jako stav předcházející, resp. následující, stavu s_m pro $n < m$, resp. $n > m$. Model pak můžeme značit jako posloupnost stavů

$$\sigma : s_0, s_1, \ldots, s_n$$

Alternativní notaci pro vyjádření stavu s_j modelu σ je zápis

$$(\sigma, j)$$

3.4 Temporální operátory

Temporální operátory rozšiřují abecedu jazyka FOL o unární a binární operátory, které definují platnost formule v čase. Tyto operátory lze rozdělit do dvou kategorií: **přímé** a nepřímé. Přímý operátor se váže ke stavu bezprostředně předcházejícímu, resp. následujícímu, v čase ke stavu aktuálnímu. Přímými operátory jsou operátory *Previous* a *Next*. Ostatní temporální operátory, které budou uvedeny dále jsou nepřímé, tedy vyjadřují vztah formule k minulosti, resp. budoucnosti.

Poznámka k použitým symbolům: Symbolem $A \models p$ se označuje stavová formule (formule bez temporálních operátorů) p, která je splnitelná z množiny předpokladů A v každém stavu modelu σ, je-li množina předpokladů prázdná, pak formule p je tautologií. Naproti tomu symbolem $AvDashp$ se označí splnitelnost temporální formule p z množiny předpokladů A, která představuje konkrétní stav modelu σ. Není-li množina předpokladů uvedená, pak se uvažuje první stav modelu σ (jinými slovy stav s_0).

„Previous" a „Next" Jak již bylo uvedeno, jedná se o jediné dva přímé operátory. Formálně jsou definovány:

$$s_t \vDash \ominus p = (t > 0) \wedge s_{t-1} \vDash p$$

$$s_t \vDash \bigcirc p = s_{t+1} \vDash p`$$

U operátoru „Previous" je na rozdíl od operátoru „Next" nezbytné ověření, zda lze substraktivní operaci provést, tato vlastnost vychází z existence minimálního prvku. V důsledku to způsobí, že formule s operátorem „Previous" nad prvním stavem je vždy nesplnitelná. Protože v některých případech může představovat tato vlastnost komplikaci, existuje tzv. „slabší" verze tohoto operátoru, která je splnitelná nad prvním prvkem

$$s_t \vDash \widetilde{\ominus} p = (t = 0) \vee ((t > 0) \wedge ((s_{t-1} \vDash p))$$

„Has been always" a „Henceforth" Dalšími unárními operátory se vyjadřuje skutečnost, že formule je platná po celou dobu od aktuálního stavu do minulosti, resp. budoucnosti.

$$s_t \vDash \boxminus p = s_j \vDash p \text{ pro každé } j \leq k$$

$$s_t \vDash \Box p = s_j \vDash p \text{ pro každé } j \geq k$$

„Once" a „Eventualy" Poslední dva unární operátory slouží k vyjádření faktu, že formule byla v některém z předcházejících, resp. bude v některém z následujících, stavů někdy splněna.

$$s_t \vDash \diamondminus p = s_j \vDash p \text{ pokud existuje } j \leq k$$

$$s_t \vDash \Diamond p = s_j \vDash p \text{ pokud existuje } j \geq k$$

„Since" a „Until" Binární operátory *Since* a *Until* jsou podmíněnou verzí operátoru *Once* a *Eventualy*, kde podmínka definuje stav, před kterým, resp. po kterém, není formule platná. Jinými slovy pro platnost formule lze definovat interval, který je vymezen aktuálním stavem a stavem, ve kterém je splnitelná podmiňující formule.

$$s_t \vDash p\mathcal{S}q = \exists j : 0 \leq j \leq t \wedge s_j \vDash q \wedge \forall i : j < i \leq t \wedge s_i \vDash p$$

$$s_t \vDash p\mathcal{U}q = \exists j : t \leq j \wedge s_j \vDash q \wedge \forall i : t < i \leq j \wedge s_i \vDash p$$

„Back-To" a „Wait for (Unless)" Operátor *Since*, resp. *Until*, garantuje, že formule q byla, resp. bude, někdy splněna. Obvykle mohou být požadována slabší tvrzení a to – buď je formule p splnitelná na od počátku, resp. do konce, od aktuálního stavu, nebo její platnost je omezená splnitelností formule q. Takovouto vlastnost vyjadřují právě operátory *Back-To* a *Wait for*.

$$s_t \vDash p\mathcal{B}q = s_t \vDash (p\mathcal{S}q) \vee \boxminus p$$

$$s_t \vDash p\mathcal{W}q = s_t \vDash (p\mathcal{U}q) \vee \Box p$$

3.5 Striktní operátory

U všech nepřímých operátoru jsou vidět operátory menší, resp. větší, nebo rovno. Existují dvě možnosti, jak vyjádřit vztah formule pouze k minulosti, resp. budoucnosti.

Přehlednější možností je zadefinování nových operátorů, které se nazývají striktní temporální operátory a obvykle se označují „stříškou" nad operátorem. Tyto operátory mají ekvivalentní definici jako výše uvedené verze, avšak používají ostrou nerovnost.

Druhou možností je použití kombinace nepřímého a přímého operátoru. Tato varianta se může stát poněkud nepřehlednější, pokud je formule delší a obsahuje více temporálních operátorů.

$$\bigcirc \Diamond p \leftrightarrow \widehat{\Diamond} p$$

3.6 Základní množina operátorů

V předchozí podkapitole byly uvedeny všechny temporální operátory, které lze použít. Celkem tedy existuje jedenáct temporálních operátorů (nepočítaje striktní verze, v důsledku skutečnosti, že je lze vyjádřit pomocí kombinace přímého a nepřímého operátoru), které lze použít pro zápis temporálních formulí. Připojí-li se ještě operátory z jazyka FOL, vznikne množina šestnácti operátoru, které lze používat. Tento počet lze zredukovat na pouhých šest formulí, pomocí kterých lze vyjádřit všech šestnáct možností.

$$\neg, \vee, \bigcirc, \mathcal{W}, \widetilde{\ominus}, \mathcal{B}$$

Jak vyjádřit ostatní operátory jazyka FOL pomocí operátoru $\neg, \vee$ lze nalézt např. v [1]. Ostatní temporální operátory se pomocí základní operátorů vyjádří následovně:

$$\Box p \approx p \mathcal{W} \mathbf{F} \qquad \boxminus p \approx p \mathcal{B} \mathbf{F}$$
$$\Diamond p \approx \neg \Box \neg p \qquad \diamondminus p \approx \neg \boxminus \neg p$$
$$p \mathcal{U} q \approx (p \mathcal{W} q) \wedge \Diamond q \qquad p \mathcal{S} q \approx (p \mathcal{B} q) \wedge \diamondminus q$$
$$\ominus p \approx \neg \widetilde{\ominus} \neg p$$

3.7 Vlastnosti temporálních operátorů

Dualita tedy vlastnost, že dvě logicky ekvivalentní formule mohou být zapsány dvěma různými způsoby.

– Budoucí operátory

$$\neg \Box p \approx \Diamond \neg p \qquad \neg \Diamond p \approx \Box \neg p$$
$$\neg (p \mathcal{U} q) \approx (\neg q) \mathcal{W} (\neg p \wedge q) \qquad \neg (p \mathcal{W} q) \approx (\neg q) \mathcal{U} (\neg q \wedge \neg q)$$
$$\neg \bigcirc p \approx \bigcirc \neg p$$

– Minulé operátory

$$\neg \boxminus p \approx \diamondminus \neg p \qquad \neg \diamondminus p \approx \boxminus \neg p$$
$$\neg (p \mathcal{S} q) \approx (\neg q) \mathcal{B} (\neg p \wedge q) \qquad \neg (p \mathcal{B} q) \approx (\neg q) \mathcal{S} (\neg q \wedge \neg q)$$
$$\neg \ominus p \approx \widetilde{\ominus} \neg p \qquad \neg \widetilde{\ominus} p \approx \widetilde{\ominus} \neg p$$

Idempotence je vlastnost, která tvrdí, že dvojí použití téhož operátoru nemění význam temporální formule.

– Budoucí operátory

$$\Box \Box p \approx \Box p$$
$$\Diamond \Diamond p \approx \Diamond p$$
$$p \mathcal{U} (p \mathcal{U} q) \approx p \mathcal{U} q \qquad p \mathcal{W} (p \mathcal{W} q) \approx p \mathcal{W} q$$
$$(p \mathcal{U} q) \mathcal{U} q \approx p \mathcal{U} q \qquad (p \mathcal{W} q) \mathcal{W} q \approx p \mathcal{W} q$$

– Minulé operátory

$$\boxminus \boxminus p \approx \boxminus p$$
$$\diamondminus \diamondminus p \approx \diamondminus p$$
$$p \mathcal{S} (p \mathcal{S} q) \approx p \mathcal{S} q \qquad p \mathcal{B} (p \mathcal{B} q) \approx p \mathcal{B} q$$
$$(p \mathcal{S} q) \mathcal{S} q \approx p \mathcal{S} q \qquad (p \mathcal{B} q) \mathcal{B} q \approx p \mathcal{B} q$$

Absorpce umožňuje vynechání opakovaně vyskytujícího se operátorů pokud jsou splněny následující podmínky

– Budoucí operátory

$$\Diamond \Box \Diamond p \approx \Box \Diamond p$$
$$\Box \Diamond \Box p \approx \Diamond \Box p$$
$$p \mathcal{W} (p \mathcal{U} q) \approx p \mathcal{W} q \qquad (p \mathcal{U} q) \mathcal{W} q \approx p \mathcal{U} q$$
$$p \mathcal{U} (q \mathcal{W} q) \approx p \mathcal{W} q \qquad (p \mathcal{W} q) \mathcal{U} q \approx p \mathcal{U} q$$

– Minulé operátory

$$\diamondsuit \boxminus \diamondsuit p \approx \boxminus \diamondsuit p$$
$$\boxminus \diamondsuit \boxminus p \approx \diamondsuit \boxminus p$$
$$p\mathcal{B}(p\mathcal{S}q) \approx p\mathcal{B}q \qquad (p\mathcal{S}p)\mathcal{B}q \approx p\mathcal{S}q$$
$$p\mathcal{S}(q\mathcal{B}q) \approx p\mathcal{B}q \qquad (p\mathcal{B}q)\mathcal{S}q \approx p\mathcal{S}q$$

Důsledkem idempotence a absorpce je fakt, že každý neprázdný řetězec operátorů $\boxminus$ a $\diamondsuit$ lze redukovat na jednu z následujících forem

$$\boxminus p \qquad \diamondsuit p \qquad \boxminus \diamondsuit p \qquad \diamondsuit \boxminus p$$

a ekvivalentně pro budoucí operátory.

3.8 Vyhodnocování formulí jazyka temporální logiky

Vyhodnocování temporálních formulí je obdobné vyhodnocování formulí jazyka FOL, avšak splnitelnost formule ve stavu s se nahrazuje splnitelností formule v čase t modelu σ. Platí tedy, že temporální formule p je splnitelná v čase t modelu σ (značeno $(\sigma, t) \vDash p$) právě tehdy, když platí

$$s_t \vDash p$$

pro $\sigma : s_0, s_1, \ldots, s_t, \ldots$ a zároveň $t \geq 0$.

3.9 Odvozovací pravidla

Dokazování pomocí logických jazyků je založeno na existenci odvozovacích pravidel, která umožní z axiomů (tedy tvrzení, která jsou obecně uznána za pravdivá a jejich platnost není nutné dokazovat) odvodit jiné tvrzení. Toto odvození se pak nazývá důkaz. V jazyce temporální logiky lze užít stejná odvozovací pravidla jako v jazyce FOL, jedná-li se o formuli bez temporálních operátorů. Některá pravidla lze použít bez ohledu na to, zda formule obsahuje či neobsahuje temporální operátory (např. *Modus ponens*) a jiná pravidla se zavádějí pro odvozování s temporálními operátory.

- Pravidlo zobecnění
$$\frac{\vDash p}{\vDash \Box p}$$
- Pravidlo specializace
$$\frac{\vDash \Box p}{\vDash p}$$
- a další.

Existují i další odvozovací pravidla, avšak jejich výčet, stejně tak jako výčet axiomů, které lze použít pro verifikaci reaktivního programů překračuje rozsah této práce. Přehled všech pravidel, včetně jejich rozebrání se nachází v [2].

4 Validace programu pomocí temporálních formulí

V kapitole 2 byly popsány způsoby, jak graficky či textově popsaný program převést na přechodový systém. Výpočet přechodového systému je pak nekonečnou sekvencí $\sigma' : s'_0, s'_1, \ldots$ stavů nad množinou Π. Nechť V je slovník obsahující Π a $\sigma : s_0, s_1, \ldots$ je modelem nad slovníkem V. Pak lze o modelu σ říci, že koresponduje s výpočtem σ', pokud každý stav s_i je identický se stavem s'_i při omezení na podslovník Π. Takový

model σ se pak nazývá P-modelem. Množina všech P-modelů nad slovníkem V se pak znační $\mathcal{M}_V(P)$. P**-dostupným** stavem se nazývá každý stav, který se vyskytuje na nějaké pozici v P-modelu.

Každá temporální formule p, pro níž je definována množina V_p, která obsahuje všechny proměnné vyskytující se ve formuli p, se nazývá P**-platnou**, pokud pro každý model z $M_V(P)$ je formule p splnitelná a zároveň $V = V_p \cup \Pi$. Je-li formule p splnitelná pro každý stav, který je P-dostupným, pak se formule p nazývá P**-stavově platnou**.

Příklad. Uvažujme program $P :: [\text{local} x : \text{int where } x = 0; x = x + 1]$ a formuli $p : (x = 0)$. Pak výpočet programu P bude mít následující tvar

$$\langle x : 0\rangle, \langle x : 1\rangle, \langle x : 2\rangle, \ldots$$

a je zřejmé, že formule p je P-platnou, neboť je splnitelná každým výpočtem programu P, avšak není P-stavově platnou.

5 Specifikace vlastností programů jazykem temporální logiky

Pro popis vlastností programů se používá tzv. lokální jazyk. Ten se skládá z lokálních formulí, na které jsou aplikované logické a temporální operátory. Lokální formule mohou popisovat vlastnosti samotných stavů, nebo přechodů vedoucích do stavů.

Vlastnosti, které se u stavů běžně popisují lze rozdělit do šesti tříd, přičemž každá z těchto tříd má základní formuli charakterizující tuto třídu.

5.1 Formule lokálního jazyka

Jak bylo výše uvedeno, lokální jazyk se skládá ze dvou typu predikátu, a to predikáty stavové, které se vyhodnocují samostatně nad stavy, tak predikáty přechodové, které se vyhodnocují nad stavem a jeho přímým předchůdcem. Mezi stavové predikáty se řadí predikáty určující pozici v programu, tzv. lokační predikáty.

Lokační predikáty jsou celkem tři. A pro příkaz

$$\ell : S : \widehat{\ell}$$

jsou definovány následovně:

- **at**$\_\ell$**, at**$\_S$

$$\text{at}\_\ell : [\ell] \in \pi$$

- **after**$\_S$

$$\text{after}\_S : \text{at}\_\widehat{\ell}$$

- **in**$\_S$ pokud je S složeným příkazem, a S' je nějakým vnořeným příkazem

$$\text{in}\_S : \bigvee_{S' \preceq S} \text{at}'_S$$

Nechť je uvažován přechod τ a jeho relace přechodu $\rho_\tau : C_r \wedge (\overline{y} = \overline{e})$, pak mohou být definovány další stavové predikáty:

– **Povolený přechod**

$$\mathit{enabled}(\tau) : C_r$$

zároveň se definuje i množina povolených přechodů $T \in \mathcal{T}$

$$\mathit{enabled}(T) : \bigvee_{\tau \in T} \mathit{enabled}(\tau)$$

– **Ukončující přechod**

$$\mathit{terminal} = \bigwedge_{\tau \in \mathcal{T}_D} \neg \mathit{enabled}(\tau)$$

Dříve bylo uvedeno, že i přechodové predikáty se vyhodnocují nad stavy. Aby mohl být zachycen přechod využívá se i bezprostředně předcházejícího stavu a tím lze vyhodnotit vlastnosti přechodu (změny, které byly realizací přechodu provedeny). Nechť je tedy zaveden predikát *first*, který říká, že se jedná o první stav posloupnosti, tedy takový stav, který nemá žádného předchůdce. (Pozn. pomocí znaku - v horním indexu bude značen bezprostředně předcházející stav, tedy zápis s^- znamená, ze se jedná o přímého předchůdce stavu s). Obecný tvar přechodové formule má podobu

$$\neg \mathit{first} \wedge \varphi(\Pi^-, \Pi)$$

Pro přechod $\tau \in \mathcal{T}$ se definuje jediný predikát **last-taken**(τ) vyjadřující, že z předcházejícího stavu byl proveden přechod. Tento predikát má tvar

$$\mathit{last\text{-}taken}(\tau) : \neg \mathit{first} \wedge \rho_\tau(\Pi', \Pi)$$

Protože v kapitole o modelech byl definován i textový model využívající zasílání zpráv, jsou definovány i predikáty pro tyto přechody. Nechť α je komunikační kanál, pak pro asynchronní komunikaci (kanál má kapacitu větší než 1) jsou definovány predikáty

– Odeslání dat

$$[\alpha \leftarrow v] : \neg \mathit{first} \wedge (\alpha = \alpha^- \bullet v)$$

– Příjem dat

$$[\alpha \rightarrow v] : \neg \mathit{first} \wedge (v \bullet \alpha = \alpha^-)$$

Pro synchronní komunikaci, tedy kapacita kanálu je rovná 1 a proces po zaslání zprávy nepokračuje, dokud není zpráva vyzvednuta jiným procesem má predikát tvar

$$\mathit{comm}(\ell, m, v) : \mathit{last\text{-}taken}(\tau_{\langle \ell, m \rangle}) \wedge (v = e^-),$$

kde ℓ je návěští příkazu odesílajícího zprávu kanálem α a m je návěští příkazů čtoucího zprávu z kanálu α.

Někdy nemusí být podstatné co je do/z kanálu zapisováno/čteno, proto se definují speciální varianty predikátu $[\alpha \rightarrow]$ a $[\alpha \leftarrow]$, které značí, že z kanálu α byla přečtena, resp. byla do něj zapsána, nějaká hodnota.

Specifikační proměnná se zavádí, je-li potřeba uchovat si hodnotu nějakého stavu, aby s touto hodnotou mohly být porovnávány hodnoty proměnných v jiném stavu.

5.2 Klasifikace vlastností

Lokační jazyk představuje množinu jednoduchých vlastností, které mohou být nad programem definovány. Složitější vlastnosti vznikají jako kompozice jednodušších vlastností na které jsou aplikované logické a temporální operátory. Požadovaná vlastnost je tedy vymezena nějakou formulí p. Bude-li uvažována množina všech možných výpočtů, které mohou nad program vzniknout, nechť je označena Σ^ω pak formule p vymezuje podmnožinu $\mathcal{P} \subset \Sigma^\omega$, tak, že každý běh, pro který je formule p splnitelná je prvkem množiny $\mathcal{P}$.

$$\forall \sigma : \sigma \in \Sigma^\omega \wedge \sigma \vDash p \leftrightarrow \sigma \in \mathcal{P}$$

Dvě formule p a q jsou ekvivalentní, pokud se jimi vymezené množiny shodují.

Vzhledem ke skutečností, že logická formule definující vlastnost definuje množinu výpočtů, které splňují danou vlastnost, lze pozorovat korelaci uzávěrových vlastností nad těmito vlastnostmi vůči disjunkci, konjunkci, negaci a množinovými operacemi sjednocení, průnik a doplněk nad množinami vymezenými těmito vlastnostmi.

$p \vee q$	odpovídá	$\mathcal{P} \cup \mathcal{Q}$
$p \wedge q$	odpovídá	$\mathcal{P} \cap \mathcal{Q}$
$\neg p$	odpovídá	$\overline{\mathcal{P}} = \Sigma^\omega \setminus \mathcal{P}$

Bezpečné vlastnosti jsou takové, jež jsou splněny v každém stavu výpočtu. Jsou tedy stavově invariantní. Mezi takovéto vlastnosti patří většinou nějaká omezení, ve smyslu během celého výpočtu nenastane případ, že by proměnná nabyla hodnoty menší než x, každé události e_2 předchází událost e_1, apod. Základní tvar formule charakterizující tuto třídu je

$$\Box p$$

pro libovolnou minulou formulí p, tedy takovou formulí, která neobsahuje budoucí operátory. Třída bezpečných vlastností je uzavřena vůči pozitivním operacím, tedy vůči průniku a sjednocení.

Garantující vlastnosti zajišťují, že se nějaká událost během výpočtu stane. Avšak nezaručují, že se bude opakovat nebo kdy nastane. Typickou vlastností kterou lze zařadit do této třídy je ukončení programu. Tedy zaručuje, že má-li program někdy skončit, tak skončí. Základní tvar formule charakterizující tuto třídu má tvar

$$\Diamond p$$

pro libovolnou minulou formuli p.

Třída garantujících vlastností není uzavřena vůči doplňku, avšak z duality operátoru základních formulí vyplývá, že komplementem garantované vlastnosti je vlastnost bezpečná a naopak komplementem bezpečné vlastnosti je vlastnost garantující. Této skutečnosti lze použít k odvození, že třída garantujících vlastností je uzavřena vůči pozitivním operacím.

Obligátní vlastnosti jsou takové vlastnosti, které nemohou být vyjádřeny samostatně pomocí bezpečné nebo garantující vlastnosti a je zapotřebí vyjádřit je disjunkcí těchto dvou vlastností, čímž vznikne třída prostých obligátních vlastností. Základní formule těchto vlastností má tvar

$$\Box p \vee \Diamond q$$

pro libovolné minulé formule p a q. Třída prostých obligátních vlastností je uzavřena vůči disjunkci, avšak aplikaci konjunkce na tyto vlastnosti vznikne silnější třída. Ta se nazývá třídou obligátních vlastností a základní tvar formule je

$$\bigwedge_{i=1}^{n} [\Box p_i \vee \Diamond q_i]$$

pro minulé formule $p_i, q_i, i = 1, \ldots, n$.

Odpovídající vlastnosti představují třídu vlastností, které garantují, že se nějaká událost bude vyskytovat nekonečně mnohokrát. Název teto třídy značí, že se obvykle využívá pro definici vlastností, že na každý podnět nastane reakce. Základní formule charakterizující tuto třídu má tvar

$$\Box \Diamond p$$

pro každou minulou formuli p. Tato třída je uzavřena vůči pozitivním operacím.

Bezpečné a garantující vlastnosti mohou být vyjádřeny jako speciální případ odpovídajících vlastností, to vyplývá z následujících ekvivalencí:

$$\Box p \sim \Box \Diamond (\boxminus p)$$

$$\Diamond p \sim \Box \Diamond (\diamondminus p)$$

Ustalující vlastnosti jsou vlastnosti, které garantují, že od jistého stavu nastane situace, že vlastnost zadaná formulí bude navždy splněna, tedy systém se ustálí. Základní tvar formule vymezující tuto třídu má tvar

$$\Diamond \Box p$$

pro minulou formuli p. Třída těchto vlastností je uzavřená vůči pozitivním operacím.

Třídy odpovídajících a ustalujících vlastností jsou duální. I pro tuto třídu platí, že bezpečné a garantující vlastnosti jsou speciálním případem ustalujících formulí.

Reaktivní vlastnosti se částečně podobají třídě obligátních vlastností, neboť se rovněž skládá z prosté reaktivní vlastnosti, která je definována jako disjunkce odpovídajících a ustalujících vlastností. Tedy prostá reaktivní formule má tvar

$$\Box \Diamond p \vee \Diamond \Box q$$

pro libovolné minulé formule p a q. Stejně je to i s uzávěrovými vlastnostmi. Třída prostých reaktivních vlastností je uzavřena vůči disjunkci, avšak konjunkcí vznikne třída silnějších vlastností. Základní tvar formule reaktivních vlastností má proto tvar

$$\bigwedge_{i=1}^{n} [\Box \Diamond p_i \vee \Diamond \Box q_i]$$

Hierarchické uspořádání obligátních a reaktivních vlastností . V důsledku skutečnosti, že konjunkcí dvou prostých reaktivních, resp. obligátních, vlastností vznikne třída silnějších vlastností dochází k hierarchickému uspořádání reaktivních, resp. obligátních, vlastností. Tedy třída vlastností vzniklá konjunkcí n prostých vlastností je slabší než třída vzniklá konjunkcí $n + 1$ prostých vlastností.

Bezpečné a postupné vlastnosti. Všechny vlastnosti vyjma vlastností bezpečných mohou být zapouzdřeny jako vlastnosti postupné. To je zapříčiněno přítomností operátoru $\Diamond$, který se vyskytuje v jejich základních formulích a garantuje, že formule se stane planou až během vykonávání programu.

5.3 Standardní formule

Dříve definované kanonické formule jsou založeny na aplikaci budoucích operátorů $\Box$ a $\Diamond$ na libovolnou minulou formuli. Obvykle je ovšem žádoucí využít dalších operátorů pro specifikaci požadovaných vlastností. Použití dalších operátorů může ovšem zkomplikovat zařazení vlastností do třídy charakterizované kanonickou formulí. Proto existují standardní formule, které používají i ostatních operátorů a umožňují klasifikaci takto vzniklých formulí.

Standardní bezpečné formule jsou formule, které vzniknou aplikací následujících dvou pravidel:

1. Každá minulá formule je standardní bezpečnou formulí.
2. Jsou-li p a q standardní bezpečné formule, pak také

$$p \wedge q \qquad p \vee q \qquad \bigcirc p \qquad \Box p \qquad p \mathcal{W} q$$

jsou také standardní bezpečné formule.

Standardní garantující formule jsou formule, které vzniknou aplikací následujících dvou pravidel:

1. Každá minulá formule je standardní garantující formulí.
2. Jsou-li p a q standardní garantující formule, pak také

$$p \wedge q \qquad p \vee q \qquad \bigcirc p \qquad \Diamond p \qquad p \mathcal{U} q$$

jsou také standardní bezpečné formule.

Standardní obligátní formule jsou formule, které vzniknou aplikací následujících dvou pravidel:

1. Každá standardní bezpečná formule je obligátní formulí.
2. Každá standardní garantující formule je standardní obligátní formulí.
3. Jsou-li p a q standardní obligátní formule, pak také

$$p \wedge q \qquad p \vee q \qquad \neg p \qquad \bigcirc p$$

jsou také standardní obligátní formule.

4. Je-li p standardní bezpečná formule, q standardní obligátní formule a r standardní garantující formule, pak

$$p\mathcal{W}q \qquad q\mathcal{U}r$$

jsou také standardní obligátní formule.

Standardní odpovídající formule jsou formule, které vzniknou aplikací následujících dvou pravidel:

1. Každá standardní obligátní formule je standardní odpovídající formulí.
2. Jsou-li p a q standardní odpovídající formule, pak také

$$p \wedge q \qquad p \vee q \qquad \bigcirc p \qquad \Box p$$

jsou také standardní odpovídající formule.
3. Je-li p a q standardní odpovídající formule a r standardní garantující formule, pak

$$p\mathcal{W}q \qquad p\mathcal{U}r$$

jsou také standardní odpovídající formule.

Standardní ustalující formule jsou formule, které vzniknou aplikací následujících dvou pravidel:

1. Každá standardní obligátní formule je standardní ustalující formulí.
2. Jsou-li p a q standardní ustalující formule, pak také

$$p \wedge q \qquad p \vee q \qquad \bigcirc p \qquad \Diamond p$$

jsou také standardní ustalující formule.
3. Je-li p a q standardní bezpečné formule a r standardní ustalující formule, pak

$$p\mathcal{W}q \qquad q\mathcal{U}r$$

jsou také standardní ustalující formule.

5.4 Bezpečnost vs. živost

Alternativním způsobem klasifikace vlastností může byt členění na vlastnosti garantující bezpečnost a vlastnosti garantující živost. Neformálně řečeno bezpečné vlastnosti garantují, že se nestane něco špatného a živé vlastnosti naproti tomu garantují, že se stane něco dobrého.

Bezpečné vlastnosti mohou být rozděleny do dvou kategorií. První jsou vlastnosti stavově invariantní. Splnění těchto vlastností nezávisí na průběhu výpočtu a musí být splněny pro každý výpočet nezávisle na jeho stavech. Mezi takovéto vlastnosti patří

– Částečná správnost – po ukončení program je splněná nějaká podmínka $\Box(ater\_P \rightarrow q)$

– V programu se nevyskytují uváznutí – pokud program skončí, pak byl vykonán celý $\Box(terminal \rightarrow after\_P)$
– Vzájemné vyloučení – žádné dva procesy nejsou současně v kritické sekci $\Box\neg(in\_C_1 \wedge in\_C_2)$ pro kritické sekce C_1 a C_2.

Druhou kategorií bezpečných vlastností jsou vlastnosti k minulosti invariantní. Tyto vlastnosti využívají minulých formulí a mohou vyjadřovat vztahy mezi aktuálním stavem a nějakým stavem předcházejícím. Takto popsané vlastnosti mohou vyjadřovat

– Monotónnost
– Existenci předcházející události
– Neexistenci předcházející události → hodnota nemůže být na výstupu dvakrát

Bude-li uvažována třída bezpečných vlastností jako podtřída postupných vlastností, pak třída vlastností garantující živost je doplňkem k třídě bezpečných vlastností. Vlastnosti, které lze vyjádřit jsou:

– Konečnost a úplná správnost – konečnost je živost garantující formule $\Diamond ater\_P$. Úplná správnost je pak získána konjunkcí Konečností a částečné správností.
– Garance vyskytující události – příkladem použití této vlastnosti může být například pro program tisknoucí sudá čísla. Pak je vymezující tvrzení: každé sudé číslo bude někdy vytištěno, apod.
– Absence stárnutí – $\Box \Diamond \neg at\_\ell$
– Dostupnost – každý proces se někdy dostane do kritické sekce.

5.5 Specifikace vlastností programů

Program, který je specifikován pomocí jazyka temporální logiky se pak skládá ze dvou částí. V první části se definují vstupní a výstupní proměnné programu včetně jejích inicializačních hodnot a případných omezení na vstupních hodnoty. Druhá část je tvořena popisem chování celého programu pomocí několika formulí. Mezi tyto formule patří jak klasické vlastnosti, které jsou uvedeny výše tak i další vlastnosti vyplývající z dokumentace.

Nad takto specifikovaným program lze pomocí výše naznačených mechanismů verifikovat správnost programů. Skutečně sestavený program v nějakém konkrétním jazyce pak musí být validován, že odpovídá popsanému modelu, nebo musí být definováno mapování z konkrétního jazyka na přechodový systém, nad kterým se pak správnost programu ověří.

5.6 Specifikace modulů

Specifikace vlastností programu poskytne omezení definované na program jako celek. Tento postup je aplikovatelný pro práci na malých projektech a v relativně malých týmech. To je zapříčiněno tím, že takovýto program pak může být verifikován pouze jako celek. Alternativou je rozdělení projektů do menších modulů. Pak se formální specifikace definuje pro každý modul samostatně, což znamená, že každý modul může být samostatně verifikován. Nezbytnou součástí specifikace modulu je pak definice jeho rozhraní, skrze které komunikuje s okolními moduly.

Každý modul, který splňuje svou specifikaci pro libovolný vstup respektující jeho rozhraní se nazývá modulárně platný. V případě modulární specifikace je nezbytné provést také kompoziční verifikaci, která ověří, že konjunkcí dílčích specifikací je splněna celková specifikace programů, jinými slovy spojením všech modulů vznikne program, který splňuje celkové pro něj definované požadavky.

6 Závěr

Cílem práce bylo zaměřit se na možnosti specifikace a verifikace programů pomocí jazyka temporální logiky. V úvodu je uveden generický model vycházející ze základního přechodového systému jakožto formalismu pro popis programů. Tento model byl postupně rozšířen o mapování programů popsaných pomocí grafického jazyka. Na to navazuje mapování programů popsaných strukturovaným programovacím jazykem na základní přechodový systém a to jak jazyka využívajícího sdílené proměnné pro vzájemnou komunikaci mezí procesy tak i jazyka založeného na zasílaní zpráv.

Další část je věnována stručnému úvodu do jazyka temporální logiky včetně popisu některých důležitých vlastností, které lze zkoumat u jeho operátorů. Velmi obecně jsou popsány metody sestavování důkazů v tomto jazyce. Na tuto část navazuje kapitola, která definuje některé důležité vlastností, které se od reaktivních systémů očekávají a uvádí jejich popis pomocí jazyka temporální logiky. Nakonec je uveden princip použití specifikace a verifikace pro popis programů. A také je zmíněna možnost rozdělení programu na moduly a využití výhod modulárního programování.

Reference

1. L.S. Cauman. *First-order logic: an introduction*. Walter de Gruyter, 1998.
2. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.