MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Analýza robustnosti spojitých dynamických systémů v distribuovaném prostředí

DIPLOMOVÁ PRÁCE

Jan Papoušek

Brno, Jaro 2013

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně dne 24. května 2013
Jan Papoušek

Vedoucí práce: RNDr. David Šafránek, Ph.D.

Poděkování

Děkuji vedoucímu své práce Davidu Šafránkovi za odborné vedení a poskytnutí cenných rad, Svenu Dražanovi za inspirativní konzultace, Tomáši Vejpustkovi za spolupráci při implementaci nástroje Parasim a celé Laboratoři systémové biologie za poskytnutí technického zázemí.

Děkuji svým rodičum Janu a Evě Papouškovým za podporu, kterou mi poskytli během studia a vůbec v celém mém dosavadním životě. Děkuji své nastávající manželce Tereze Doležalové za naše dlouhé rozpravy formující mé životní směřování, jenž vyústilo nejen v tuto práci.

Bez Vás by tato práce nevznikla.

Shrnutí

Práce prezentuje algoritmus pro analýzu dynamických systémů zadaných pomocí soustavy obyčejných diferenciálních rovnic vzhledem k chování charakterizovanému vlastností temporální logiky signálů. Výsledkem analýzy je představa o tom, jakým způsobem ovlivňují změny modelu jeho chování. Popisovaná metoda se zakládá na již existujícím algoritmu a výpočtu lokální robustnosti.

Algoritmus byl implementován v programovacím jazyce Java do podoby nástroje Parasim tak, aby analýzu bylo možno spustit v prostředí se sdílenou nebo distribuovanou pamětí. Výpočetní model a architektura nástroje umožňují komponenty odpovídající jednotlivým částem algoritmu snadno nahrazovat, případně použít tuto platformu pro jiný typ výpočtu.

Vlastnosti algoritmu a škálovatelnost implementace pro sdílenou i distribuovanou paměť byly ověřeny spuštěním analýzy nad vybranými modely.

Klíčová slova

dynamický systém, soustava diferenciálních rovnic, STL, monitoring, analýza vlastnosti, robustnost

Obsah

Kapitola 1

Úvod

Okolo nás se vyskytují různé systémy specifických vlastností chování v čase, k jejichž pochopení se stále častěji používá modelování. Růst významu modelů se stupňuje s rozšiřujícím se používáním výpočetní techniky, která je schopná s těmito modely efektivně pracovat. Modelování se již nepoužívá pouze v tradičních oblastech, jako je například předpověď počasí, ale s postupem času proniká i do oblastí, jakými je například modelování procesů v živých organismech, kterému se věnuje systémová biologie [33].

S rostoucím významem modelů se zdá být stále důležitější dokázat formulovat vlastnosti, které od modelů očekáváme, za použití nějaké formální logiky a následně je automatizovaným způsobem nad těmito modely ověřovat. Není však vhodné spokojit se s ověřením vlastnosti pro jedno konkrétní nastavení hodnot proměnných a parametrů modelu. Analýza by měla jít více do hloubky a nahlížet na model obecněji. Model je možné vychýlit z jeho ideálního nastavení nebo naopak jeho ideální nastavení vzhledem k dané vlastnosti najít. V kontextu modelů živých organismů takový druh analýzy otevírá možnost studovat systém v méně příznivých nebo dokonce pro daný organismus životu nebezpečných podmínkách.

Cílem této diplomové práce je naimplementovat algoritmus právě pro takový druh analýzy, který bude schopen z různých ohodnocení parametrů a proměnných efektivně najít ta ohodnocení, která splňují požadovanou vlastnost. Implementovaný algoritmus vychází z algoritmu prezentovaného v diplomové práci Svena Dražana [10] a principu lokální robustnosti použitého v nástroji Breach [6], který se tímto tématem rovněž zabývá. Na rozdíl od algoritmu použitého v nástroji Breach však implementovaný algoritmus není takovou mírou provázán s numerickou simulací a používá jednodušší způsob pokrytí množiny ohodnocení parametrů a proměnných.

Výsledkem práce je volně dostupná aplikace Parasim, která vznikla ve spolupráci s Tomášem Vejpustkem, jenž zajistil uživatelské rozhraní. Tvorba této aplikace proběhla v rámci Laboratoře systémové biologie[1] pod do-

1. `http://sybila.fi.muni.cz/`

hledem Svena Dražana. Parasim umožňuje provést výpočet analýzy nejen na jednom počítači, ale i v distribuovaném prostředí a tím analýzu urychlit.

Následující text se nejprve v kapitole 2 zabývá základními pojmy, jako je definice modelu a logiky pro vyjádření vlastností. Následuje kapitola 3 popisující řešený problém, původní algoritmus, ze kterého práce vychází, pojem lokální robustnosti a provedené úpravy v algoritmu. Kapitola 4 se věnuje implementaci modulárního systému a výpočetního modelu. Kapitola 5 popisuje evaluaci, která zahrnuje provedení několika analýz a vyhodnocení naměřených dat. Na závěr, v kapitole 6, je práce shrnuta a jsou nastíněny směry, kterými se lze ubírat dále v budoucnu.

Kapitola 2

Pojmy a východiska

Tato kapitola se věnuje základním pojmům nutným k pochopení algoritmu pro analýzu dynamických systémů představeného dále v kapitole 3 a kontextu, v jakém byl navržen. Jedná se zejména o popis reprezentace zkoumaných modelů a požadovaných vlastností.

2.1 Modelování

Před popisem samotného modelování pomocí obyčejných diferenciálních rovnic je nutné říci, co se od vytvářených modelů zpravidla očekává. Model má zprostředkovat zjednodušený pohled na zkoumaný systém a umožnit tím systém snáze pochopit a případně předpovídat některá jeho chování. Fakt, že model je jen zjednodušením, znamená, že se vždy od reality liší a reflektuje jen některé aspekty chování zkoumaného systému [29, str. 48].

Jednoduchým příkladem modelu je mapa. Zřejmě nemůžeme od mapy očekávat, aby obsahovala všechny aspekty zahrnuté ve skutečném světě. Svým způsobem je mapa již od počátku „špatně“, nabízí pouze jistou abstrakci systému a může i zkreslovat náš pohled. Přesto nelze popírat její užitečnost. Z příkladu mapy je také zřejmé, že se nemodeluje systém, ale problém. Existuje celá řada druhů map od turistických, automap až po mapy podloží a každá z nich má svůj specifický účel [29, str. 47 – 58].

Pro tuto práci jsou důležité modely, které lze simulovat. Model definuje pravidla, podle kterých se systém chová a simulace umožňuje podívat se na chování systému v čase, ať už diskrétním či spojitém. Pro účely simulace je samozřejmě potřeba znát stav systému v počátečním čase, od kterého se jeho další chování odvíjí.

2.1.1 Modelování pomocí obyčejných diferenciálních rovnic

Hojně užívaným způsobem modelování, kde vystupuje spojitý čas, jsou obyčejné diferenciální rovnice. Stav systému se vyjádří pomocí stavových

proměnných $\mathbf{X} = (x_1, x_2, \dots x_n)$. Každé stavové proměnné přísluší diferenciální rovnice prvního řádu, ve které vystupuje Lipschitzovsky spojitá [12, str. 149 – 163] funkce $f_i : [t, \infty) \times \mathbb{R}^n \to \mathbb{R}$, která popisuje, jak se hodnota stavové proměnné mění v čase. Tvar takové rovnice je vidět ve schématu 2.1. Celý systém označujeme zkráceně funkcí f danou předpisem 2.2.

$$\frac{dx_i}{dt} = f_i(t, \mathbf{X}) \tag{2.1}$$

$$f(t, \mathbf{X}) = \big(f_0(t, \mathbf{X}), \dots, f_n(t, \mathbf{X})\big) \tag{2.2}$$

Pro účely simulace není nutné znát úplné řešení této soustavy rovnic, ale postačuje pouze znalost vývoje systému od počátečního času t_0, kterému odpovídá počáteční stav $\mathbf{X}(t_0)$. V praxi se setkáváme s tím, že neznáme ani tento přesný vývoj, nýbrž pouze jeho aproximaci, kterou poskytují metody pro řešení problému výchozích podmínek [17].

Tyto metody hledají aproximaci v diskrétním čase a chyba, s níž se vypočítaná aproximace liší od skutečného řešení, je shora ohraničená uživatelem danou hodnotou. Fakt, že si může uživatel takto zvolit toleranci chyby, je jednou z nejdůležitějších vlastností těchto metod. Nastavení chyby může samozřejmě v případě nízké tolerance a některých systémů vyústit ve výkonnostní problémy.

Jestliže je dána numerická metoda $\mathcal{M}_\epsilon(f, \mathbf{X}(t_0), \Delta t)$, kde ϵ je relativní chyba a Δt požadovaný časový krok, pak zpravidla pracujeme se sekvencí bodů danou předpisem 2.3, ve kterém τ představuje délku numerické simulace.

$$\begin{aligned} &\mathcal{M}^\tau_\epsilon(f, \mathbf{X}(t_0), \Delta t) = (\mathbf{X}_0, \mathbf{X}_1, \dots \mathbf{X}_k), \\ &\quad \text{kde} \quad \mathbf{X}_i \sim \mathbf{X}(t_i), t_i = t_0 + i \cdot \Delta t, \\ &\qquad\qquad k \cdot \Delta t \leq \tau \wedge (k+1) \cdot \Delta t > \tau \end{aligned} \tag{2.3}$$

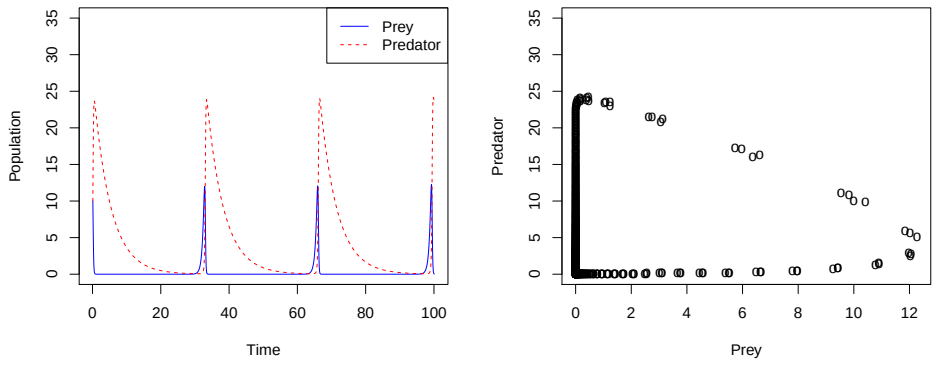
2.1.2 Příklad modelu

Známým příkladem modelu využívajícího soustavy diferenciálních rovnic je model popisující vztah predátora a kořisti [26] definovaný rovnicemi 2.4. Obsahuje stavové proměnné pro množství kořisti (x) a počet predátorů (y), dále parametry pro přirozený přírůstek kořisti (α), „žravosť“ predátorů (β), přirozený úbytek predátorů (γ) a schopnost reprodukce predátorů (δ).

$$\begin{aligned} \tfrac{dx}{dt} &= x \cdot (\alpha - \beta \cdot y) \\ \tfrac{dy}{dt} &= -y \cdot (\gamma - \delta \cdot x) \end{aligned} \tag{2.4}$$

Model je samozřejmě zjednodušením reality. V systému se nachází pouze dva druhy zvířat, býložravá kořist a masožravý predátor. U kořisti se předpokládá bezproblémový přístup k potravě, a proto přirozeně přibývá. Naproti tomu reprodukce predátora je závislá na přísunu masité potravy, tzn. množství kořisti v systému.

I na tomto jednoduchém modelu je však možné pozorovat netriviální chování, o čemž se lze přesvědčit při pohledu na grafy 2.1(a) a 2.1(b). Systém má tendenci oscilovat. Periodicky dochází k nárůstu populace predátora, to vyústí v pokles populace kořisti, následně u predátorů v důsledku nedostatku potravy převáží úmrtnost nad rozsahem reprodukce, množství kořisti opět naroste a tento cyklus se znovu opakuje.



(a) vývoj systému v čase

(b) stavy, kterými systém prochází

Obrázek 2.1: Model systému obsahující predátora a kořist s ohodnocením parametrů $\alpha = 2$, $\beta = 0.5$, $\gamma = 0.2$ a $\delta = 0.6$.

Síla modelů a simulace se ukazuje být v tom, že tento druh informace získáme, aniž bychom prováděli experiment se skutečným systémem se skutečnými liškami a zajíci. Samozřejmě se na závěr neobejdeme bez validace výsledků simulace oproti realitě, ale tomu může předcházet velké množství experimentů na počítači, které by v reálných podmínkách stály mnoho prostředků nebo by ani nebyly uskutečnitelné. Za zmínku stojí například model popisující šíření nákazy populací [20], což je téma, u něhož si opravdu lze jen těžko představit experimentování na reálném systému.

2.1.3 Modelování chemických reakcí

Modelování pomocí obyčejných diferenciálních rovnic je natolik obecný a účinný nástroj, že jej lze použít pro popis jevů z mnoha oblastí. Jednou z typických oblastí, kde se rovnice používají, jsou chemické reakce. Pro systém elementárních chemických reakcí lze za jistých předpokladů automatizovaně zkonstruovat systém diferenciálních rovnic, kde stavovými proměnnými jsou koncentrace jednotlivých látek. Elementárními chemickými reakcemi se rozumí ty chemické reakce, u nichž dochází k přímé přeměně reaktantů v produkty bez reakčních mezikroků nebo v jejichž případě je možné tyto mezikroky zanedbat [15]. Schémata 2.5 ukazují, jak tento převod konkrétně vypadá pro jednotlivé elementární chemické reakce a specifickou kinetickou konstantu k. Koncentraci látky X značíme $[\mathrm{X}]$.

$$\begin{array}{ll}
\xrightarrow{k} A & \rightsquigarrow \frac{d[\mathrm{A}]}{dt} = k \\
A \xrightarrow{k} & \rightsquigarrow \frac{d[\mathrm{A}]}{dt} = -k \cdot [\mathrm{A}] \\
A \xrightarrow{k} B & \rightsquigarrow \frac{d[\mathrm{A}]}{dt} = -k \cdot [\mathrm{A}], \frac{d[\mathrm{B}]}{dt} = k \cdot [\mathrm{A}] \\
AB \xrightarrow{k} A + B & \rightsquigarrow \frac{d[\mathrm{AB}]}{dt} = -k \cdot [\mathrm{AB}], \frac{d[\mathrm{A}]}{dt} = \frac{d[\mathrm{B}]}{dt} = k \cdot [\mathrm{A}] \\
A + B \xrightarrow{k} AB & \rightsquigarrow \frac{d[\mathrm{A}]}{dt} = \frac{d[\mathrm{B}]}{dt} = -k \cdot [\mathrm{A}] \cdot [\mathrm{B}], \frac{d[\mathrm{AB}]}{dt} = k \cdot [\mathrm{A}] \cdot [\mathrm{B}] \\
A + A \xrightarrow{k} AA & \rightsquigarrow \frac{d[\mathrm{A}]}{dt} = -2k \cdot [\mathrm{A}]^2, \frac{d[\mathrm{AB}]}{dt} = k \cdot [\mathrm{A}]^2
\end{array} \tag{2.5}$$

Pro obecné reakce abstrahující určitou kaskádu elementárních reakcí univerzální převod neexistuje. Zde je již nutné přihlédnout k typu reakce. V důsledku toho, že se zanedbají mezireakce s meziprodukty, může výsledný systém diferenciálních rovnic obsahovat méně proměnných a je tak snazší jej simulovat.

Je vhodné poznamenat, že nástroj, který popisuje tato práce v kapitole 4, načítá model zapsaný v jazyce SBML [16, 9]. Tento jazyk představuje standard v oboru systémové biologie pro sdílení modelů a lze z něj automatizovaným způsobem získat model ve formě systému diferenciálních rovnic.

2.2 Vlastnosti modelovaných systémů

Aby bylo možné modely automatizovaně analyzovat, je nutné vyjadřovat se o jejich vlastnostech exaktně. Jazyk, který je k tomuto účely nutné použít, musí být schopen popsat chování systému v čase. V kontextu obyčejných diferenciálních rovnic je chováním trajektorie, v případě simulace sekvence bodů s časovým razítkem. Například u modelu uvedeného v sekci 2.1.2 je vhodné popsat oscilaci populace kořisti nebo predátora. To lze provést

tak, že budeme požadovat, aby množství jedinců daného druhu opakovaně přesáhlo maximální a minimální mez. Jak konkrétně tento požadavek zformulovat, je ukázáno v následující části této kapitoly.

K vyjádření vlastností nad lineárními běhy systémů se nejčastěji používá lineární temporální logika [32] (*linear temporal logic*, LTL), případně logiky z ní odvozené. LTL umožňuje se zjednodušeně vyjadřovat o stavech systému v čase formulacemi typu „v budoucnu nastane ...“, „vždy platí ...“ apod. Tato logika se definuje nad nekonečnými běhy, a proto je v této práci použita temporální logika signálů [27] (*signal temporal logic*, STL) založená na temporální logice metrických intervalů [3] (*metric interval temporal logic*, MITL), která se od LTL liší zejména přidáním časových intervalů u temporálních operátorů. STL tedy umožňuje formulovat výroky, které jsou částečně závislé na čase, jako je například „za hodinu až dvě nastane ...“ či „za třicet minut bude celé čtyři hodiny platit ...“.

2.2.1 Signál

Pro účely vyjadřování se o chování modelu pomocí STL se zavádí pojem signálu [27]. Zvolme si časovou doménu $\mathbb{T} = \mathbb{R}_{\geq 0}$ a signál konečné délky nad libovolnou doménou $\mathbb{D}$ jako parciální zobrazení $s : \mathbb{T} \to \mathbb{D}$. Definičním oborem tohoto zobrazení nechť je interval $l = [0, r)$, kde $r \in \mathbb{Q}_{\geq 0}$ nazýváme délkou signálu a značíme ji $|s| = r$. Pro všechna $t \geq r$ položíme $s[t] = \perp$. Takto definované signály lze sdružovat pomocí párovací funkce $||$.

$$\begin{aligned} &s_1 : \mathbb{T} \to \mathbb{D}_1, s_2 : \mathbb{T} \to \mathbb{D}_2 \\ &s_1||s_2 : \mathbb{T} \to \mathbb{D}_1 \times \mathbb{D}_2 \\ &s_1||s_2 = s_{12}, \text{kde } \forall t \in \mathbb{T}.s_{12}[t] = (s_1[t] \times s_2[t]) \end{aligned} \tag{2.6}$$

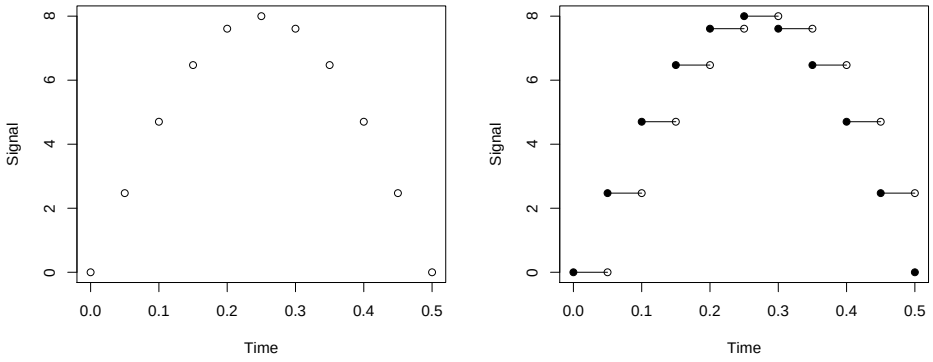
Pro případ, že se délky skládaných signálů liší, definujeme výslednou délku složeného signálu jako $|s_{12}| = min(|s_1|, |s_2|)$. Standardní cestou lze na těchto sdružených signálech definovat projekční funkce.

$$s_1 = \pi_1(s_{12}) \quad s_2 = \pi_2(s_{12}) \tag{2.7}$$

Pro libovolnou funkci $f : \mathbb{D} \to \mathbb{D}'$ a signál $s : \mathbb{T} \to \mathbb{D}$ je zápisem $f(s_1)$ myšleno skládání funkcí $f \circ s_1 : \mathbb{T} \to \mathbb{D}'$, kde $f(\perp) = \perp$.

Je dobré si povšimnout, že definice signálu je konzistentní s tím, jak chápeme chování modelu, tedy jako trajektorii v $\mathbb{R}^n$, kde n je počet stavových proměnných. Zároveň je však třeba si uvědomit, že výstupem nume-

rické simulace není spojitá trajektorie, nýbrž pouze sekvence bodů s časovým razítkem. Z praktických důvodů je dále v této kapitole tato sekvence chápána jako po částech konstantní funkce.



(a) nasimulovaná data (b) po částech konstantní funkce

Obrázek 2.2: Příklad převodu sekvence bodů na po částech konstantní funkci.

2.2.2 Temporální logika signálů

Důležitým aspektem zde použité logiky jsou uzavřené časové intervaly $I = [a, b]$, kde $a, b \in \mathbb{Q}_{\geq 0}$, jimiž jsou omezeny všechny temporální operátory. Konečnost intervalů je jedním z rozdílů oproti klasické temporální logice metrických intervalů. Toto omezení je plně v souladu s tím, že modely porovnáváme s reálnými systémy, které pozorujeme konečný čas. Tento předpoklad značně zjednodušuje další analýzu.

Nechť $U = \{\mu_1, \mu_2, \mu_3, \ldots, \mu_k\}$ je množina efektivně vyčíslitelných funkcí $\mu_i : \mathbb{R}^n \to \{T, F\}$, které zpravidla odpovídají predikátům tvaru $f(\mathbf{X}) \geq k$ nebo $f(\mathbf{X}) \leq k$. Všimněme si, že nemá smysl v predikátech používat samotnou rovnost, protože numerická simulace vrací data s určitou chybou. K těmto vyčíslitelným funkcím přísluší složený $s = \mu_1(x)|| \ldots ||\mu_k(x)$, indexy skládaných signálů představují atomické propozice p.

Gramatiku temporální logiky signálů definujeme podle předpisu 2.8, ve kterém p značí atomickou propozici.

$$\varphi := T \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 \tag{2.8}$$

Ze základních formulí lze odvodit další standardně používané výrokové a temporální operátory. Nejpoužívanějšími jsou výrokový operátor $\vee$ a temporální operátory $\mathcal{F}$ a $\mathcal{G}$, které intuitivně odpovídají už zmíněným výrokům „v budoucnu nastane ...“ a „vždy platí ...“.

$$\begin{array}{ll} \varphi_1 \vee \varphi_2 & \equiv \neg\varphi_1 \wedge \neg\varphi_2 \\ \mathcal{F}_{[a,b]}\varphi & \equiv T\mathcal{U}_{[a,b]}\varphi \\ \mathcal{G}_{[a,b]}\varphi & \equiv \neg\mathcal{F}_{[a,b]}\neg\varphi \end{array} \tag{2.9}$$

Relace $(s,t) \models \varphi$ značí, že daný signál s splňuje formuli φ počínaje pozicí v čase t, a je definována induktivně předpisem 3.7. Signál s splňuje formuli φ právě tehdy, když $(s,0) \models \varphi$.

$$\begin{array}{ll} (s,t) \models p & \iff \pi_p(s)[t] = T \\ (s,t) \models \neg\varphi & \iff (s,t) \not\models \varphi \\ (s,t) \models \varphi_1 \wedge \varphi_2 & \iff (s,t) \models \varphi_1 \text{ a současně } (s,t) \models \varphi_2 \\ (s,t) \models \varphi_1 \mathcal{U}_{[a,b]} \varphi_2 & \iff \exists t' \in [t+a, t+b].(s,t') \models \varphi_2 \\ & \quad \text{a současně } \forall t'' \in [t,t'].(s,t'') \models \varphi_1 \end{array} \tag{2.10}$$

Operátor $\mathcal{U}$ je zde definován s trochu jinou sémantikou, než je běžné. Požaduje se zde silnější podmínka – aby existoval stav (čas), pro který platí obě vlastnosti φ_1 a φ_2, tedy aby existoval čas $t' \in [t+a, t+b]$ takový, že $(s,t') \models \varphi_1$ a současně $(s,t') \models \varphi_2$. To však nemění sémantiku ostatních známých temporálních operátorů $\mathcal{F}$ a $\mathcal{G}$.

$$\begin{array}{ll} (s,t) \models \mathcal{F}_{[a,b]}\varphi & \longleftrightarrow \exists t' \in [t+a, t+b].(s,t') \models \varphi \\ (s,t) \models \mathcal{G}_{[a,b]}\varphi & \longleftrightarrow \forall t' \in [t+a, t+b].(s,t') \models \varphi \end{array} \tag{2.11}$$

Standardní sémantika temporálních logik obecně neumožňuje ověření platnosti temporálních operátorů na konečných signálech kromě některých případů, jako je splněnost $\mathcal{F}\varphi$ nebo nesplněnost $\mathcal{G}\varphi$, jejichž platnost může být ověřena v konečném čase. Tento problém naštěstí odpadá zavedením časových intervalů. I přesto však existují formule a signály, u kterých o platnosti rozhodnout nelze. Příkladem může být formule $F_{[a,b]}\varphi$ a signál o délce kratší než b.

Z tohoto důvodu se definuje délka nezbytná k ověření platnosti dané formule nad daným signálem, opět induktivně.

$$\begin{array}{ll} ||p|| & = 0 \\ ||\neg\varphi|| & = ||\varphi|| \\ ||\phi_1 \wedge \phi_2|| & = max(||\varphi_2||, ||\varphi_2||) \\ ||\phi_1 \mathcal{U}_{[a,b]} \phi_2|| & = max(||\varphi_2||, ||\varphi_2||) + b \end{array} \tag{2.12}$$

2.2.3 Příklad vlastnosti

Temporální logika signálů umožňuje formulovat celou řadu vlastností. Mezi ty nejznámější a nejčastěji používané patří oscilace. Je dobré si uvědomit, že oscilaci lze chápat mnoha různými způsoby. V praxi se nevyplatí klást přísná omezení na přesnost periody, velikost amplitudy či přesnost stavu, kterým systém periodicky prochází.

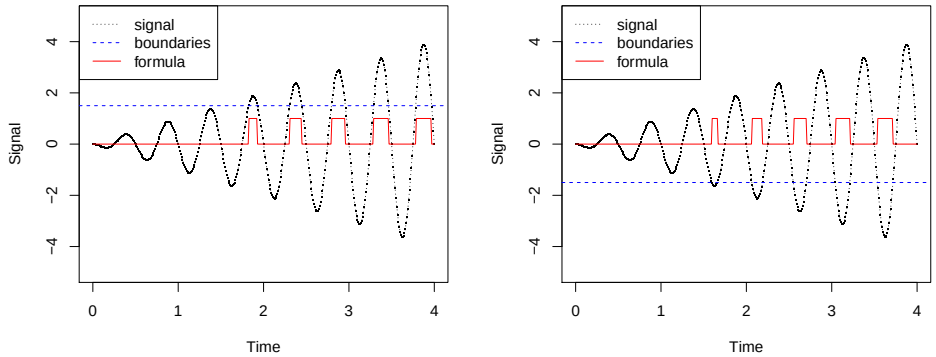
Představme si systém o jedné stavové proměnné x, ve které se vzrůstající intenzitou osciluje hodnota této proměnné. Perioda oscilace je konstantní, avšak její amplituda ne. V této práci použitá logika se vyjadřuje o hodnotách proměnné x, tudíž oscilaci budeme chápat jako periodické překračování dolní a horní meze. Atomickými propozicemi jsou tedy predikáty $[x] \geq k$ a $[x] \leq -k$, jejichž platnost lze vidět na obrázcích 2.3(a) a 2.3(b).

Jeden cyklus lze popsat tak, že se hodnota sledované proměnné nachází nad horní mezí a zároveň někdy v budoucnu klesne pod dolní mez, tedy $[x] \geq k \wedge \mathcal{F}_{[0,\frac{1}{2}]}[x] \leq -k$. Přidání operátoru $\mathcal{F}$ zajistí, že se v daném času do určité doby provede jeden oscilační cyklus, viz obrázek 2.3(d). To, že systém osciluje znamená, že toto platí pro každý časový okamžik, což vyjádříme operátorem $\mathcal{G}$.

Ve zde uvedeném příkladu systém neosciluje s požadovanou amplitudou již od začátku, tudíž je potřeba do formule přidat ještě nějaké čekání v podobě operátoru $\mathcal{F}$. Výsledná formule tedy vypadá následovně:

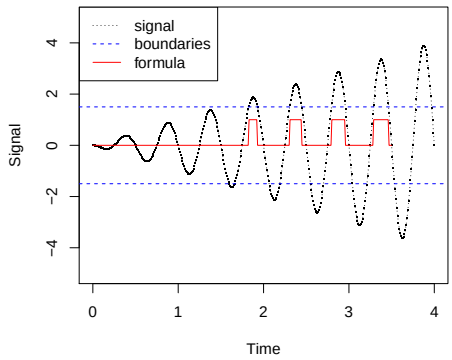
$$\mathcal{F}_{[0,\text{čekání}]}\mathcal{G}_{[0,\text{doba oscilace}]}\mathcal{F}_{[0,\text{perioda}]}\Big([x] \geq k \wedge \mathcal{F}_{[0,\text{perioda}]}[x] \leq -k\Big) \tag{2.13}$$

Z uvedeného příkladu je zřejmě, že platnost formule nad daným signálem lze ověřit algoritmem, jehož průběh bude kopírovat strukturu formule. Jak přesně ověřovací algoritmus vypadá, ukáže sekce 3.3, která navíc lehce rozšíří chápání pravdivosti jako takové.

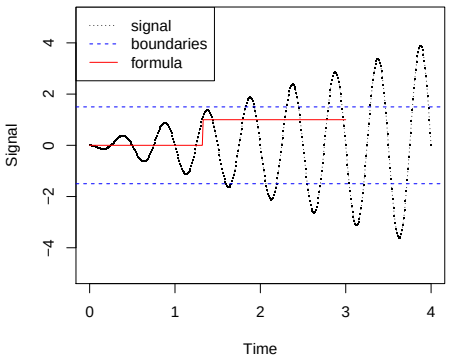


(a) $[x] \geq k$

(b) $[x] \leq -k$



(c) $[x] \geq k \wedge \mathcal{F}_{[0,\frac{1}{2}]}[x] \leq -k$



(d) $\mathcal{F}_{0,\frac{1}{2}}([x] \geq k \wedge \mathcal{F}_{[0,\frac{1}{2}]}[x] \leq -k)$

Obrázek 2.3: Pravdivostní hodnota atomických propozic v čase.

Kapitola 3

Algoritmus pro analýzu dynamických systémů

Tato kapitola ukáže jednu z možností, jak přistoupit k analýze modelů zadaných pomocí obyčejných diferenciálních rovnic. Zde uvedená analýza se snaží řešit následující problémy:

1. Máme k dispozici již hotový model, jehož chování odpovídá chování skutečného systému pro jedno konkrétní nastavení iniciálních hodnot stavovým proměnným. Splňuje tento model požadované vlastnosti i pro jiná nastavení?

2. Máme kostru modelu, v němž se vyskytuje několik parametrů, jejichž hodnota není známá. Jak nastavit parametry modelu tak, aby splňoval dané chování [4]?

V bodě 1 lze jít dál než k pouhé kontrole modelů. Můžeme si představit poměrně přesný model, který využijeme k analýze jeho chování v podmínkách, které nelze navodit u reálného systému. Typickým příkladem může být živý organismus v toxickém prostředí nebo extrémně vysoká nákaza šířící se celosvětovou populací.

3.1 Definice problému

Nechť je dynamický systém $\mathcal{DS} = (\mathbf{X}, f(\mathbf{P}))$, kde $\mathbf{X} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ je vektor stavových proměnných a $\mathbf{P} = (p_1, \ldots, p_m) \in \mathbb{R}^m$ je vektor parametrů. Dynamiku systému popisují obyčejné diferenciální rovnice $\frac{dx_i}{dt} = f_i(\mathbf{P})(\mathbf{X})$ a $f_i : \mathbb{R}^m \to (\mathbb{R}^n \to \mathbb{R}^n)$. Tento systém rovnic souhrnně označíme jednou rovnicí $\frac{d\mathbf{X}}{dt} = f(\mathbf{P})(\mathbf{X})$. Na rozdíl od obecného modelu zadaného pomocí systému obyčejných diferenciálních rovnic uvažujeme pouze autonomní systémy, tedy budeme předpokládat, že funkce stojící na pravé straně nezávisí na čase, a proto je čas z definicí funkcí zcela vynechán.

Pro každou stavovou proměnnou x_i je dán interval $\mathcal{I}_{x_i} = [\iota_{x_i}^{min}, \iota_{x_i}^{max}]$ a pro každý parametr p_j interval $\mathcal{I}_{p_j} = [\iota_{p_j}^{min}, \iota_{p_j}^{max}]$. Tyto intervaly omezují

nastavení iniciálních hodnot stavových proměnných x_i, respektive ohodnocení parametrů p_j, a určují tak prostor iniciálních podmínek [10, str. 23].

$$\mathcal{I} = \mathcal{I}_{x_1} \times \mathcal{I}_{x_n} \times \ldots \times \mathcal{I}_{p_1} \times \ldots \times \mathcal{I}_{p_m} \tag{3.1}$$

Pro vektor $\mathbf{V} = (x_1, \ldots, x_n, p_1, \ldots, p_m) \in \mathcal{I}$ definujme projekční funkce $\pi_x(\mathbf{V}) = (x_1, \ldots, x_n)$ a $\pi_p(\mathbf{V}) = (p_1, \ldots, p_m)$. Je-li dána numerická metoda $\mathcal{M}_\varepsilon$, formule temporální logiky signálů φ, dynamický systém $\mathcal{DS}$ a prostor iniciálních podmínek $\mathcal{I}$, pak řešeným problémem je najít části prostoru $\mathcal{S}, \mathcal{N} \subseteq \mathcal{I}$ takové, že platí vztah 3.2 [10, str. 23].

$$\begin{aligned} &\forall \mathbf{V} \in \mathcal{S}.\mathcal{M}_\varepsilon^\tau(f(\mathbf{P}), \mathbf{X}, \Delta t) \models \varphi, \text{ kde } \mathbf{P} = \pi_p(\mathbf{V}), \mathbf{X} = \pi_x(\mathbf{V}) \\ &\forall \mathbf{V} \in \mathcal{N}.\mathcal{M}_\varepsilon^\tau(f(\mathbf{P}), \mathbf{X}, \Delta t) \not\models \varphi, \text{ kde } \mathbf{P} = \pi_p(\mathbf{V}), \mathbf{X} = \pi_x(\mathbf{V}) \end{aligned} \tag{3.2}$$

Tyto části prostoru popisují ohodnocení počátečních stavů a parametrů, za kterých se daný systém vyvíjí s požadovanou vlastností φ, respektive bez požadované vlastnosti. Naivní algoritmus řešící tento problém do prostoru iniciálních podmínek vloží určité množství bodů a tyto body použije pro simulaci chování modelu, nad kterým se následně provede ověření vlastnosti. Počet bodů, tedy míra zahuštění prostoru iniciálních podmínek, závisí na požadované přesnosti analýzy. I přes nesporné výhody tohoto přístupu snadno narazíme na výpočetní limity potřebného množství bodů, a proto je vhodné pokusit se počet bodů omezit.

3.2 Původní algoritmus

Původní algoritmus, na kterém staví tato práce vychází z velice důležitého předpokladu: „Většina řešení začínajících v iniciálních bodech blízko sebe zůstávají blízko sebe i v průběhu času [10, str. 25].“ Předpokládá se tedy, že chování určená blízkými hodnotami z prostoru iniciálních podmínek mají i blízkou míru platnosti dané formule. Není tedy třeba zjišťovat chování pro všechny body prostoru iniciálních podmínek, ale jen pro určitou množinu reprezentantů, pro kterou platí [10, str. 25]:

1. Chování blízké reprezentantovi zůstane blízké na celém časovém intervalu, na kterém je daná formule ověřována.

2. Množina reprezentantů pokrývá celý prostor iniciálních podmínek.

Algoritmus do prostoru iniciálních podmínek vkládá body tak dlouho, dokud si trajektorie chování určených blízkými body jsou vzdálenější než daná vzdálenost δ. Nad simulovanými chováními se ověří platnost formule. Výsledkem je určité množství bodů, u kterých dostáváme informace, zda z nich simulovaná chování splňují či nesplňují danou vlastnost. Tyto body nastíní hranice regionů platnosti a neplatnosti se zvolenou přesností δ.

Pseudokód 1 Analýza prostoru iniciálních podmínek

Vstup: $\mathcal{DS} = (\mathbf{X}, f), \mathcal{I}, \varphi, \Delta t, \delta, \varepsilon$
Výstup: RESULT $= \{([\mathbf{F}_0]_0, s_1), \ldots ([\mathbf{F}_k]_0, s_k)\}$ ▷ body a splněnost φ

1: $\mathrm{M}_{new} \leftarrow$ počáteční zahuštění $\mathcal{I}$
2: RESULT $\leftarrow \emptyset$
3: **while** $\mathrm{M}_{new} \neq \emptyset$ **do**
4: $\quad \mathrm{M}_{old} \leftarrow \mathrm{M}_{new}, \mathrm{M}_{new} \leftarrow \emptyset$
5: $\quad$ **for** $([\mathbf{X}_{main}]_0, [\mathbf{P}_{main}]) \in \mathrm{M}_{old}$ **do**
6: $\quad\quad$ TRAJ$_{main} \leftarrow \mathcal{M}_{\varepsilon}^{||\varphi||}(f_{\mathbf{P} \leftarrow [\mathbf{P}_{main}]}, [\mathbf{X}_{main}]_0, \Delta t)$
7: $\quad\quad$ SATISFIED$_{main} \leftarrow$ splněnost φ nad TRAJ$_{main}$
8: $\quad\quad$ RESULT $\leftarrow$ RESULT $\cup \{($SATISFIED$_{main}, [\mathbf{X}_{main}]_0, [\mathbf{P}_{main}])\}$
9: $\quad\quad$ NEIGH $\leftarrow$ body sousedící s bodem $([\mathbf{X}]_0, [\mathbf{P}])$
10: $\quad\quad$ **for** $([\mathbf{X}_{neigh}]_0, [\mathbf{P}_{neigh}]) \in$ NEIGH **do**
11: $\quad\quad\quad$ TRAJ$_{neigh} \leftarrow \mathcal{M}_{\varepsilon}^{||\varphi||}(f_{\mathbf{P} \leftarrow [\mathbf{P}_{neigh}]}, [\mathbf{X}_{neigh}]_0, \Delta t)$
12: $\quad\quad\quad$ SATISFIED$_{neigh} \leftarrow$ splněnost φ nad TRAJ$_{neigh}$
13: $\quad\quad\quad$ RESULT $\leftarrow$ RESULT $\cup \{($SATISFIED$_{main}, [\mathbf{X}_{main}]_0, [\mathbf{P}_{main}])\}$
14: $\quad\quad\quad$ DISTANCE $\leftarrow$ vzdálenost TRAJ$_{main}$ a TRAJ$_{neigh}$
15: $\quad\quad\quad$ **if** DISTANCE $> \delta$ **then**
16: $\quad\quad\quad\quad \mathrm{M}_{new} \leftarrow \mathrm{M}_{new} \cup \{(\frac{[\mathbf{X}_{main}]_0 + [\mathbf{X}_{neigh}]_0}{2}, \frac{[\mathbf{P}_{main}] + [\mathbf{P}_{neigh}]}{2})\}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **end for**
20: **end while**

Je samozřejmé otázkou, jakým způsobem konkrétně probíhá počáteční zahuštění prostoru iniciálních podmínek, co přesně obsahuje množina sousedů daného bodu a jak se změří vzdálenost dvou chování. Poslední zmíněnou otázkou se podrobně zabývá diplomová práce Svena Dražana [10]. Zbytek bude ještě rozveden v sekci 3.4 společně s tím, jak zvolit hodnotu δ.

3.3 Robustnost

Platnost formule lze chápat trochu šířeji než prosté „platí“ / „neplatí“. Nemusíme se pouze ptát, zda dané chování splňuje danou formuli, otázku lze posunout dál. Jak moc dané chování splňuje danou formuli? Jak moc je vlastnost nesplněna? Pro účely této práce zmíněné otázky zcela postačují, ale samozřejmě je možné požadovat ještě více. Odpověď lze kvantizovat a získat tak představu o míře, s jakou je daný systém robustní vůči změnám podmínek, teplot nebo koncentrací chemických látek.

V této sekci zavedeme pojem *lokální a globální robustnosti*. Lokální robustností rozumíme míru, do jaké je daná vlastnost splněna na jednom chování. Globální robustnost na druhé straně vztáhneme na celý systém, a pak zahrnuje míru platnosti dané vlastnosti nad větším množství chování, která vzniknou tzv. *perturbacemi*. Existuje jedno referenční chování za ideálních podmínek, a pak mnoho perturbovaných chování za podmínek ne tak ideálních. Perturbace lze chápat různě, v této práci odpovídají prostoru iniciálních podmínek definovaném v sekci 3.1.

Jednou z věcí, které je třeba předem zmínit, je, že pro účely vylepšení algoritmu pro analýzu dynamických systémů, se na robustnost díváme z pohledu chování systému [8]. To znamená, že se při výpočtu lokální robustnosti snažíme ohraničit prostor okolo jednoho chování, ve kterém má daná vlastnost stejnou platnost. Podobně lze k problému přistoupit z opačné strany [31]. Ve formuli jsme schopni identifikovat volné proměnné, napočítat podprostor prostoru ohodnocení těchto proměnných, ve kterém je formule pro dané chování splněna, a určit vzdálenost již konkrétní formule s dosazenými hodnotami za volné proměnné od tohoto napočítaného podprostoru.

3.3.1 Lokální robustnost

Nechť s_1, s_2 jsou signály nad doménou $\mathbb{D}$ a $d : \mathbb{D}^n \to \mathbb{R}^+$ funkce určující vzdálenost dvou bodů v prostoru $\mathbb{R}^n$. Vzdálenost dvou signálů zavedeme předpisem 3.3.

$$\sigma(s_1, s_2) = \sup_{t \in \mathbb{R}^+} \{d(s_1(t), s_2(t))\} \tag{3.3}$$

Od robustnosti ρ budeme požadovat konzistentní chování s již zavedenou dvouhodnotovou platností formule.

$$\rho(\varphi, s) = \begin{cases} \inf\{\sigma(s, s') | \forall s'.s' \not\models \varphi\} & \text{pokud } s \models \varphi \\ -\inf\{\sigma(s, s') | \forall s'.s' \models \varphi\} & \text{pokud } s \not\models \varphi \end{cases} \tag{3.4}$$

Oborem hodnot funkcí μ_i odpovídajících atomickým propozicím tentokrát není množina $\{T, F\}$, nýbrž množina reálných čísel $\mathbb{R}$. Toto chápání je spíše technického charakteru. U nejčastěji používaných atomických propozic, kde $f(\mathbf{X}) = x_i$, je převod z chápání popsaného v sekci 2.2.2 přímočarý.

$$\begin{aligned} \mu_i = f(\mathbf{X}) \geq k &\rightsquigarrow \mu_i = f(\mathbf{X}) - k \\ \mu_i = f(\mathbf{X}) \leq k &\rightsquigarrow \mu_i = k - f(\mathbf{X}) \end{aligned} \tag{3.5}$$

Podobně jako při výpočtu dvouhodnotové platnosti formule i u robustnosti je potřeba odkazovat se na robustnost v určitém čase $\rho(\varphi, s, t)$, která se definuje induktivně ke struktuře formule. Výslednou robustnost dostaneme položením $\rho(\varphi, s) = \rho(\varphi, s, 0)$.

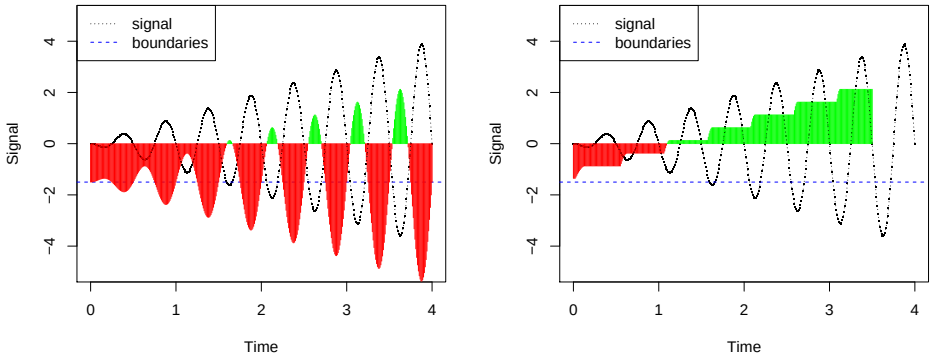
$$\begin{aligned} \rho(p, s, t) &= \pi_p(s)[t] \\ \rho(\neg\varphi, s, t) &= -\rho(\varphi, s, t) \\ \rho(\varphi_1 \wedge \varphi_2, s, t) &= \min\Big(\rho(\varphi_1, s, t), \rho(\varphi_1, s, t)\Big) \\ \rho(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2, s, t) &= \max_{t' \in [t+a, t+b]} min\Big(\rho(\varphi_2, s, t'), \min_{t'' \in [t, t']} \rho(\varphi_1, s, t'')\Big) \end{aligned} \tag{3.6}$$

Pro odvozené temporální operátory $\mathcal{F}$ a $\mathcal{G}$ lze definovat robustnost samostatně, což umožňuje naimplementovat její efektivnější výpočet.

$$\begin{aligned} \rho(\mathcal{F}_{[a,b]}\varphi, s, t) &= \max_{t' \in [t+a, t+b]} \Big(\rho(\varphi, s, t')\Big) \\ \rho(\mathcal{G}_{[a,b]}\varphi, s, t) &= \min_{t' \in [t+a, t+b]} \Big(\rho(\varphi, s, t')\Big) \end{aligned} \tag{3.7}$$

Jestliže je k dispozici analyzovaný primární signál, lze použít robustnost pro definici tzv. sekundárních signálů příslušejících každé podformuli φ' dané formule φ, kde $s_{\varphi'}[t] = \rho(\varphi', s, t)$. a $|s_{\varphi'}| = |s| - ||\varphi'||$. Tyto sekundární signály dávají informaci o tom, jak moc se lze od primárního signálu vzdálit, aby daná podformule zůstala ještě platná v případě kladných hodnot sekundárních signálů nebo neplatná v případě hodnot záporných. Krajní hodnotou je signál nulový, který značí hranici platnosti formule.

Vraťme se k příkladu formule popisující oscilaci systému s jednou stavovou proměnnou ze sekce 2.2.3. Ukážeme si, jak vypadá sekundární signál pro podformuli $\mathcal{F}_{[0,\frac{1}{2}]}[x] \leq -k$.



(a) $[x] \leq -k$ (b) $\mathcal{F}_{[0,\frac{1}{2}]}[x] \leq -k$

Obrázek 3.1: Znázornění sekundárních signálů pro některé z podformulí vlastnosti oscilace. Zelené podbarvení značí platnost dané formule v daném čase, červené naopak neplatnost.

3.3.2 Výpočet lokální robustnosti

Za zmínku stojí způsob, jakým lze výpočet robustnosti naimplementovat. Od nejvíce zanořených podformulí se počítají sekundární signály, které se použijí pro výpočet sekundárních signálů pro nadřazené podformule. Není překvapením, že sekundární signál pro základní predikátové operátory $\neg$ a $\wedge$ lze spočítat v lineárním čase vzhledem k délce signálu.

Pro hledání maxima, respektive minima pro všechny podsekvence dané sekvence prvků existuje algoritmus s lineární časovou složitostí [24]. Díky tomu lze výpočet sekundárního signálu i odvozených operátorů $\mathcal{F}$ a $\mathcal{G}$ provést rovněž v lineárním čase. Použitím pomocné funkce, jejíž struktura je popsaná v pseudokódu 3, získáme předpis

$$\begin{aligned} s_{\mathcal{F}_{[a,b]}\varphi} &= \textsc{Signal}([a,b],\varphi,>) \\ s_{\mathcal{G}_{[a,b]}\varphi} &= \textsc{Signal}([a,b],\varphi,<) \end{aligned} \tag{3.8}$$

Pro výpočet robustnosti formule $\varphi \mathcal{U}_{[a,b]} \psi$ lze použít vztahu 3.9 [7], který umožňuje formuli ve výpočtu nahradit konjunkcí několika podformulí, pro něž lze robustnost spočítat v lineárním čase. Zejména to platí pro $\varphi \mathcal{U} \psi$ bez požadavku na dodržení intervalu $[a, b]$. Výpočet robustnosti pro operátor $\mathcal{U}$ s omezením na interval $[a, b]$ má proto rovněž lineární časovou složitost [7].

Pseudokód 2 datová struktura LEMIRE-QUEUE[24]

Vstup: $\prec \subseteq \mathbb{R}^2$ ▷ ostré uspořádání

```
deque                                   ▷ fronta s přístupem k oběma koncům
function LEMIRE-QUEUE.OFFER(time, value)
    while ¬deque.ISEMPTY() ∧ value ≺ DEQUEUE.GETLAST().value do
        deque.REMOVELAST()
    end while
    deque.OFFER(time, value)
end function
function LEMIRE-QUEUE.PEEK( )
    return deque.GETFIRST()
end function
function LEMIRE-QUEUE.POLL( )
    return deque.REMOVEFIRST()
end function
```

Pseudokód 3 pomocná funkce pro sekundárního signálu

```
function SIGNAL([a, b], s_φ, ≺)
    queue ← nová LEMIRE-QUEUE s uspořádáním ≺
    monitor ← nové prázdné asociativní pole
    t' ← t_0 + a
    while t' ≤ t_0 + b − Δt do
        queue.OFFER(t', s_φ[t'])
        t' ← t' + Δt
    end while
    t' ← t_0
    while t' ≤ |s| do
        qeue.OFFER(t' + b, s_φ[t' + b])
        monitor[t'] ← queue.PEEK()
        if queue.PEEK().time < t' + a + Δt then
            queue.POLL()
        end if
        t' ← t' + Δt
    end while
    return monitor
end function
```

$$\rho(\varphi \mathcal{U}_{[a,b]} \psi, s, t) = \rho\Big(\mathcal{G}_{[0,a]}\varphi \wedge \mathcal{F}_{[a,b]}\psi \wedge \mathcal{F}_{[a,a]}(\varphi \mathcal{U} \psi)\Big) \tag{3.9}$$

3.3.3 Globální robustnost

Lokální robustnosti lze využít k analýze toho, do jaké míry je model robustní vzhledem k dané vlastnosti jako celek. Je možné si představit systém, u něhož se hodnoty jeho parametrů, případně počátečních hodnot, vychylují, tzv. perturbují, od ideálního stavu. Často požadujeme, aby systém byl schopen do určité velikosti perturbací vykazovat jisté chování.

Mějme systém $\mathcal{S}$, množinu perturbací P s pravděpodobnostmi výskytu a ohodnocovací funkci $D^{\mathcal{S}}_{\varphi}$. Míru, s jakou je systém schopen zachovat vlastnost φ při daných perturbacích, definujeme předpisem 3.10 [21]. Za ohodnocovací funkcí je vhodné zvolit lokální robustnost $D^{\mathcal{S}}_{\varphi} = \rho(\varphi, s_p)$. Nahrazením vzniká předpis 3.11.

$$R^{\mathcal{S}}_{\varphi,P} = \int_{p \in P} prob(p) \cdot D^{\mathcal{S}}_{\varphi}(p) dp \tag{3.10}$$

$$R^{\mathcal{S}}_{\varphi,P} = \int_{p \in P} prob(p) \cdot \rho(\varphi, s_p) dp \tag{3.11}$$

V kontextu ukázaného algoritmu pro analýzu dynamických systémů není globální robustnost až tak důležitý pojem, nicméně nabízí užitečnou metriku k porovnávání modelů.

3.4 Upravený algoritmus

Nyní se vraťme k algoritmu pro analýzu dynamických systémů. V jeho původní podobě vystupuje parametr δ, se kterým se porovnává vzdálenost hlavní trajektorie chování s trajektorií sousední, viz řádek 15 pseudokódu 1. Pro účely této práce je postačující již uvedená definice 3.3 vzdálenosti dvou trajektorií jako $\sigma(s_1, s_2) = \sup_{t \in \mathbb{R}^+} \{d(s_1(t), s_1(t))\}$, nicméně je samozřejmě možné použít jiný předpis. Hodnota parametru δ je pro celý prostor chování stejná, což vede k podobnému zahuštění jak v regionech prostoru iniciálních podmínek, kde daná formule určitě platí, určitě neplatí, tak i v regionech, kde se platnost formule mění.

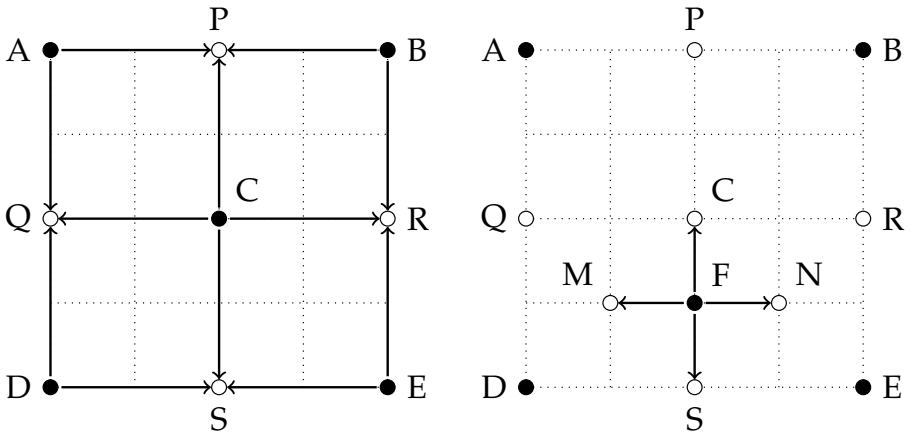
Pro zvýšení efektivity algoritmu je vhodné, aby pro regiony, kde je platnost formule již zřejmá, nedocházelo k dalšímu zahušťování a simulaci

trajektorií chování. Naopak pro regiony, kterými prochází hranice změny platnosti, by k zahušťování docházet mělo. Jak tyto části prostoru od sebe odlišit?

Absolutní hodnotu lokální robustnosti nad danou trajektorií chování lze chápat jako odchylku, o kterou je možné se od trajektorie vzdálit, aby platnost formule byla zachovaná. Nabízí se tak nahrazení konstanty δ na řádku 15 pseudokódu 1 právě tímto číslem.

3.5 Zahušťování

Na závěr této kapitoly je vhodné se ještě zmínit o způsobu, jakým jsou vybírány body pro iniciální zahuštění, případně další body poté, co se nějaká ze sousedních trajektorií chování vzdálila od hlavní trajektorie více, než připouští napočítaná lokální robustnost.



(a) 1. iterace zahuštění (b) 2. iterace zahuštění

Obrázek 3.2: Znázornění průběhu výpočtu pro dvourozměrný prostor iniciálních podmínek. Černé tečky znázorňují hlavní a bílé pomocné body, které odpovídají bodům X_{main} a X_{neigh} v pseudokódu 1. Ke každému bodu X odpovídá signál s_X. Hrana $X \to Y$ značí porovnání $\left|\rho(\varphi, s_X)\right| \geq \rho(s_X, s_Y)$.

Na obrázku 3.2(a) je ukázáno, jak vypadá iniciální zahuštění pro dvourozměrný prostor iniciálních podmínek. Algoritmus vloží jeden hlavní bod (C) do středu prostoru. Tento prostřední bod obklopí dvěma vedlejšími

body v každé dimenzi iniciálního prostoru (P, Q, R, S) ve vzdálenosti poloviny velikosti prostoru v dané dimenzi. Takto vzniklé body opět obklopí dalšími, tentokrát již hlavními (A, B, D, E). Tento postup se opakuje, dokud není prostor iniciálních podmínek dostatečně zahuštěn. Každý z hlavních bodů s sebou během výpočtu nese odkaz na příslušné vedlejší body. Toto iniciální zahuštění je označeno jako první iterace.

Obrázek 3.2(b) znázorňuje situaci, kdy došlo k druhé iteraci zahuštění, protože $\left|\rho(\varphi, s_C)\right| < dist(s_C, s_S)$. Ze středu úsečky CS se vytvoří nový hlavní bod F. Pro každou dimenzi se vytvoří pomocné body tentokrát ve vzdálenosti jedné čtvrtiny velikosti prostoru iniciálních hodnot v dané dimenzi, tím se hlavní bod C v této nové iteraci zahuštění stává bodem vedlejším.

V nejméně příznivém scénáři je v iteracích $0, 1, \ldots, i$ nutno pracovat s $(2^i + 1)^d$ body, kde d je dimenze prostoru iniciálních podmínek. Algoritmus v mnoha případech ke svému výpočtu nevyžaduje takové množství bodů. Exponenciální nárůst počtu potřebných bodů se však i přesto od určité iterace zahuštění objeví. To je způsobeno zejména regiony prostoru iniciálních podmínek, kterými prochází hranice platnosti dané formule.

Nutnost analyzovat exponenciální množství trajektorií chování společně s výpočetní náročností numerické simulace vede k tomu, že analýza i poměrně malých modelů může trvat nesmírně dlouho. Tato práce se snaží tento problém řešit distribucí výpočtu na více počítačích s využitím datového paralelismu.

Kapitola 4

Implementace

Tato kapitola se věnuje aplikaci Parasim, jejíž jsem hlavním spoluautorem. Tato aplikace implementuje algoritmus pro analýzu dynamických systémů zmíněný v sekci 3.4. Aplikace Parasim vznikla na základě prototypu dostupného v diplomové práci Svena Dražana [10]. Cílem prototypu bylo názorně zobrazit průběh výpočtu původního algoritmu. Uživateli se ukazuje, jakým způsobem se počítá vzdálenost mezi trajektoriemi chování, kde je nutné zahušťovat, a jak dopadlo ověření platnosti. Vzhledem k tomu, že prototyp například pro nalezení trajektorií chování používá pouze jednoduchou metodu numerické simulace, není vhodné jej použít k analýze složitějších modelů.

V nové implementaci bylo oproti prototypu třeba zahrnout následující:

1. úprava algoritmu dle sekce 3.4;
2. paralelní výpočet ve sdílené nebo distribuované paměti;
3. rozšiřitelnost, modularita a otevřenost k rozdílné implementaci již naimplementovaných částí;
4. zobrazení výsledků analýzy pro vícedimenzionální prostory iniciálních podmínek.

Tato práce se zabývá všemi zmíněnými body kromě bodu 4, který je však v Parasimu již také vyřešen. Vzhledem k monolitické implementaci prototypu nedošlo k jeho úpravám a rozšíření, ale byla vytvořena od základu nová aplikace. Bod 3 je důležitý z několika důvodů. Pro různé části algoritmu existuje více způsobů, jak je naimplementovat. Existuje například mnoho nástrojů umožňujících numerickou simulací na základě systému diferenciálních rovnic získat trajektorii chování. V případě výpočtu robustnosti se nemusíme omezit pouze na temporální logiku signálů, ale můžeme zavést logiku novou, expresivnější.

Kapitola je členěna do tří hlavních částí. První z nich se věnuje architektuře aplikace, zejména jejímu jádru, které zajišťuje modularitu a základní

služby nezbytné pro načítání a správu dalších rozšíření. Další sekce popisuje výpočetní model, jenž byl vytvořen za účelem sjednocení implementace výpočtu ve sdílené a distribuované paměti. V závěru kapitoly se nachází stručný popis rozšíření použitých v algoritmu pro analýzu dynamických systémů.

4.1 Architektura

Parasim je aplikace napsaná v programovacím jazyku Java verze 7 a skládá se z většího množství artefaktů pro sestavovací nástroj Maven. Zdrojové kódy jsou pod licencí GNU GPL verze 3 [1] k dispozici v Git [5] repozitáři[1]. Artefakty sestavené z poslední verze těchto zdrojových kódů jsou publikovány do Maven repozitáře *snapshot*[2], o což se stará veřejná instance nástroje Jenkins [2][3]. Artefakty vydaných verzí jsou k nalezení v Maven repozitáři *release*[4]. Použití aplikace Parasim dostupné v podobě artefaktu `org.sybila.parasim.application:parasim` je popsáno v příloze A.

Jádro Parasimu tvoří artefakt `org.sybila.parasim:core`, který řídí životní cyklus všech modulů a zajišťuje základní funkcionalitu. To zajišťuje kontejner, jehož instance je vytvořena a spuštěna v rámci aplikace. Kontejner je velkou mírou inspirován konceptem *Context and Dependency Injection* zavedeným v Java EE 6 [18] specifikací JSR-299 [19]. V žádném případě se nejedná o implementaci tohoto standardu, nýbrž jen o volnou inspiraci některými koncepty, které umožňují úplné oddělení rozhraní od implementace, lepší testovatelnost a práci s pamětí.

Nejdůležitějším aspektem je použití kontextů, které mají různou délku a které je možné zanořovat. Rozšíření pro jednotlivé kontexty nabízejí služby. Aplikace definuje na různých místech v kódu závislosti na těchto službách. Kontejner se stará o to, aby v případě požadavku na nějakou službu bylo zavoláno příslušné rozšíření, které ji poskytuje. Rozšíření vytvoří instanci služby a v případě vypršení kontextu tuto instanci vhodným způsobem zničí.

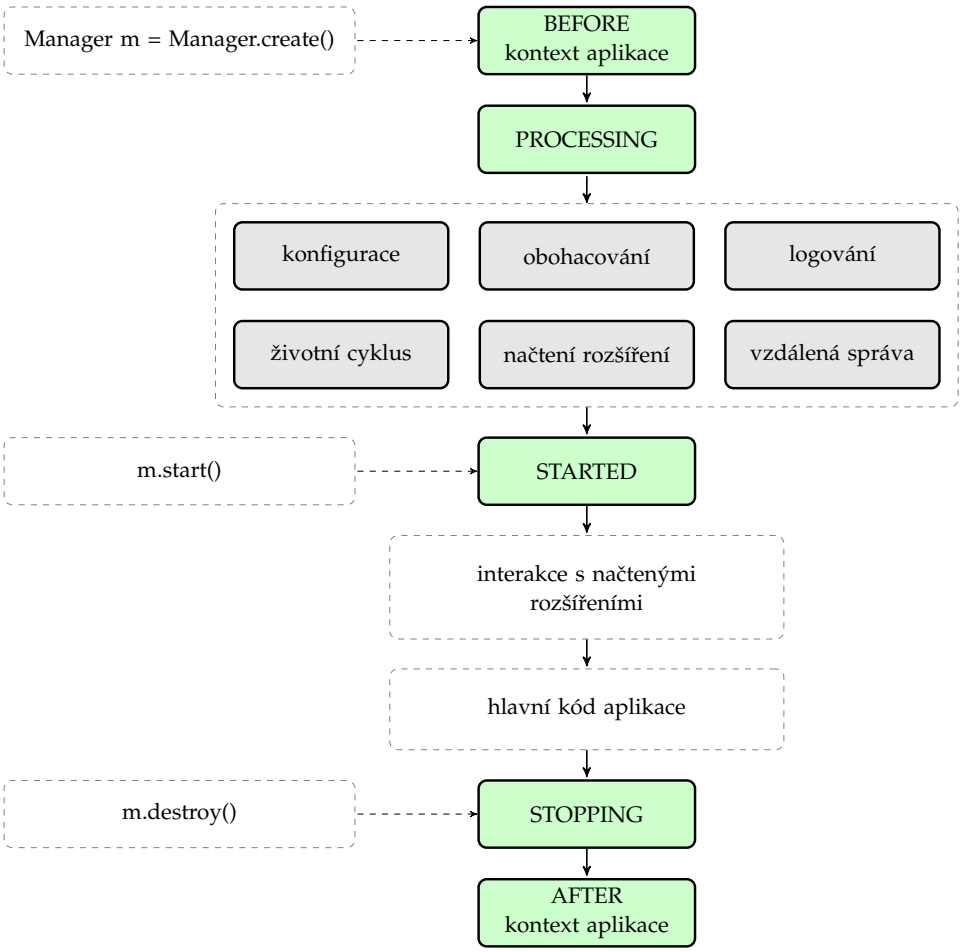
Za vytvoření a zničení instance služby je tak zodpovědný autor rozšíření, který také definuje příslušná rozhraní. Aplikace není téměř žádným způsobem závislá na implementaci rozhraní, a jednotlivé implementace tak lze jednoduše zaměňovat, aniž bychom v aplikačním kódu cokoliv měnili.

---

1. `https://github.com/sybila/parasim`
2. `http://repository-sybila.forge.cloudbees.com/snapshot/`
3. `http://www.cloudbees.com/`
4. `http://repository-sybila.forge.cloudbees.com/release`

4.1.1 Životní cyklus

Aby bylo možné jádro Parasimu používat, je nutné nastartovat jeho životní cyklus pomocí třídy `ManagerImpl`. Tato třída představuje vstupní bod pro použití veškeré dále popsané funkcionality včetně jednotného přístupu ke konfiguraci nebo injektování instancí služeb do atributů daných objektů. Životní cyklus řízený vytvořením, nastartováním a destrukcí tohoto manažera je znázorněn diagramem na obrázku 4.1.



Obrázek 4.1: Životní cyklus Parasimu. Zelené obdélníky představují události, které se propagují napříč rozšířeními. Šedými obdélníky je znázorněna základní funkcionalita. Čerchovaným ohraničením jsou vyznačeny části Java kódu.

Jakmile je manažer vytvořen, jsou načtena základní rozšíření, kterým je oznámena událost `ManagerProcessing`. Poté je možné manažera nastartovat, což vyústí ve vytvoření aplikačního kontextu. S vytvořením každého kontextu je spojena událost `Before`, která je propagovaná do všech rozšíření náležejících tomuto kontextu a v případě jiného než aplikačního kontextu i do rozšíření kontextu rodičovského. Po vytvoření aplikačního kontextu následuje událost `ManagerStarting`, na kterou mohou reagovat rozšíření definovaná mimo jádro Parasimu. Jakmile aplikace skončí, je nutné manažera zničit. Následuje vyvolání poslední události, na kterou mohou reagovat načtená rozšíření, `ManagerStopping` a zničení aplikačního kontextu spojené s událostí After, případně všech další dosud nezničených kontextů.

4.1.2 Kontexty

Manažer a objekty kontextů umožňují vytvářet nové kontexty skrze rozhraní ContextFactory. Aby mohl Parasim od sebe rozlišit jednotlivé kontexty, používá speciální anotace rozsahů. Samotná deklarace takové anotace je označená anotací `Scope`, jak je ukázáno ve zdrojovém kódu 4.1.

```
1 @Scope
2 @Documented
3 @Retention(RetentionPolicy.RUNTIME)
4 @Target(ElementType.TYPE)
5 public @interface Application {}
```

Zdrojový kód 4.1: Anotace rozsahu

Když je k dispozici anotace rozsahu rozlišující kontext, je možné vytvořit kontext nový, jak je ukázáno v ukázce kódu 4.2. Vývojář je zodpovědný za destrukci nepotřebné kontextu, aby rozšířením, která se nacházejí v tomto kontextu, umožnil uvolnit zdroje.

```
1 // vytvori kontext Scope1
2 // s rodicovskym kontextem Application
3 Context context1 = manager.context(Scope1.class);
4 // vytvori kontext Scope2
5 // s rodicovskym kontextem Scope1
6 Context context2 = context1.context(Scope2.class);
7 ...
8 context2.destroy();
9 context1.destroy();
```

Zdrojový kód 4.2: Vytvoření kontextu

4.1.3 Služby

Základní funkcionalitou jádra Parasimu je poskytovat instance služeb, které jsou vytvořeny pomocí rozšíření. Služby se definují pomocí rozhraní a může pro ně existovat více implementací. Aby byl Parasim schopen od sebe rozlišit jednotlivé implementace, používá kvalifikátory. Kvalifikátor je anotace, v jejíž deklaraci byla použita anotace `Qualifier`.

Pokud na daném místě aplikace není důležité, jaká implementace dané služby bude použita, případně není známo, jaké kvalifikátory jsou vůbec k dispozici, je možné pro výchozí implementaci použít kvalifikátor `Default`. Je na autorovi rozšíření, aby poskytl smysluplnou výchozí implementaci poskytované služby.

Příkladem vhodného použití kvalifikátorů je rozšíření poskytující výpočet robustnosti pro danou formuli nad danou trajektorii chování. Zde je možné zavést různé kvalifikátory pro různé temporální logiky. Výchozí implementace takové služby by měla být schopna spočítat robustnost pro všechny podporované typy temporálních logik, Je však možné, že se v takovéto implementaci bude nacházet méně efektivní algoritmus či analýza předložené formule, což zbrzdí výpočet.

```
1 @Qualifier
2 @Target({
3     ElementType.FIELD,
4     ElementType.METHOD,
5     ElementType.PARAMETER})
6 @Retention(RetentionPolicy.RUNTIME)
7 @Documented
8 public @interface Default {}
```

Zdrojový kód 4.3: Kvalifikátor

Manažer i kontexty implementují rozhraní `Resolver`, které umožňuje na základě rozhraní a kvalifikátoru získat instance dané služby. Pokud se v daném kontextu nenachází žádné rozšíření poskytující danou službu, je zavolán rodičovský kontext.

```
1 Manager manager = ...
2 Enrichment enrichment = manager.resolve(
3         Enrichment.class,
4         Default.class);
```

Zdrojový kód 4.4: Získání instance služby

Parasim používá ještě jeden, jednodušší typ služeb. Tyto služby jsou dostupné pouze pomocí manažera, nelze je od sebe odlišit pomocí kvalifikátoru a nejsou závislé na kontextu. Manažer poskytuje všechny dostupné implementace daného rozhraní v jedné kolekci. Tento typ služeb primárně slouží k ovlivňování chování rozšíření. Lze pomocí nich například naslouchat událostem z logování.

4.1.4 Rozšíření

Parasim je schopen načíst rozšíření, která jsou v době vytvoření manažera na `Java class path`. Všechna v aplikaci zatím použitá rozšíření byla přibalena do souboru JAR aplikace pomocí sestavovacího nástroje Maven, a proto jsou k dispozici bez dalšího nastavování. Rozšíření musí obsahovat následující části:

- alespoň jednu třídu, která implementuje rozhraní LoadableExtension;
- soubor `META-INF/services/org.sybila.parasim.core.spi.LoadableExtension` obsahující plné názvy všech tříd z daného rozšíření implementující LoadableExtension, jeden název na jednom řádku.

Rozhraní LoadableExtension obsahuje jedinou metodu, ve které je předán objekt pro registraci tříd rozšíření. Je možné registrovat služby nezávislé na kontextu a tzv. pozorovatele, což jsou třídy poskytující služby a naslouchající událostem. Podrobnější příklad toho, jak rozšíření může vypadat, se nachází v příloze B. Tato sekce obsahuje pouze vysvětlení nejdůležitějších konceptů.

Zmínění pozorovatelé jsou schopni pomocí metod označených anotací `Provide` poskytovat instance služeb. V deklaraci metody se mohou nacházet parametry, které je manažer jádra Parasimu schopen vyhodnotit a injektovat. Tyto parametry lze označovat kvalifikátory. V případě, že není žádný kvalifikátor k dispozici, je pro vyhodnocení hodnoty parametru použit kvalifikátor `Default`.

V případě, že je manažer požádán, aby poskytl instanci služby, metoda poskytující danou službu je zavolána, poskytnutý objekt propagován jako událost do všech dostupných rozšíření, a poté uložen pro případ, že by byla služba požadována znovu. Výjimku tvoří poskytující metody, pro jejichž návratový typ nelze vytvořit proxy[5]. Takové poskytující metody jsou zavolány již během načítání rozšíření. Pokud v čase volání poskytující metody

5. Typickým příkladem typů, pro které nelze vytvořit proxy, jsou finální třídy.

nelze najít hodnoty pro všechny její parametry, dojde k vyhození výjimky a pravděpodobně i k pádu celé aplikace.

```
/**
 * selze, pokud neni v aktualnim kontextu k dispozici
 * retezec pro dosazeni za parametr metody
 */
@Provide
public Functionality provideFunctionality(
        String required) {

    return new FunctionalityImpl(required);
}
```

Zdrojový kód 4.5: První metoda poskytující službu `Functionality`

Jestliže chce autor rozšíření záviset na jiných službách pouze volně, lze použít u parametrů poskytujících metod anotace `Inject` způsobem ukázaným ve zdrojovém kódu 4.6.

```
/**
 * pokud neni v aktualnim kontextu k dispozici
 * retezec pro dosazeni za parametr metody,
 * bude pouzita hodnota null
 */
@Provide
public Functionality provideFunctionality(
        @Inject(required=false) String optional) {

    return new FunctionalityImpl(optional);
}
```

Zdrojový kód 4.6: Druhá metoda poskytující službu `Functionality`

Druhým typem metod v pozorovatelských třídách jsou metody naslouchající událostem. Aby bylo možné tyto metody rozlišit, je první parametr těchto metod označen anotací `Observes` a typ tohoto parametru určuje typ naslouchané události. Pro další parametry těchto metod platí stejná pravidla jako pro parametry poskytujících metod. Pro odeslání vlastních událostí je k dispozici služba `EventDispatcher`.

4.1.5 Konfigurace

Jádro Parasimu nabízí jednoduchý způsob konfigurace jednotný pro všechna rozšíření a tím vybízí jejich autory, aby ve svých rozšířeních umožnili konfigurací změnit co nejvíce věcí. Zároveň je však kladen důraz na to, aby rozšíření bylo funkční samo o sobě bez toho, aby jej uživatel musel konfigurovat.

Pro autory rozšíření je k dispozici služba `ParasimDescriptor`, která je schopná na základě názvu rozšíření vrátit hodnoty konfiguračních proměnných ve formě řetězců a dále služba `ExtensionDescriptorMapper` schopná namapovat hodnoty konfiguračních proměnných do atributů u objektu v jazyce Java. Mapování umí automaticky převést řetězce do nejpoužívanějších datových typů.

Autor rozšíření tedy vytvoří konfigurační třídu s atributy, jejichž názvy se shodují s požadovanými názvy konfiguračních proměnných. Výchozí hodnoty těchto atributů jsou zároveň výchozími hodnotami pro konfiguraci rozšíření. Ukázka toho, jak může vypadat zpřístupnění konfigurace v rozšíření je k dispozici v příloze C.

Při užívání aplikace lze konfiguraci lze změnit dvěma způsoby:

- Pomocí systémové proměnné `parasim.config.file` v jazyce Java je možné nastavit cestu k XML souboru. Výchozí cesta k tomuto souboru je nastavena na „parasim.xml“. Tento soubor obsahuje pojmenované sekce a v těchto sekcích nastavení pro jednotlivé konfigurační proměnné. Názvy proměnných se shodují s názvy atributů konfiguračních objektů.

- Pro každou konfigurační proměnnou lze přepsat její hodnotu pomocí systémové proměnné v jazyce Java, jejíž název je `parasim.<název rozšíření>.<název konfigurační proměnné>`. Názvy konfiguračních proměnných i název rozšíření se zde uvádí v tečkové notaci, například pro atribut `timeUnit` a rozšíření `example` se bude systémová proměnná nazývat `parasim.example.time.unit`.

Pokud je nutné nějakou část konfigurace zpřístupnit změnám nejen při startu aplikace, ale i při jejím běhu skrze systémové proměnné, je potřeba mít na paměti okamžik, ve kterém se vytváří konfigurační objekt. Jakmile je totiž již konfigurační objekt vytvořen, nelze hodnoty konfigurace jemu náležící změnit. Po změně systémové proměnné je možné si znovunačtení konfiguračního objektu vynutit zničením příslušného kontextu a vytvořením nového. Na to však není vhodný aplikační kontext, protože jeho zničení prakticky znamená vypnutí aplikace.

4.1.6 Obohacování

Aby nebylo nutné si na všech místech předávat instanci manažera, případně instance všech potřebných služeb, nabízí Parasim rozšiřitelný mechanismus obohacování objektů. To je umožněno pomocí služby `Enrichment`. Tato služba spouští objety na kontextu nezávislých služeb `Enricher` na dané instanci, které ji dokáží různým způsobem vylepšit.

Jádro Parasimu obsahuje dvě implementace rozhraní `Enricher`, které umožní zpřístupnit poskytující metody a atributy podobně jako u rozšíření a které injektují služby do atributů. Za tímto účelem se používají již zmíněné anotace `Provide` a `Inject`. Tyto anotace lze použít ve dvou nastaveních, v jednom nastavení bude při obohacování vyhozena výjimka, pokud poskytovaný případně injektovaný objekt není k dispozici. Pokud je anotace použita s přiřazením `required=false`, výjimka se nevyhodí.

4.1.7 Vzdálený přístup

Další části práce ukážou, jak je možné Parasim použít k distribuovanému počítání. To je umožněno pomocí rozšíření pro vzdálený přístup, které zpřístupňuje získání některých služeb nacházejících se v aplikačním kontextu jiného vzdáleného stroje. Na stroji je nejprve nutné tuto funkcionalitu aktivovat skrze rozhraní `Loader.java`, jak je ukázáno ve zdrojovém kódu 4.7. Ukázaný kód nastartuje server pro práci s *Remote Method Invocation* [13] (RMO) a na tento server vystaví `Loader.java`.

```
1 manager.resolve(Loader.class, Default.class)
2         .load(Loader.class, Default.class);
```

Zdrojový kód 4.7: Spuštění serveru

Ostatní stroje jsou pak schopny vynutit si vystavení služeb z aplikačního kontextu tohoto stroje na server pro RMI. Aby bylo možné službu vystavit, je nutné, aby implementovala rozhraní `java.rmi.Remote`. Jakmile je server na vzdáleném stroji aktivován, je možné s ním komunikovat podobně jako ve zdrojovém kódu 4.8.

```
1 HostControl control = new HostControlImpl(
2         new URI("localhost"))
3
4 if (!control.isRunning(true)) {
5     throw new IllegalStateException();
6 }
7
```

```
8 RemoteServis servis = control
9         .lookup(RemoteServis.class, Default.class);
```

Zdrojový kód 4.8: Přístup ke vzdálenému serveru

Jakákoliv interakce s takto získanými službami vyvolá síťovou komunikaci a veškerá logika služby se vyhodnocuje na straně vzdáleného stroje.

4.2 Výpočetní model

Parasim obsahuje rozšíření pro snadnější provádění jistého druhu výpočtu. Základní jednotku výpočtu zde představuje instance. Během průběhu počítání lze tuto instanci rozdělit na více dalších a ty počítat nezávisle na sobě. Jednotlivé instance se mohou dále dělit a vrací mezivýsledky, které je možné slučovat pomocí asociativní a komutativní operace. Jak přesně se výpočetní instance dělí a jak jsou mezivýsledky slučovány, určuje ten, kdo implementuje algoritmus.

Důležitým aspektem však je, že se vývojář implementující algoritmus nemusí starat o způsob, jakým konkrétně budou výpočetní instance spočteny. To na druhou stranu vynucuje některá omezení, která musí vývojář při implementaci výpočtu dodržet. Parasim zatím nabízí jednotné rozhraní pro počítání ve sdílené a dístribuované paměti.

4.2.1 Reprezentace výsledku

Třída reprezentující výsledek pro daný výpočet musí implementovat rozhraní `Mergeable`. Toto rozhraní si vynucuje krom uchování dat také definici komutativní a asociativní operace pro slučování mezivýsledků. Současně je nezbytné, aby třída byla schopna serializace. Serializace je nezbytná pro výpočet v distribuované paměti, kde se data po skončení mezivýpočtu posílají mezi počítači po síti.

Z formálního hlediska již není kladen na datový typ výsledku žádný požadavek, nicméně je třeba mít na paměti, že se mezivýsledek v případě distribuovaného počítání serializuje a že je mezi stroji posílán po síti, tudíž velikost mezivýsledků může velkou měrou ovlivnit výpočetní čas. Podobně i metoda pro slučování mezivýsledků by neměla být výpočetně příliš náročná, protože spojování výsledků má zpravidla na starost jedno vlákno, respektive jeden stroj.

4.2.2 Reprezentace výpočtu

Jakýkoliv výpočet je v Parasimu definován jako třída implementující rozhraní `Computation`. Ve třídě vývojář implementuje algoritmus výpočtu za pomoci injektovaných služeb. V případě potřeby lze určit způsob, jakým se má instance zachovat ve chvíli, kdy již není zapotřebí.

V případě distribuovaného počítání se instance výpočtu může i několikrát posílat mezi stroji po sítí. Z tohoto důvodu je nutné, aby i třída definující výpočet byla serializovatelná. To v praxi znamená, že musí být serializovatelné i hodnoty všech atributů, které nejsou injektovány. Bohužel chybová hláška z virtuálního stroje Javy není v případě, že třída nesplňuje podmínky serializovatelnosti, příliš popisná. Zejména tento typ chyb se tedy velice špatně opravuje.

4.2.3 Životní cyklus výpočtu

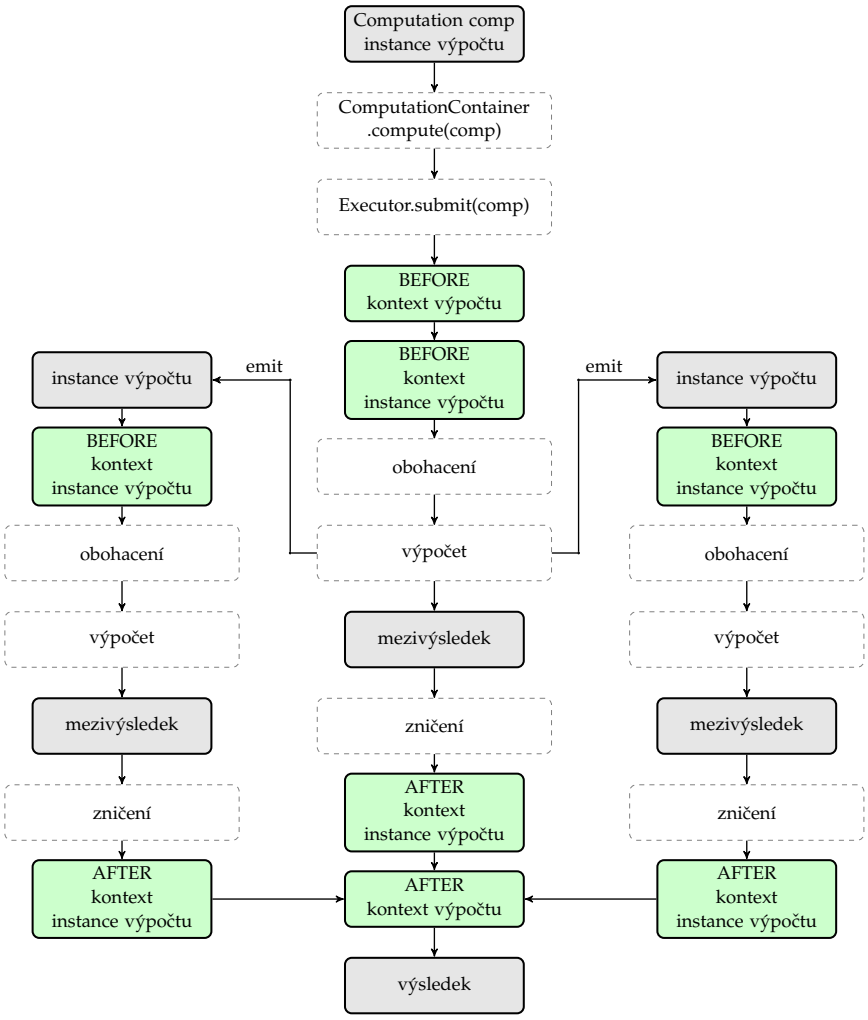
Vstupním bodem pro spuštění výpočtu je služba `ComputationContainer`. Tento kontejner rozhoduje, jaké použít prostředí pro výpočet na základě dostupné konfigurace a dostupných anotací. Pomocí anotace `RunWith` lze zvolit mezi prostředím se sdílenou nebo distribuovanou pamětí. Pokud tato anotace není k dispozici, je použito výchozí prostředí, které lze předefinovat globálně v konfiguraci Parasimu.

Jakmile kontejner provedl analýzu výpočetní instance, předává tuto instanci dále ke zvolenému výpočetnímu prostředí, ve kterém se instance spustí. Toto prostředí vytvoří kontext výpočtu, který je sdílen napříč celým výpočtem na jednom stroji. Pozor, v případě prostředí s distribuovanou pamětí, se tento kontext vytvoří na každém použitém stroji. Tento kontext je vhodný pro bezstavové služby a cache, do které si jednotlivé výpočetní instance mohou ukládat informace, jejichž získání je výpočetně náročné. Například u algoritmu pro analýzu dynamických systémů se v tomto kontextu nacházejí již nasimulované trajektorie chování nebo služba schopná spočítat pro trajektorii chování její lokální robustnost.

Pro každou výpočetní instanci je dále vytvořen další kontext, který již není sdílený s žádnou další instancí a jehož délka trvání může být mnohem kratší než u výpočetního kontextu. Tento kontext je vhodný pro služby, které by v případě sdílení napříč více vlákny musely být synchronizovány. V Parasimu se v tomto kontextu nachází služba pro simulaci trajektorie chování, která pro tento účel používá proces nástroje GNU Octave [11]. Není žádoucí, aby jeden proces nástroje GNU Octave byl sdílen více vlákny.

Aby se usnadnilo použití služeb z různých rozšíření, je před samotným

spuštěním výpočetní instance použito obohacování zmíněné v sekci 4.1.6. Až po tomto bodu se instance výpočtu spustí. Do této chvíle se v kontejneru nenachází žádná souběžnost a vše je prováděno pouze sekvenčně. V těle výpočtu však mohou být vytvářeny další instance výpočtu, a ty pak pomocí asynchronního volání služby `Emitter` emitovány do výpočetního prostředí, které je zodpovědné za jejich spuštění. Průběh počítání je tudíž podobný jako u modelu Fork/Join [23] uvedeného v jazyce Java verze 7.



Obrázek 4.2: Schéma průběhu výpočtu ve sdílené paměti. Zelené obdélníky představují události, šedými obdélníky objekty a čerchovaným ohraničením jsou vyznačeny části Java kódu.

Jakmile je výpočetní instance s výpočtem hotová, je vrácen výsledek a zničí se kontext spojený s touto instancí. Kontejner sbírá mezivýsledky a postupně je slučuje s dalšími. Až svůj výpočet ukončí poslední instance, zničí se výpočetní kontext na všech použitých strojích. Konečný výsledek a případně i aktuální podoba sloučených mezivýsledků je v aplikaci k dispozici v podobě objektu `Future`.

4.2.4 Výpočetní prostředí

Jak již bylo řečeno, Parasim nabízí dva typy prostředí, ve kterém lze výpočet spustit. Výchozím prostředím, které bude použito bez jakéhokoliv nastavování, je počítání ve sdílené paměti. Toto prostředí využívá standardního způsobu nepřímé práce s vlákny ve virtuálním stroji Javy pomocí třídy implementující `java.util.concurrent.ExecutorService`. Instance této třídy má k dispozici určité množství vláken, která jsou v kapitole 5 označována jako výpočetní. Těmto vláknům se posílají ke zpracování objekty implementující `java.util.concurrent.Callable`.

Takto odeslané objekty `java.util.concurrent.Callable` již nelze získat zpět, což způsobuje drobné komplikace. Aby bylo možné model pro sdílenou paměť převyužít i pro prostředí s distribuovanou pamětí, je zapotřebí umožnit balancovat výpočet napříč více stroji. V případě, že některý ze strojů není vytížený, kontejner zajistí, aby se na tento stroj přesunula instance výpočtu ze stroje více vytíženého. Z tohoto důvodu obsahuje Parasim mezivrstvu (třída `Mucker`). Instance výpočtu jsou nejprve uloženy do fronty a objektu `java.util.concurrent.ExecutorService` jsou dále posílány z této fronty tak, aby dostupná vlákna byla co nejvíce vytížená. Pokud dojde k přesouvání instancí výpočtu mezi stroji, jsou k tomu využity instance z fronty daného stroje.

Centrálním bodem výpočtu je třída implementující `MutableStatus`. Skrze tento bod se dorozumívají všechny objekty starající se o řádný průběh počítání. Využívají k tomu systém událostí a naslouchajících objektů implementujících rozhraní `ProgressListener`. Použité události jsou:

emitted Událost nastane ve chvíli, kdy je emitován nový výpočet pomocí služby `Emitter`.

computing Událost nastane ve chvíli, kdy je instance výpočtu z fronty výpočtů poslána vláknům k provedení a ještě před tím, než je pro instanci vytvořen příslušný kontext.

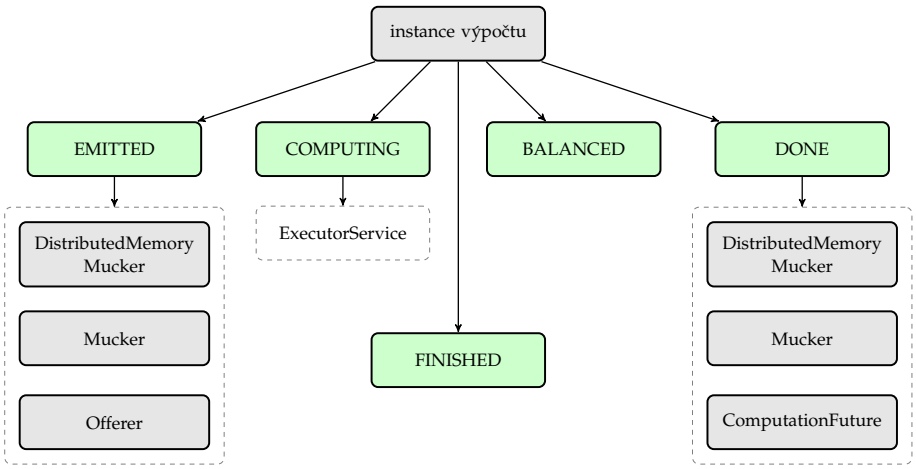
done Událost nastane ve chvíli, kdy je instance výpočtu se svým výpočtem

hotova. Stane se tak ještě před tím, než je vrácen výsledek, zničen kontext a zničena instance.

finished Událost nastane ve chvíli, kdy již není k dispozici žádná instance výpočtu.

balanced Událost nastane ve chvíli, kdy dojde k přesunutí instance výpočtu z jednoho stroje na jiný, a to pouze na stroji, kam byla instance přesunuta.

V případě prostředí s distribuovanou pamětí je průběh výpočtu složitější. Než začne samotný výpočet, je nutné na strojích, které chceme použít, spustit podporu pro vzdálený přístup zmíněný v sekci 4.1.7. Na každém z těchto strojů je výpočet zaregistrován pod náhodným identifikátorem a vytvoří se výpočetní kontext. V celém mechanismu se tedy nacházejí dva typy strojů – stroj, ze kterého byl výpočet iniciován (*master*), a stroje, mezi které je distribuovaná práce (*slave*).



Obrázek 4.3: Schéma událostí, které nastávají během výpočtu, a objektů, které na tyto události reagují.

Na rozdíl od prostředí se sdílenou pamětí se zde nacházejí synchronizační body dva. Výpočet na každém *slave* stroji řídí lokální status, který skrze události volá příslušné objekty nutné k obsluze lokálního výpočtu. Tento lokální status ještě před tím, než zavolá naslouchající objekty implementující rozhraní `ProgressListener`, kontaktuje s příslušnou událostí vzdálený status nacházející se na stroji *master*. Status na *master* stroji před-

stavuje synchronizační bod pro všechny stroje participující na daném výpočtu. Díky tomu, že je kontaktován při každé výpočetní události, může *master* stroj balancovat výpočetní instance napříč *slave* stroji. K tomuto účelu používá objekt `DistributedMemoryMucker`.

4.2.5 Možná nastavení výpočtu

Jednotlivé výpočty se od sebe mohou velkou mírou lišit, a je proto vhodné, aby celý mechanismus počítání v Parasimu byl co nejsnázeji nastavitelný. Jednou z věcí, která může výpočet ovlivnit, je pořadí, v jakém se jednotlivé instance budou počítat. Dalším důležitým faktorem může být výběr instance, která se přesune při balancování mezi méně a více vytíženým strojem. Tyto dva atributy se zdají být kritické pro implementaci uvažovaného algoritmu pro analýzu dynamického systému, protože obsahuje výpočetně velice náročnou numerickou simulaci, kvůli které je vhodné používat cache.

Ukládání trajektorií chování je paměťově poměrně náročné, a proto se po nějaké době musí trajektorie z paměti vymazat. Pokud se výpočetní instance provádějí v pořadí daném iterací zahušťování, je možné některé trajektorie vymazat z paměti dříve. V případě virtuálního stroje Javy se navíc menší využití paměti projeví i na výkonu *Garbage collector* [30].

Pokud přesouváme jednu výpočetní instanci z jednoho stroje na druhý, je třeba si uvědomit, že každý z těchto strojů má již vybudovanou svoji cache. Cílem balancování je přesunout nejlépe takovou výpočetní instanci, která bude pracovat s trajektoriemi nenacházejícími se v cache zdrojového stroje. Výpočet takové instance bude na cílovém stroji trvat nejvíce tak dlouho jako na stroji zdrojovém.

Parasim k tomuto účelu nabízí rozhraní `Selector`, které z dané kolekce vybere jeden její prvek. Pomocí anotace `RunWith` lze nastavit třídu implementující toto rozhraní pro účely balancování nebo výběru instance pro další počítání. U těchto objektů je opět použito mechanismu obohacování ze sekce 4.1.6, a tudíž mají k dispozici všechny služby z výpočetního kontextu na daném stroji. Vývojář však musí mít na paměti, že se tyto objekty budou volat velice často, a proto by výběr z kolekce neměl být příliš výpočetně náročný.

4.3 Dostupná rozšíření

Na závěr této kapitoly je vhodné uvést rozšíření, která jsou zatím pro účely aplikace implementující algoritmus pro analýzu dynamických systémů dostupná. Zde je uveden pouze výčet s krátkým popisem. Možné způsoby

konfigurace jsou k dispozici v příloze D.

computation-simulation Rozšíření poskytuje numerickou simulaci. Pro sekvenci bodů vrátí sekvenci trajektorií chování. Umožňuje rovněž již nasimulovanou trajektorii chování prodloužit o zvolený čas. Současná implementace používá volně dostupný nástroj GNU Octave [11].

computation-cycledetection Rozšíření pro detekci cyklu na trajektorii chování. V současné implementaci nástroje toto rozšížení není použito. Nicméně ostatní rozšíření jsou schopna s výsledky analýzy pracovat. Od detekování cyklu se upustilo z důvodu použití konečných intervalů u temporálních operátorů ve zkoumaných vlastnostech.

computation-density Rozšíření dokáže určit vzdálenosi mezi hlavní trajektorií a sousedními trajektoriemi chování a na základě této vzdálenosti zahustit prostor iniciálních podmínek. Obsahuje cache pro již použité trajektorie. Z důvodu nepřesnosti při počítání s reálnými čísly v jazyce Java se jako klíč v této paměti používají souřadnice obsahující zlomky, jejichž jmenovatel je mocninou čísla 2. Tento zlomek určuje, v jaké části prostoru iniciálních podmínek, který je po celou dobu výpočtu stejně velký, se trajektorie chování nachází.

computation-lifecycle Rozšíření poskytující již dříve zmíněný výpočetní model použitý v Parasimu.

V současné době Parasim obsahuje více rozšíření, než je zde zmíněno. Nicméně tato rozšíření se nevztahují k samotnému algoritmu, ale spíše k samotné aplikaci. Tato aplikace umí spravovat projekty s modely a nastavením pro analýzu. Také umožňuje výsledky analýzy zobrazovat a ukládat do souboru.

Kapitola 5

Evaluace

Předešlé kapitoly představily upravený algoritmus pro analýzu dynamických systémů a jeho implementaci v rámci nástroje Parasim. Následující kapitola prezentuje průběh analýzy nad vybranými modely v různě nastaveném výpočetním prostředí se sdílenou a distribuovanou pamětí. Nejprve si ukážeme, jakým způsobem byly vybrány analyzované modely. Následuje popis těchto modelů a prezentace naměřených dat. Naměřená data obsahují především výpočetní čas, který byl nutný k provedení experimentu v různých konfiguracích výpočetního prostředí. V závěru kapitoly se nachází interpretace měření.

5.1 Možné parametry

Průběh výpočtu popisovaného algoritmu pro analýzu dynamický systémů lze rozdělit do tří hlavních částí:

1. zahuštění iniciálního prostoru požadovanými body;
2. získání trajektorií chování;
3. ověření dané vlastnosti nad nasimulovanými trajektoriemi.

Ukazuje se, že bod 2 je výpočetně nejnáročnější, tvoří více než 90 % výpočetního času a přímo závisí na množství trajektorií chování nutných k analýze. Toto množství samozřejmě plyne z modelu, vlastnosti a nastavení perturbací. V algoritmu se nové trajektorie vytváří v bodu 1, který je parametrizován maximálním počtem provedených iterací zahuštění.

Dále je pravděpodobné, že z hlediska rozdělení práce v paralelním a distribuovaném prostředí bude rozdíl mezi analýzou menšího počtu dlouhých trajektorií, jejichž simulace je náročná, a mnoha krátkých trajektorií, které lze nasimulovat rychle. Délka trajektorie závisí na intervalech nacházejících se v ověřované formuli.

Z pohledu vstupních dat experimentu se tedy nabízí následující parametry:

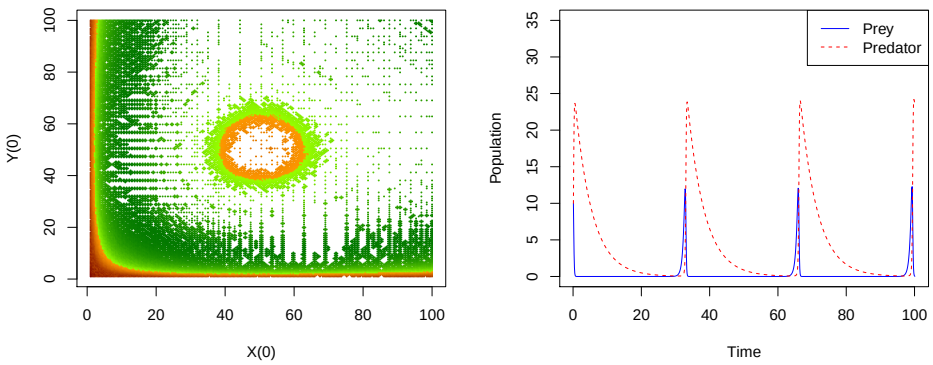
1. maximální počet iterací zahuštění;
2. počet perturbovaných parametrů a proměnných;
3. minimální délka trajektorie nutná k ověření dané vlastnosti;
4. vlastnosti modelu ovlivňující zahušťování.

5.2 Použité modely

Pomocí aplikace Parasim bylo provedeno několik experimentů pokrývajících výše uvedené vlastnosti, tyto experimenty zahrnují tři modely. U dvou z nich jsou dostupné tři konfigurace. Jednotlivé konfigurace se od sebe liší jen v několika detailech, ale i tyto detaily mají obrovský vliv na průběh analýzy, což bude ukázáno později v sekci 5.3.

5.2.1 Predátor a kořist

Prvním analyzovaným modelem je systém predátora a kořisti již dříve popsaný v sekci 2.1.2. Model obsahuje dvě proměnné, proměnnou x pro kořist a proměnnou y pro predátora. Analýza používá perturbaci iniciálních hodnot těchto proměnných v prostoru $[1, 100] \times [1, 100]$ a zkoumá oscilaci pomocí vlastnosti 5.1, která vznikla úpravou předpisu 2.13. Délka oscilace je pak dána konkrétní konfigurací, která byla k analýze použita. Hodnoty parametrů jsou nastaveny na $\alpha = \frac{1}{10}$, $\beta = \frac{2}{10000}$, $\gamma = \frac{1}{10}$ a $\delta = \frac{2}{10000}$.



(a) analýza na vybraných parametrech (b) vývoj systému v čase

Obrázek 5.1: Predátor a kořist

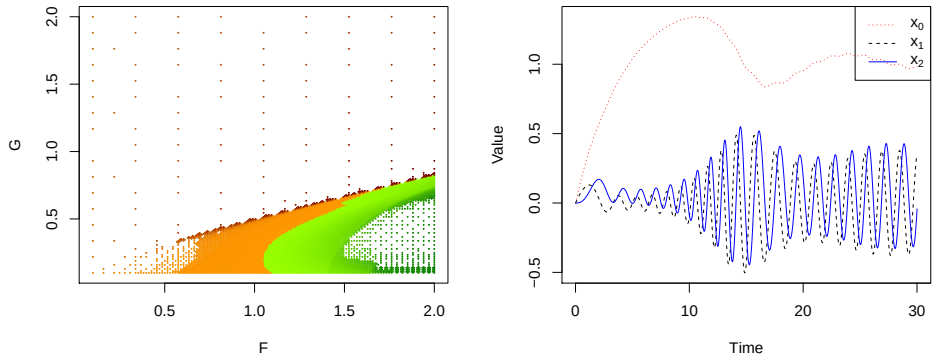
$$\mathcal{F}_{[0,100]}\mathcal{G}_{[0,\text{délka oscilace}]}\mathcal{F}_{[0,50]}(x \geq 40 \wedge \mathcal{F}_{[0,40]}x \leq 40) \tag{5.1}$$

Analýza byla provedena ve třech různých nastaveních, jejichž jména nesou pro další účely prefix `lotkav` a jejichž konkrétní popis je k dispozici v tabulce 5.1.

K zahuštění dochází zejména v extrémních hodnotách proměnných x a y, kde systém vůbec neosciluje, a ve středních hodnotách, ve kterých systém sice osciluje, ale okolo jiné hodnoty, než je požadováno v uvažované vlastnosti. Charakter zahuštění je vidět na obrázku 5.1(a).

5.2.2 Lorenzův atraktor

Dalším analyzovaným modelem je Lorenzův atraktor [25] skládající se ze tří stavových proměnných, který je odvozen ze zjednodušených rovnic proudění vzduchu v atmosféře. Ačkoliv tento model vykazuje v určitých hodnotách chaotické chování, je možné u něj nalézt na jedné z proměnných oscilaci.



(a) analýza na vybraných parametrech (b) vývoj systému v čase

Obrázek 5.2: Lorenzův systém

Konkrétní podoba modelu je dána rovnicemi 5.2. Uvažovány jsou perturbace parametrů F a G v prostoru iniciálních podmínek $[\frac{1}{10}, 2] \times [\frac{1}{10}, 2]$. Hodnoty fixních parametrů jsou $a = \frac{1}{4}$ a $b = 4$ a iniciální hodnoty $x_1 = x_2 = x_3 = 0$.

$$\begin{aligned}\frac{dx_0}{dt} &= a \cdot F - x_1^2 - x_2^2 - a \cdot x_0 \\ \frac{dx_1}{dt} &= x_0 \cdot x_1 + G - b \cdot x_0 \cdot x_2 - x_1 \\ \frac{dx_2}{dt} &= b \cdot x_0 \cdot x_1 + x_0 \cdot x_2 - x_2\end{aligned} \tag{5.2}$$

Analyzovanou vlastností je opět oscilace, tudíž lze s malými úpravami použít opět vlastnost 2.13. Pozorovanou proměnnou je x_1, která osciluje okolo 0.

$$\mathcal{F}_{[0,5]}\mathcal{G}_{[0,\text{délka oscilace}]}\mathcal{F}_{[0,5]}(x \geq \frac{1}{100} \wedge \mathcal{F}_{[0,5]}x \leq -\frac{1}{100}) \tag{5.3}$$

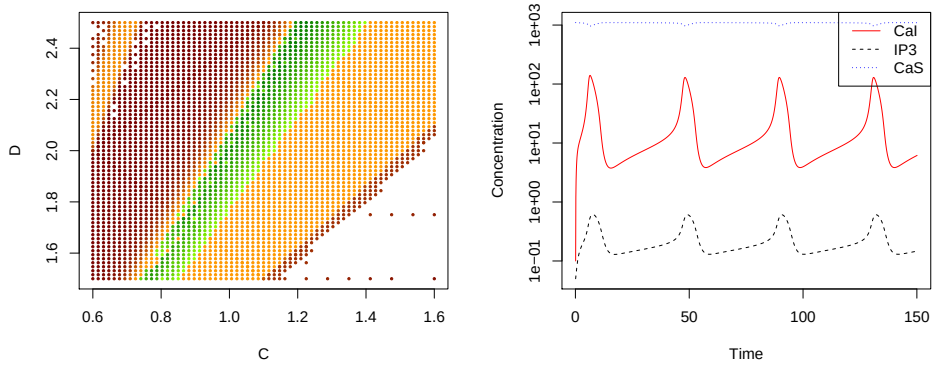
Lorenzův atraktor je velikostí modelu podobný predátoru a kořisti, nicméně v průběhu analýzy nedochází k tak rovnoměrnému zahušťování prostoru iniciálních podmínek. Popis konfigurací modelu použitých v rámci evaluace se nacházejí v tabulce 5.1, jejich názvy začínají prefixem `lorenz84`.

5.2.3 Oscilace vápníku

Posledním uvažovaným experimentem je model oscilace vápníku [28] pocházející z volně dostupné databáze biologických modelů [22]. Model lze z této databáze stáhnout za použití identifikátoru `BIOMD0000000224`. Hodnoty fixních parametrů a výchozí koncentrace látek jsou získané ze staženého SBML souboru, který byl bez jakýchkoliv změn importován do aplikace Parasim.

$$\begin{aligned}\frac{dCaI}{dt} &= (1-g) \cdot \left(\frac{A \cdot \left(\frac{IP_3}{2}\right)^4}{\left(\frac{IP_3}{2}+k_1\right)^4} + L \right) \cdot CaS - \frac{B \cdot \left(\frac{CaI}{100}\right)^2}{\left(\frac{CaI}{100}\right)^2 + k_2^2} \\ \frac{dIP3}{dt} &= C \cdot (1 - \frac{k_3}{CaI \cdot \frac{1}{100} + k_3} \cdot \frac{1}{1+R}) - D \cdot \frac{IP_3}{2} \\ \frac{dCaS}{dt} &= \frac{B \cdot \left(\frac{CaI}{100}\right)^2}{\left(\frac{CaI}{100}\right)^2 + k_2^2} - (1-g) \cdot \left(\frac{A \cdot \left(\frac{IP_3}{2}\right)^4}{\left(\frac{IP_3}{2}+k_1\right)^4} + L \right) \cdot CaS \\ \frac{dg}{dt} &= E \cdot \left(\frac{CaI}{100}\right)^4 \cdot (1-g) - F\end{aligned} \tag{5.4}$$

Model obsahuje velké množství parametrů a čtyři stavové proměnné CaI, IP_3, CaS a g, u tří z nich je požadována oscilace. Jak je vidět na obrázku 5.3(a), zahušťování během analýzy navíc probíhá velice rovnoměrně. Během analýzy byly perturbovány čtyři parametry k_1, k_2, C a D v prostoru iniciálních podmínek $[\frac{1}{10}, \frac{9}{10}] \times [\frac{1}{10}, \frac{2}{10}] \times [\frac{6}{10}, \frac{16}{10}] \times [\frac{15}{10}, \frac{25}{10}]$.



(a) analýza na vybraných parametrech (b) vývoj systému v čase

Obrázek 5.3: Oscilace vápníku

Požadovanou vlastností je opět oscilace. I když sledujeme oscilaci více proměnných, oscilují tyto proměnné synchronně, a proto lze opět použít upravenou vlastnost ze schématu 2.13:

$$\mathcal{F}_{[0,5]}\mathcal{G}_{[0,\text{délka oscilace}]}\mathcal{F}_{[0,50]}(\varphi_1 \wedge \mathcal{F}_{[20,50]}\varphi_2),$$
$$\text{kde } \varphi_1 \equiv CaI \geq 100 \wedge IP_3 \geq \tfrac{1}{2} \wedge g \geq \tfrac{9}{10} \qquad (5.5)$$
$$\varphi_2 \equiv CaI \leq 15 \wedge IP_3 \leq \tfrac{2}{10} \wedge g \leq \tfrac{4}{10}$$

	počet iterací zahušťování	délka oscilace
`lotkav-common`	8	300
`lotkav-iterations`	10	300
`lotkav-long-property`	8	6000
`lorenz84-common`	8	15
`lorenz84-iterations`	10	15
`lorenz84-long-property`	8	150
`meyer91-common`	6	150

Tabulka 5.1: Jednotlivé konfigurace experimentů, které byly pro účely evaluace použity. Vstupní soubory k experimentům jsou k dispozici v adresáři `benchmark/experiments` repozitáře aplikace Parasim.

5.3 Měření

Experimenty byly spuštěny v prostředí se sdílenou i distribuovanou pamětí s různým počtem dostupných procesorových jader, respektive strojů. Pro každé výpočetní vlákno Parasimu běží dva procesy nástroje Octave, který obstarává numerickou simulaci. Při měření ve sdílené paměti byla aplikace Parasim spouštěna s n výpočetními vlákny na $2n$ procesorových jádrech.

Na strojích pro prostředí s distribuovanou pamětí byla aplikace Parasim spouštěna se dvěma výpočetními vlákny a konfigurace těchto počítačů se lišily od počítače použitého pro prostředí s pamětí sdílenou. V obou prostředích měl virtuální stroj Javy k dispozici 4 GB paměti.

sdílená paměť: 64 jader, 2.27 GHz; paměť 450 GB; Red Hat Enterprise Linux Server release 6.4; Java 1.7.0_13-b20 (64 bit); Octave 3.4.3

distribuovaná paměť 4 jádra, 2.0 GHz; paměť 16 GB; NixOS 0.2pre-git; Java 1.7.0_13-b20 (64 bit); Octave 3.6.4

Kompletní výsledky měření pro všechny uvažované konfigurace modelů jsou k dispozici v příloze E. Ke každé konfiguraci jsou k dispozici čtyři grafy:

- čas nutný k provedení analýzy v prostředí se sdílenou a distribuovanou pamětí, pro názornost je v grafu červeně uvedeno ideální zrychlení, které je odvozeno z času výpočtu s jedním výpočetním vláknem, respektive na jednom počítači[1];
- počet simulovaných hlavních trajektorií chování během analýzy, pro názornost je v grafu uveden počet hlavních trajektorií, který by byl použit při naivním zahušťování odpovídajícím dané iteraci;
- množství neplatných přístupů do paměti v závislosti na počtu strojů v distribuovaném prostředí, tato paměť pomáhá předejít počítání duplicitních trajektorií.

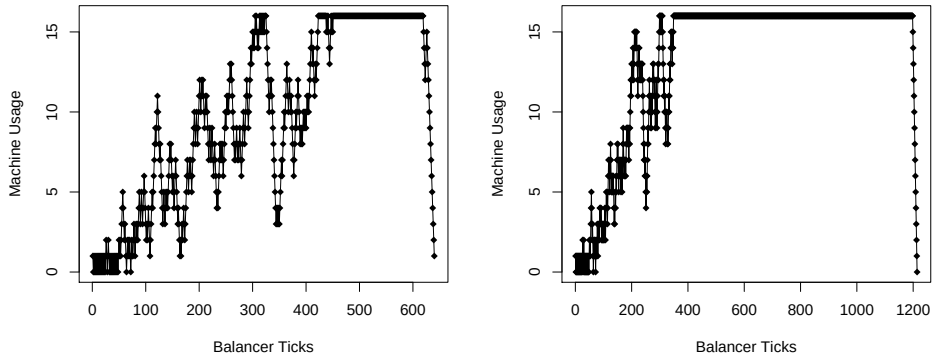
Experiment s modelem v konfiguraci `meyer91-common` byl spouštěn pouze v distribuovaném prostředí, a to minimálně na dvou strojích kvůli velkému množství trajektorií, se kterými je během analýzy nutno pracovat. Toto množství trajektorií je dáno počtem perturbovaných parametrů.

1. V případě, že výpočet s jedním výpočetním vláknem, respektive na jednom počítači trvá nepoměrně déle, může dojít k tomu, že v grafu vychází zrychlení větší než ideální.

5.4 Interpretace měření

5.4.1 Prostředí s distribuovanou pamětí

Při pohledu na grafy času nutného k provedení analýzy plyne, že implementace prostředí s distribuovanou pamětí pro jeden až šestnáct strojů škáluje poměrně dobře. Výjimku tvoří Lorenzův systém s maximálním množstvím iterací nastaveným na 8, což je vidět na grafu E.3(b). Při takto nastavené analýze se prostor iniciálních podmínek zahušťuje pouze v poměrně malém regionu a z toho důvodu se při malém množství iterací nestihne vytvořit dostatečné množství trajektorií tak, aby se plně vytížily všechny stroje.



(a) `lorenz84-common` (b) `lorenz84-iterations`

Obrázek 5.4: Množství strojů v čase, kterým náleží neprázdná fronta výpočtů. Každý tik odpovídá jednomu provedenému balancování.

Rozdělování práce mezi výpočetní jednotky vede k možnosti duplicitních výpočtů, a proto Parasim používá cache, ve které ukládá již dříve analyzované trajektorie. V prostředí s více stroji si každý počítač udržuje svou verzi paměti s již analyzovanými trajektoriemi. Jelikož je tato cache lokální, je možné, že při balancování výpočtu napříč stroji dojde k tomu, že se některé trajektorie analyzují vícekrát.

V současné implementaci se dává přednost rychlému výběru instance výpočtu pro přesun kvůli balancování, protože se na výběr čeká v kritické sekci na hlavním počítači (master), skrze který je celý výpočet řízen. Sofistikovanější, ale časově náročnější výběr zvýší čekací čas na komunikaci mezi výpočetními stroji a hlavním strojem. Ukazuje se, že podíl du-

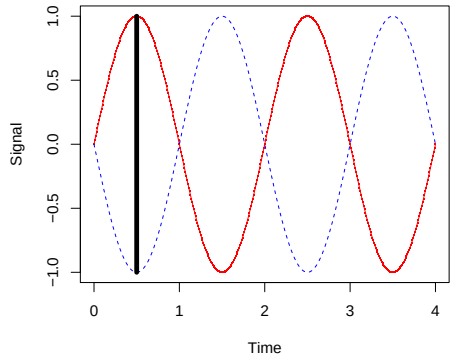
plicitní práce u náročnějších modelů tvoří pouze zlomek výpočtu, na druhou stranu u menších analýz se může množství duplicitně počítaných trajektorií vyšplhat i na 15 %. Je na zvážení, zda by se nevyplatilo používat časově náročnější balancovací metodu, která by se u větších a na komunikaci náročnějších analýz vypínala.

5.4.2 Prostředí se sdílenou pamětí

Ve sdílené paměti odpadá problém s duplicitními výpočty, protože cache s již napočítanými trajektoriemi je sdílená napříč všemi výpočetními vlákny. Při menším množství jader se projevují nedostatky Parasimu. Například na grafu E.3(b) výpočetního času u experimentu `lotkav-long-property` je vidět, že při srovnání použití jednoho a dvou výpočetních vláken dochází k většímu než dvojnásobnému zrychlení. To je pravděpodobně způsobeno množstvím vláken vytvořených aplikací Parasim a knihovnou pro komunikaci s aplikací Octave, kterých je více, než dokáže efektivně obsloužit daný počet jader. Tento „overhead“ se při větším počtu jader projevuje méně.

5.4.3 Další pozorování

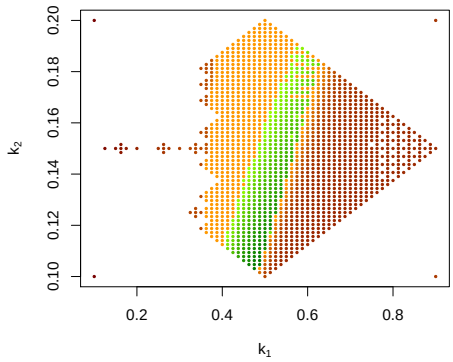
Hlavní motivací použít v této práci popsaný algoritmus je snížit počet trajektorií chování, které je nutné nasimulovat pro provedení analýzy daného dynamického systému. Je samozřejmě vhodné se ptát, zda při provedených experimentech došlo k nějakému výraznějšímu zlepšení oproti naivním metodám zahušťování.



Obrázek 5.5: Příklad fázového posunu, který vede k větší vzdálenosti, i když oba signály splňují oscilační vlastnost. Tuto vzdálenost znázorňuje černá úsečka.

Při pohledu na grafy množství použitých hlavních trajektorií v závislosti na iteraci zahušťování je zřejmé, že ve většině případů dochází k obrovské úspoře. Jedinou výjimkou je experiment `lotkav-long-property`, což je vidět na obrázku E.3(c). U tohoto experimentu sice výsledek analýzy dopadne podobně jako u experimentu `lotkav-common`, ale větší délka časového intervalu v analyzované vlastnosti je náchylnější na posun fází oscilace mezi jednotlivých trajektoriemi chování. Posun fáze vede k větší vzdálenosti trajektorií, a proto se prostor mezi nimi musí dále zahušťovat. Jak takový fázový posun může vypadat, ukazuje obrázek 5.5.

Obrázek E.7(b) u experimentu `meyer91-common` je lehce zkreslený nedokonalostí popisované heuristiky zahušťování, kdy blízkost dvou trajektorií vyústí v to, že se nezahustí poměrně velký prostor okolo nich. Kdyby v tomto prostoru k nějakému zahuštění došlo, je možné, že by docházelo k dalším iteracím zahuštění. Příkladem nedokonalého zahušťování je výsledek analýzy oscilace vápníku za použití perturbace parametrů k_1 a k_2 na obrázku 5.6.



Obrázek 5.6: Příklad analýzy, u které blízkost trajektorií pro iniciální body v extrémních hodnotách zapříčiní nedostatečné zahuštění.

Kapitola 6

Závěr

Cílem diplomové práce bylo implementovat v prostředí s distribuovanou pamětí algoritmus pro analýzu dynamických systémů zadaných pomocí soustavy diferenciálních rovnic vzhledem k vlastnostem definovaným v temporální logice signálů. Implementovaný algoritmus byl převzat z diplomové práce Svena Dražana [10] a rozšířen o lokální robustnost, jejíž použití umožňuje efektivnější pokrytí prostoru iniciálních podmínek. Implementace takto upraveného algoritmu vyústila ve vytvoření volně dostupného nástroje Parasim[1].

Na základě identifikovaných parametrů, jež mohou ovlivnit výpočet, byly vybrány modely, nad kterými se poté spustila analýza v různě nastaveném výpočetním prostředí. Z průběhu této analýzy a jejích výsledků vyplývá, že až na výjimky implementace ve sdílené i distribuované paměti škáluje. Pro dosažení tohoto výsledku nebylo zapotřebí žádných sofistikovanějších metod pro balancování výpočtu. Z naměřených dat také plyne, že použití robustnosti a prezentovaného způsobu pokrytí prostoru iniciálních podmínek má svůj význam, protože ve většině případů se oproti naivním metodám ušetřilo velké množství práce.

Princip analýzy je ve velké míře podobný tomu, který se používá v nástroji Breach [6]. Nicméně na rozdíl od něj je Parasim založen na volně dostupných technologiích, je snadno rozšiřitelný a podporuje distribuované počítání. Také způsob pokrytí prostoru iniciálních podmínek a použití robustnosti se liší.

Byl naimplementován vlastní výpočetní model, který abstrahuje od výpočetního prostředí a umožňuje snadno přecházet z prostředí se sdílenou pamětí do prostředí z pamětí distribuovanou a naopak. Díky tomuto aspektu a modulární architektuře je Parasim otevřen pro rozšiřování a implementaci dalších algoritmů.

Ve stávající implementaci je pro numerickou simulaci použit Octave, který se však ukázal v mnoha případech nevyhovující. Simulace trvá příliš

1. `https://github.com/sybila/parasim/wiki`

dlouho a často ani neposkytne požadované výsledky. Použití jiného nástroje pro řešení systému diferenciálních rovnic nebylo předmětem této práce, nicméně do budoucna se jeví jako vhodné místo nástroje Octave použít například nástroj COPASI [14], který je přímo určen pro analýzu biologických modelů.

Literatura

[1] GNU General Public License, version 3. http://www.gnu.org/licenses/gpl.html, cit. 2013-04-13.

[2] Jenkins CI. http://jenkins-ci.org/, cit. 2013-04-13.

[3] Alur, R.; Feder, T.; Henzinger, T. A.: The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, ročník 43, č. 1, 1996: s. 116–146.

[4] Aster, R.; Borchers, B.; Thurber, C.: *Parameter Estimation and Inverse Problems*. Academic Press, Academic Press, 2012.

[5] Chacon, S.: *Pro Git*. Berkely, CA, USA: Apress, první vydání, 2009.

[6] Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, Springer, 2010, s. 167–170.

[7] Donzé, A.; Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In *Formal Modeling and Analysis of Timed Systems*, Springer, 2010, s. 92–106.

[8] Donzé, A.; Fanchon, E.; Gattepaille, L. M.; aj.: Robustness Analysis and Behavior Discrimination in Enzymatic Reaction Networks. *PLoS ONE*, ročník 6, č. 9, 09 2011: str. e24246.

[9] Dräger, A.; Rodriguez, N.; Dumousseau, M.; aj.: JSBML: a flexible Java library for working with SBML. *Bioinformatics*, ročník 27, č. 15, 2011: s. 2167–2168.

[10] Dražan, S.: Výpočetní analýza nelineárních dynamických systémů. 2011.
URL https://is.muni.cz/th/139891/fi_m/

[11] Eaton, J. W.; Bateman, D.; Hauberg, S.: *GNU Octave Manual Version 3*. Network Theory Ltd., 2008.

[12] Eriksson, K.; Estep, D.; Johnson, C.: *Applied Mathematics: Body and Soul: Volume 1: Derivatives and Geometry in IR3*. Applied Mathematics, Body and Soul, Springer, 2004.

[13] Grosso, W.: *Java RMI*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., první vydání, 2001.

[14] Hoops, S.; Sahle, S.; Gauges, R.; aj.: COPASI—a complex pathway simulator. *Bioinformatics*, ročník 22, č. 24, 2006: s. 3067–3074.

[15] Horn, F.; Jackson, R.: General mass action kinetics. *Archive for Rational Mechanics and Analysis*, ročník 47, č. 2, 1972: s. 81–116.

[16] Hucka, M.; Finney, A.; Sauro, H. M.; aj.: The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, ročník 19, č. 4, 2003: s. 524–531.

[17] Iserles, A.: *A first course in the numerical analysis of differential equations*. New York, NY, USA: Cambridge University Press, 1996.

[18] Jendrock, E.; Evans, I.; Gollapudi, D.; aj.: *The Java EE 6 Tutorial: Basic Concepts*. Upper Saddle River, NJ, USA: Prentice Hall Press, Čtvrté vydání, 2010.

[19] JSR-299 Expert Group: JSR-299: Contexts and Dependency Injection for the Java EE platform. Technická zpráva, Oracle Corporation, Prosinec 2009.
URL `http://download.oracle.com/otndocs/jcp/web_beans-1.0-fr-eval-oth-JSpec/`

[20] Kermack, W. O.; McKendrick, A.: A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, ročník 115, č. 772, Srpen 1927: s. 700–721.

[21] Kitano, H.: Towards a theory of biological robustness. *Molecular Systems Biology*, ročník 3, č. 1, Září 2007.

[22] Le Novere, N.; Bornstein, B.; Broicher, A.; aj.: BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic acids research*, ročník 34, č. suppl 1, 2006: s. D689–D691.

[23] Lea, D.: A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, ACM, 2000, s. 36–43.

[24] Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. *arXiv preprint cs/0610046*, 2006.

[25] Lorenz, E.: Irregularity: a fundamental property of the atmosphere*. *Tellus A*, ročník 36, č. 2, 2010: s. 98–110.

[26] Lotka, A. J.: Elements of Physical Biology. *Nature*, ročník 116, 1925: s. 461–461.

[27] Maler, O.; Nickovic, D.: Monitoring temporal properties of continuous signals. *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, 2004: s. 71–76.

[28] Meyer, T.; Stryer, L.: Calcium spiking. *Annual review of biophysics and biophysical chemistry*, ročník 20, č. 1, 1991: s. 153–174.

[29] Pelánek, R.: *Modelování a simulace komplexních systémů. Jak lépe porozumět světu*. Brno: Masarykova univerzita, první vydání, 2012.

[30] Printezis, T.; Detlefs, D.: A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd international symposium on Memory management*, ISMM '00, New York, NY, USA: ACM, 2000, s. 143–154.

[31] Rizk, A.; Batt, G.; Fages, F.; aj.: A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*, ročník 25, č. 12, 2009: s. i169–i178.

[32] Strejček, J.: Expressiveness and Model Checking of Temporal Logics [online]. 2007 [cit. 2013-03-18].
URL `http://theses.cz/id/5qu2zj/`

[33] Westerhoff, H. V.; Hofmeyr, J.-H. S.: *What is systems biology? From genes to function and back*. Springer, 2005.

Příloha A

Způsob použití aplikace Parasim

Pro použití aplikace Parasim je nutné mít na svém počítačí nainstalovanou Javu verze 7 a Octave ve verzi 3.6.x. Poslední verzi Parasimu je možné si stáhnout ze stránek projektu [1] v podobě archivu JAR. Na těchto stránkách se rovněž nachází podrobný návod, jak aplikaci Parasim používat. Zde je uvedeno jen několik příkladů. Stažený archiv lze spustit standardní cestou:

```
java -jar parasim-2.0.0.Final-dist;
```

V případě, že je Parasim spuštěn bez jakýchkoliv dalších parametrů, nastartuje se grafické uživatelské rozhraní pro správu experimentů. Pro získání nápovědy k jednotlivým argumentům je nutné použít přepínač `-h`:

```
java -jar parasim-2.0.0.Final-dist -h;
```

K nastartování serveru pro distribuované počítání lze použít přepínač `-s`, je rovněž vhodné nastavit adresu stroje v síti:

```
java -Dparasim.remote.host=pheme01 -jar \
    parasim-2.0.0.Final-dist -s;
```

Na stránkách projektu je rovněž k dispozici Git repozitář, který krom zdrojových kódů obsahuje i projekty experimentů připravené ke spuštění. Tento repozitář je možné si stáhnout za použití následujícího příkazu:

```
git clone git://github.com/sybila/parasim.git;
```

V ukázkových projektech je k dispozici i model predátora a kořisti, jehož analýzu lze po načtení spustit z grafického uživatelského rozhraní nebo z příkazové řádky:

```
java -jar parasim-2.0.0.Final-dist\
    -e experiments/lotkav/oscil.experiment.properties;
```

1. `https://github.com/sybila/parasim/wiki`

Příloha B

Ukázka rozšíření pro Parasim

```
org.sybila.parasim.myextension.MyExtension
```

Zdrojový kód B.1: META-INF/services/org.sybila.parasim.core.spi.LoadableExtension

```
public class MyExtension implements LoadableExtension {
    public void register(ExtensionBuilder builder) {
        builder.extension(FunctionalityRegistrar.class);
    }
}
```

Zdrojový kód B.2: org/sybila/parasim/myextension/MyExtension.java

```
@ApplicationScope
public class FunctionalityRegistrar {

    private Context context;

    /**
     * Poksytni sluzbu Functionality pod kvalifikatorem
     * @Default
     */
    @Default
    @Provide
    public Functionality provideFunctionality(...) {
        return new FunctionalityImpl();
    }

    /**
     * Pokud je vytvoren kontext, oznam ostatnim
     * pozorovatelum udalost 'Hello World!'.
     */
```

```
20      public void hello(
21              @Observes Before event,
22              EventDispatcher eventDispatcher) {
23
24          if (before.getLoad().equals(OwnScope.class)) {
25              eventDispatcher.fire("Hello World!");
26          }
27      }
28
29      /**
30       * Kontext je vytvoren v momente,
31       * kdy je poskutnuta sluzna Functionality
32       */
33      public void startOwnContext(
34              @Observes Functionality event,
35              ContextFactory contextFactory) {
36
37          context = contextFactory
38                  .context(OwnScope.class);
39      }
40
41      /**
42       * Manazer se vypina, a proto je potreba
43       * vytvoreny kontext znicit.
44       */
45      public void stopOwnContext(
46              @Observes ManagerStopping event) {
47
48          if (context != null) {
49              context.destroy();
50          }
51      }
52
53      public interface Functiononality {
54      }
55
56      public static class FunctiononalityImpl
57              implements Functiononality {
58      }
59
```

```
60      @Scope
61      @Documented
62      @Retention(RetentionPolicy.RUNTIME)
63      @Target(ElementType.TYPE)
64      public @interface OwnScope {
65      }
66  }
```

Zdrojový kód B.3: org/sybila/parasim/myextension/FunctionalityRegistrar.java

Příloha C

Ukázka konfigurace pro Parasim

```
public class ExampleConfig {
    private long timeoutAmount = 10;
    private TimeUnit timeoutUnit = TimeUnit.SECONDS;

    public long timeoutAmount() {
        return timeoutAmount;
    }

    public TimeUnit timeoutUnit() {
        return timeoutUnit;
    }
}
```

Zdrojový kód C.1: Konfigurační třída

```
public ExampleConfig provideConfig(
        ParasimDescriptor descriptor,
        ExtensionDescriptorMapper mapper) {

    ExtensionDescriptor extDescriptor = descriptor
            .getExtensionDescriptor("example");
    ExampleConfig c = new ExampleConfig();
    if (extDescriptor != null) {
        mapper.map(extDescriptor, c);
    }
    return c;
}
```

Zdrojový kód C.2: Metoda poskytující konfiguraci

```
<parasim>
    <extension qualifier="example">
```

```
        <property name="timeoutAmount">2</property>
        <property name="timeoutUnit">days</property>
    </extension>
</parasim>
```

Zdrojový kód C.3: parasim.xml

Příloha D

Konfigurace dostupných rozšíření

Následuje seznam rozšíření a jejich dostupných konfiguračních proměnných. V závorkách u názvů rozšíření je uveden klíč, pod kterým je nutné k rozšířením přistupovat v konfiguraci. V závorkách u názvů konfiguračních proměnných se nachází výchozí hodnota.

D.1 Aplikace (`application`)

warmupComputationSize (30)
: Minimální počet hlavních trajektorií, se kterými pracuje jedna nezbytková výpočetní instance v zahřívací fázi výpočtu.

warmupBranchFactor (4)
: Maximální počet výpočetních instancí, které se emitují z jedné instance v zahřívací fázi výpočtu.

warmupIterationLimit (2)
: Číslo poslední iterace zahušťování, která ještě patří do zahřívací fáze výpočtu.

computationSize (60)
: Minimální počet hlavních trajektorií, se kterými pracuje jedna nezbytková výpočetní instance.

branchFactor (2)
: Maximální počet výpočetních instancí, které se emitují z jedné instance.

showingRobustnessComputation (`true`)
: Zapíná a vypíná možnost zobrazit okno s výpočtem robustnosti pro jeden iniciální bod po kliknutí na vizualizaci výsledků analýzy.

D.2 Logování (`logging`)

configFile
Konfigurační pro logovací knihovnu `log4j`. Pokud není specifikován použije se se konfigurační soubor `log4j.properties` distribuovaný společně s rozšířením.

level (`info`)
Úroveň logovacích zpráv, které se zobrazí v konzoli. Možné hodnoty jsou – `all`, `debug`, `error`, `fatal`, `info`, `off`, `trace` a `warn`.

D.3 Vzdálená správa (`remote`)

host (`InetAddress.getLocalHost().getHostAddress()`)
Adresa stroje v síti.

D.4 Výpočetní model (`computation-lifecycle`)

numberOfThreads (počet dostupných procesorových jader)
Počet použitých výpočetních vláken ve sdílené paměti.

nodeThreshold ($\frac{3}{2}\times$ počet dostupných procesorových jader)
Minimální počet současně počítaných výpočtů ve službě `ExecutorService`, který se výpočetní kontejner snaží udržet.

balancerMultiplier ($\frac{3}{2}$)
K balancování dochází pouze pokud $b >$ `balancerMultiplier` $\cdot\, i$, kde b je počet výpočetních instancí na nejvíce zatíženém stroji a i počet instancí na nejméně vytíženém.

balancerBusyBound (1)
K balancování dochází pouze pokud je počet výpočetních instancí na nejvíce zatíženém stroji vyšší než toto číslo.

balancerIdleBound (1)
K balancování dochází pouze pokud je počet výpočetních instancí na nejméně zatíženém stroji nižší než toto číslo.

nodes
Stroje použité pro distribuované počítání. Na těchto strojích musí být Parasim spuštěn s přepínačem `-s`.

defaultExecutor (org.sybila.parasim.computation.lifecycle.api.SharedMemoryExecutor)
Výchozí výpočetní prostředí.

D.5 Numerická simulace (`simulation`)

lsodeIntegrationMethod (`nonstiff`)
Integrační metoda pro LSODE, k dispozici jsou – `adams`, `nonstiff`, `bdf` a `stiff`.

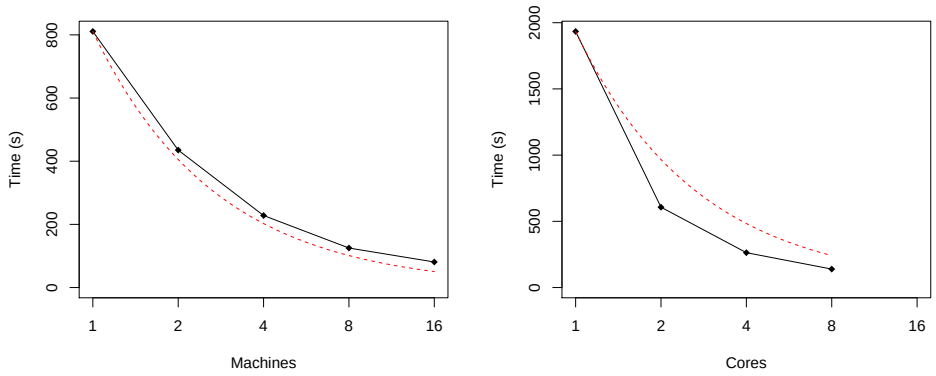
odepkgFunction
Pokud je nastaveno, je použita integrační metoda z balíku `odepkg`, dostupné hodnoty jsou – `ode5r`, `ode78`, `odebda`, `odebdi`, `odekdi`, `oders` a `odesx`.

D.6 Detekce cyklu (`cycledetection`)

relativeTolerance ($\frac{1}{100}$)
Relativní tolerance pro ověření ekvivalence dvou bodů pro účely detekce cyklu.

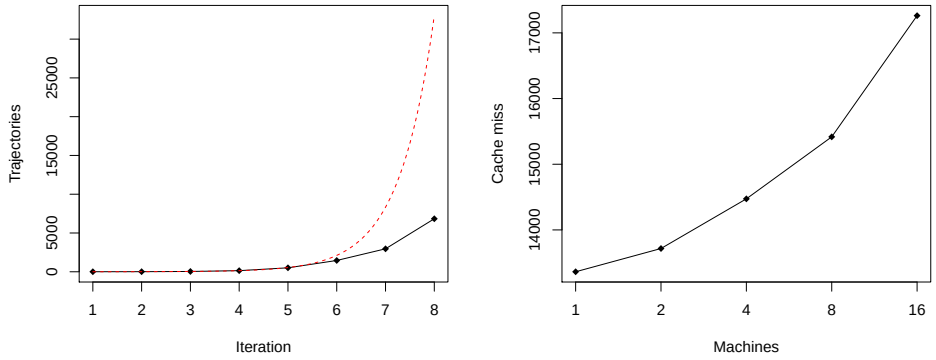
Příloha E

Výsledky měření



(a) čas nutný k provedení analýzy v distribuovaném prostředí

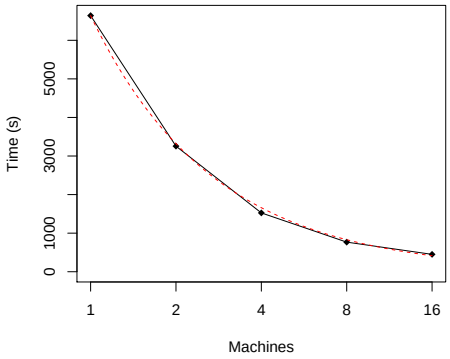
(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí



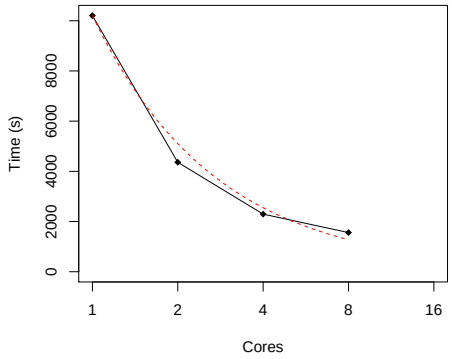
(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací

(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

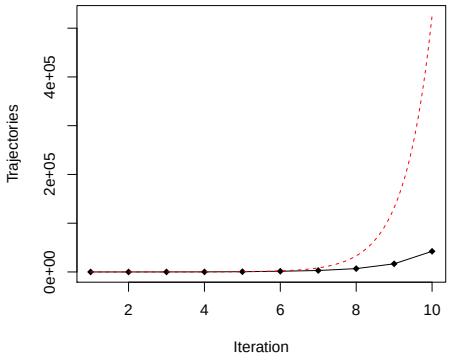
Obrázek E.1: `lotkav-common`



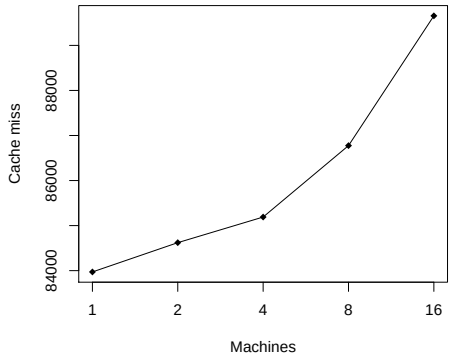
(a) čas nutný k provedení analýzy v distribuovaném prostředí



(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí

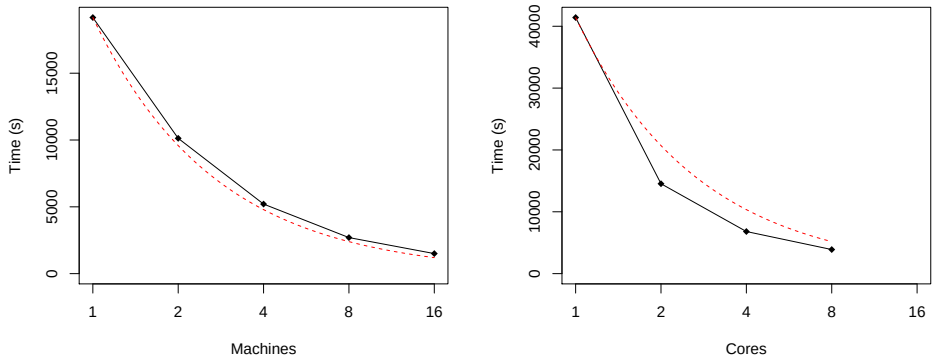


(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací



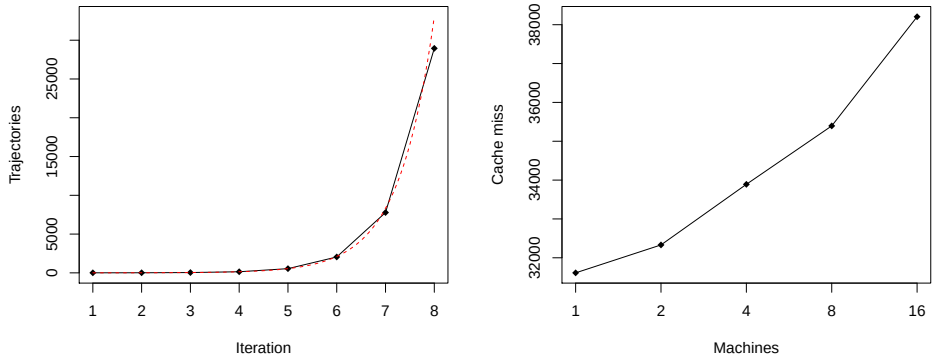
(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

Obrázek E.2: `lotkav-iterations`



(a) čas nutný k provedení analýzy v distribuovaném prostředí

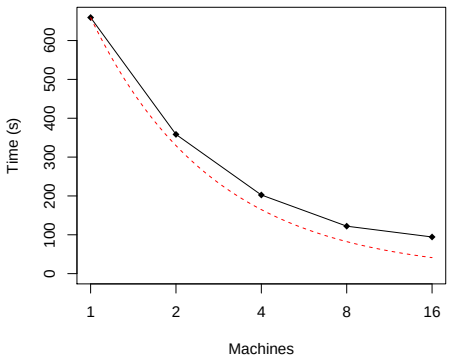
(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí



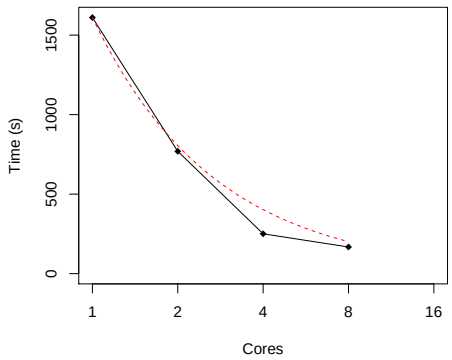
(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací

(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

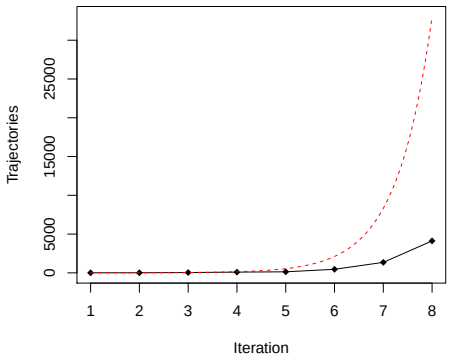
Obrázek E.3: `lotkav-long-property`



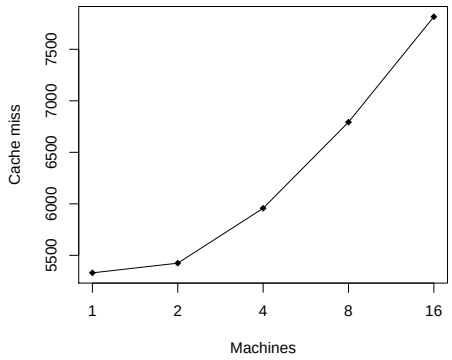
(a) čas nutný k provedení analýzy v distribuovaném prostředí



(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí

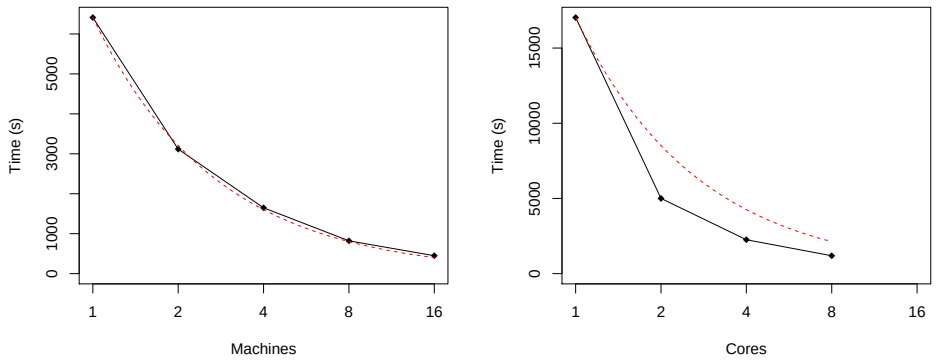


(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací



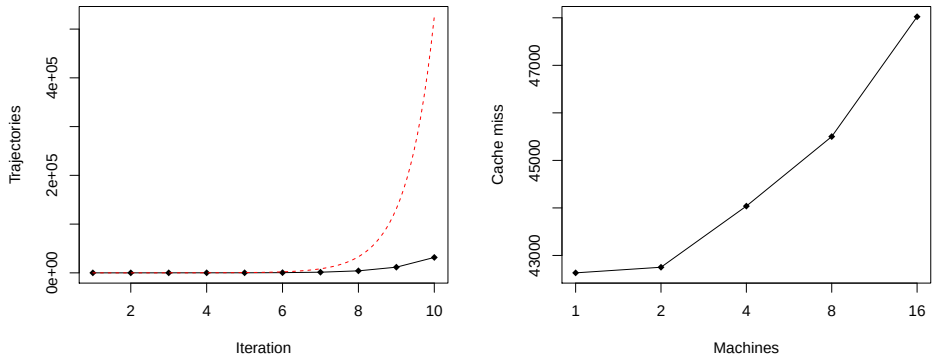
(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

Obrázek E.4: `lorenz84-common`



(a) čas nutný k provedení analýzy v distribuovaném prostředí

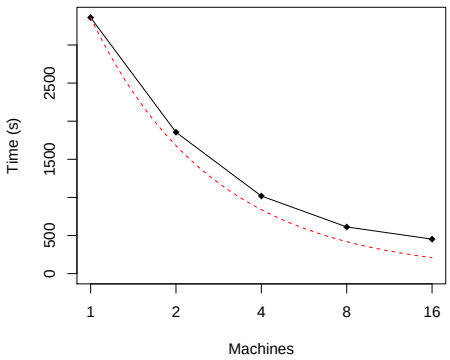
(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí



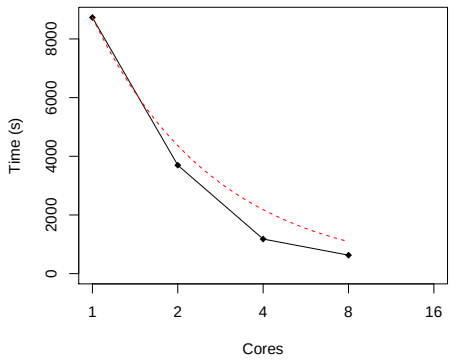
(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací

(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

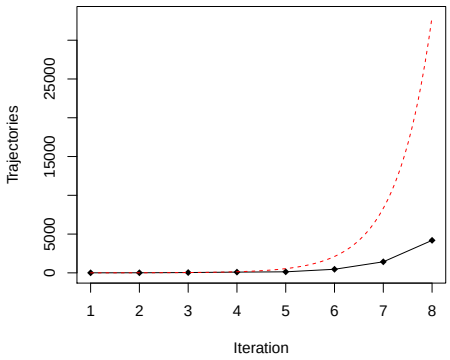
Obrázek E.5: `lorenz84-iterations`



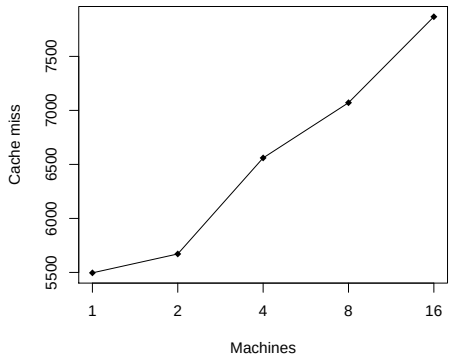
(a) čas nutný k provedení analýzy v distribuovaném prostředí



(b) čas nutný k provedení analýzy v prostředí se sdílenou pamětí

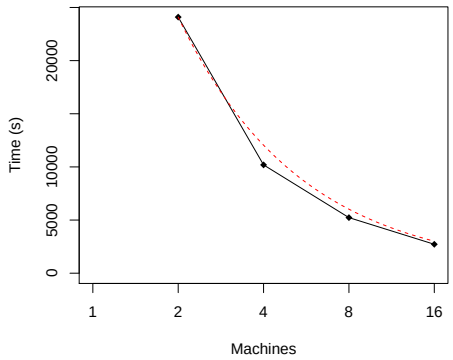


(c) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací

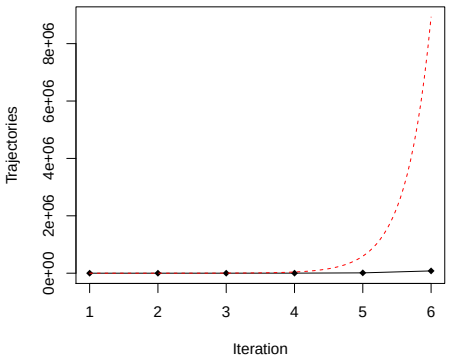


(d) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

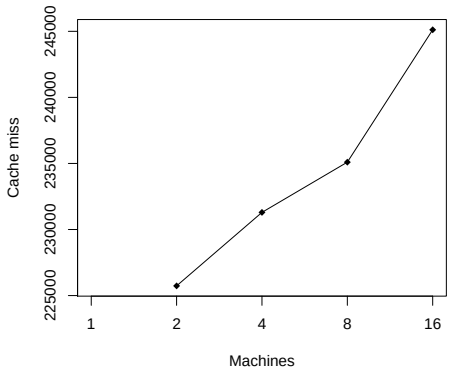
Obrázek E.6: `lorenz84-long-property`



(a) čas nutný k provedení analýzy v distribuovaném prostředí



(b) počet simulovaných hlavních trajektorií chování v porovnání s naivním zahušťováním odpovídajícímu danému počtu iterací



(c) počet neplatných přístupů do paměti, kde se ukládají již analyzivané trajektorie chování

Obrázek E.7: `meyer91-common`