ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# ZADÁNÍ DIPLOMOVÉ PRÁCE

**Název:** SAT s diferenciálními rovnicemi
**Student:** Bc. Tomáš Kolárik
**Vedoucí:** doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
**Studijní program:** Informatika
**Studijní obor:** Návrh a programování vestavných systémů
**Katedra:** Katedra číslicového návrhu
**Platnost zadání:** Do konce letního semestru 2018/19

## Pokyny pro vypracování

Realistické modely vestavěných systémů modelují nejen samotnou hardwarovou část, ale i fyzikální okolí. Pro tento účel mohou např. sloužit řešiče, které rozšíří řešič pro splnitelnost formulí ve výrokové logice o diferenciální rovnice. Cílem práce je vytvoření takovéhoto řešiče, který přitom používá pro řešení diferenciálních rovnic klasické numerické metody [3, 4] (na rozdíl od existujících řešičů [1, 2], které používají k tomu metody na základě intervalové aritmetiky).

Metodologie:

1) Obeznamte se s existujícími řešiči pro kombinaci problému SAT s obyčejnými diferenciálními rovnicemi [1,2].
2) Spolu se školitelem navrhněte rozhraní řešiče včetně vstupního jazyka na základě SMTLIB standardu [5].
3) Analyzujte vhodnost a možnosti použití existujícího softwaru pro implementaci návrhu [3,4,6].
4) Implementujte návrh.
5) Na základě výpočetních experimentů porovnejte Vaši implementaci s alespoň jedním existujícím řešičem.

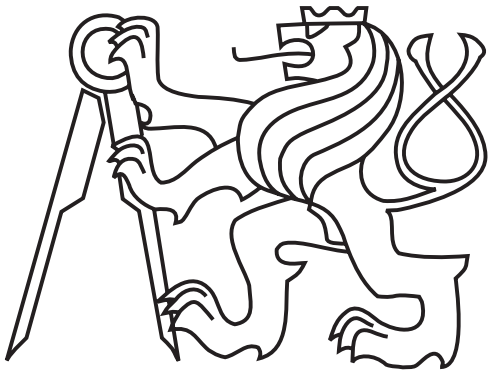
## Seznam odborné literatury

[1] http://dreal.github.io/
[2] http://www.avacs.org/tools/isatode/
[3] http://computation.llnl.gov/projects/sundials
[4] http://headmyshoulder.github.io/odeint-v2/
[5] http://smtlib.cs.uiowa.edu/
[6] http://verify.inf.usi.ch/opensmt

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 25. října 2017



Diplomová práce

# SMT řešič s diferenciálními rovnicemi

***Bc. Tomáš Kolárik***

Katedra číslicového návrhu
Vedoucí práce: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

3. května 2018

# Poděkování

Děkuji zejména celé své rodině za veškerou podporu a motivaci během celého studia. Dále panu vedoucímu doc. Dipl.-Ing. Dr. techn. Stefanu Ratschanovi za trpělivou a ochotnou asistenci s touto prací.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 3. května 2018 .....

### Odkaz na tuto práci

Kolárik, Tomáš. *SMT řešič s diferenciálními rovnicemi*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018. Dostupný také z WWW: ⟨`https://github.com/Tomaqa/sos`⟩.

# Abstrakt

**Klíčová slova** SAT, SMT, numerické metody pro ODE, analýza hybridních systémů

# Abstract

**Keywords** SAT, SMT, numerical methods for ODEs, hybrid systems analysis

# Obsah

# Seznam obrázků

# Úvod

Většina lidí si z informačních technologií jako první vybaví stolní počítače, notebooky či mobily. Dnešní mobilní telefony jsou specifické tím, že jejich funkce značně souvisí s vnějšími podněty z reálného světa: komunikují přes různá bezdrátová připojení, pořizují zvukové i obrazové záznamy, ovládají se dotykovou obrazovkou, atd. To všechno znamená interakci s fyzikálním světem pomocí různých snímačů a akčních členů. Existuje však odvětví zařízení, které se od mobilů liší v podstatě jen v jediné, nicméně naprosto zásadní specifikaci. Jsou to bezpečnostně kritické systémy reálného času, jejichž hlavní rozdíl tkví v tom, že na základě vnějších podnětů *musí* být vykonány dané akce bezpodmínečně do nějakého časového okamžiku. Jakmile se to nestane (ať už opožděně nebo vůbec), dojde k nějaké katastrofě, jejíž následky budou velmi drahé (finanční prostředky, ale i lidské životy). Typickým příkladem takových systémů jsou např. dopravní prostředky (letadla, tramvaje) a průmyslová zařízení (robotické stroje). S rostoucími požadavky na tyto systémy však závratně roste počet různých stavů, ve kterých se mohou nacházet, a které také chceme rozlišovat na přípustné a nepřípustné. Jak lze ale uchopit takto komplexní problémy, které ještě musí splňovat časové požadavky?

Takové systémy už lidstvo používá mnoho desítek let, přesto je téma této práce aktuální. Je to způsobeno tím, že do určité doby stačilo tyto systémy jen simulovat, tj. vygenerovat reprezentativní sadu vstupních dat a kontrolovat výstupy, zda odpovídají zadání. To naráží na potíž, že u bezpečnostně kritických aplikací je záhodno testovat téměř všechny možné přípustné vstupy. Vzhledem k tomu, že množství kombinací různých vstupních dat roste exponenciálně s počtem sledovaných specifikací, došli vývojáři do bodu, kdy byl tento postup testování již příliš dlouhý, drahý a nespolehlivý. Tehdy se začalo přecházet na jiný způsob ověření spolehlivosti systémů — pomocí jejich *modelu* — matematického popisu, který zanedbává nedůležitá hlediska a soustředí se na funkce systému. Pro takový zjednodušený model systému potom lze formálně (zcela přesně) dokázat, zda mohou či nemohou nastat zakázané stavy. Ve své práci se

zabývám možnými nástroji sloužícími k ověření takovýchto modelů systémů.

## Cíl práce

Systémy jsou typicky modelovány *přechodovým systémem*, který popisuje možné stavy systému a přechody mezi nimi (stavový prostor). Jednoduché systémy lze modelovat diskrétním časem a libovolnými stavy. K jejich analýze lze použít SAT řešič rozšířený o rovnice a nerovnice (aritmetická omezení) — *Satisfiability Modulo Theories (SMT)* [1]. Problém nastává, pokud chceme modelovat čas spojitě, což je pro fyzikální systémy zcela přirozené. Pak je nutné sáhnout po diferenciálních rovnicích, neboť je přechod ve spojitých veličinách chápán jako změna — derivace. Zadání je omezeno na obyčejné diferenciální rovnice (obsahují funkce pouze jedné nezávislé proměnné) — *Ordinary differential equation (ODE)*. SMT řešiče a řešiče diferenciálních rovnic již existují, ale často jen separátně. Cílem mé práce bylo vhodně zvolit oba jednotlivé existující řešiče a propojit je. To souviselo také s nutností definovat společný vstupní jazyk a snažit se držet standardů.

Mou osobní motivací bylo zejména mé zalíbení v SAT řešičích, se kterými jsem v minulosti reálně pracoval, a také v modelování systémů přechodovými systémy. Není náhodou, že se tyto premisy shodují přesně se zadáním.

*/* Struktura a návaznost ... */*

## Požadavky zadání

Zadání požaduje propojení existujících SMT a ODE řešičů a s tím související úkoly:

- najít ODE řešič používající klasické numerické metody,
- srovnat výsledný program s existujícími SMT+ODE řešiči a dosáhnout pokud možno větší výkonnosti,
- navrhnout společné rozhraní a vstupní jazyk pro SMT+ODE na základě *SMT-LIB* standardu [2],
- implementovat návrh.

## Rešerše

Nejprve bylo nutné se seznámit s již existujícími nástroji, které:

- řeší jak SMT, tak ODE, ale používají pomalou intervalovou aritmetiku;
- řeší jen SMT,
- řeší jen ODE pomocí klasických numerických metod.

Kapitola 1

# Teoretická část

V této kapitole se zabývám teoretickými podklady problémů spjatými s touto prací. Pojmy neuvádím zcela přesně, spíše dávám přednost srozumitelnosti. Tato práce klade větší důraz na praktickou část.

## 1.1 Formulace problémů

Tato sekce popisuje konkrétní problémy a jejich varianty, kterými se v této práci zabývám. Možnosti jejich řešení jsou uvedeny až v následující sekci rešerše řešičů.

### 1.1.1 Problém splnitelnosti: SAT

labelss:theory:formulation:sat Problém splnitelnosti Booleovské formule je základním problémem ze třídy NP-úplných problémů[1] v oboru teorie složitosti. Jedná se o široce studovaný problém implementovaný v řadě velmi efektivních specializovaných řešičů, které jsou využívány v různých aplikacích, díky možnosti převoditelnosti. Přestože se jedná o těžký problém, i rozsáhlé praktické instance (např. se stovkami tisíc proměnných) je možné řešit rychle, neboť výskyt instancí s těmi nejobtížnějšími kombinacemi je v praxi nepravděpodobný.

SAT je definován jako úloha nalezení ohodnocení Booleovských proměnných $\boldsymbol{y}$ ve formuli $F$ v Booleově algebře, tj.

$$\exists \boldsymbol{y} : \; F(\boldsymbol{y}) = 1 \tag{1.1}$$

Výstupem je buď nalezené ohodnocení $\boldsymbol{y}$, nebo (typicky) `unsat` v případě, že formule není splnitelná.

Základní verze obsahuje existenční kvantifikátor $\exists$. Pokud je použit obecný kvantifikátor $\forall$, jedná se o problém tautologie, který je co-NP-úplný (doplněk

[1] U tohoto NP problému bylo jako u prvního prokázáno, že na něj lze v polynomiálním čase převést jakoukoli úlohu ze třídy NP [3].

k NP-úplnému). Pokud je povolena kombinace obou kvantifikátorů, hovoříme o problému *kvantifikované Booleovské proměnné* (angl. *QBF*) a dostáváme se o třídu složitosti výše.

Existují i optimalizační varianty tohoto problému, často ve formě vážené splnitelnosti, kde proměnné nebo klauzule mají přiřazeny váhy a úkolem je nalézt řešení s maximální vahou proměnných či klauzulí, které jsou či nejsou splněny, apod.

**Bounded Model Checking (BMC)** (omezené ověření modelu) je jedna z hlavních aplikací SAT problému, které slouží k automatizované formální verifikaci systému reprezentovaného přechodovým systémem [4]. Hlavním cílem je dokázání správnosti modelu, tj. zda není možné dospět do zakázaných stavů. K účelům specifikace takových přechodových systémů se používá temporální logika, většinou LTL nebo CTL.

Základní myšlenka techniky BMC spočívá v symbolickém nalezení protipříkladu, který má omezenou délku, vůči zkoumané formuli ze specifikací. Využívá SAT řešiče — nalezení ohodnocení proměnných znamená nalezení protipříkladu, neboli porušení specifikací. Opačný případ je obtížnější, neboť teprve projití cest pokrývajících všechny dosažitelné stavy dokazuje, že zakázané stavy nemohou nastat, což může vyžadovat prohledání obrovského stavového prostoru. Algoritmus se tedy opakuje se zvyšující délkou zkoumaných cest dokud není nalezen protipříklad, nebo dokud není dosaženo maximální meze.

Alternativní použití BMC spočívá ve zkoumání negované formule — potom nalezení protipříkladu omezené délky naopak znamená, že formule je vždy splněna.

### 1.1.2 Satisfiability Modulo Theories (SMT)

Jedná se o rozšíření problému SAT o další domény než je Booleova algebra, tzn. dokáže operovat i s proměnnými, jejichž definiční obor je rozsáhlejší než jen $\{0, 1\}$, nemusí být dokonce ani *konečný*[2] (např. v našem případě kombinování s ODE jsou typickou doménou reálná čísla). Stále je hlavním zájmem ověřování splnitelnosti vstupních formulí.

Klíčovým pojmem v SMT je *teorie*, která je zodpovědná za definování funkcí a pravidel nad jejími prvky. Speciálním případem teorie je též teorie Booleovy algebry, která bývá v SMT řešičích implementována jako teorie základní.

Hlavní motivací SMT oproti SAT je využití aritmetických funkcí a pravidel, které zlepšují vyjadřovací schopnosti daného jazyka. Řešení SMT může být také efektivnější, než kdyby byla formule celá zakódována do SAT. Složitost

[2]Zatím se bavíme o matematickém modelu, v konečném důsledku jsou však domény v implementacích vždy konečné, protože počítače mají omezenou velikost. Univerzum však není v SMT podstatné.

rozhodování SMT se ale dramaticky liší s ohledem na zvolenou teorii: může být i polynomiální, ale i horší než exponenciální [5].

#### 1.1.2.1 Teorie

Teorie prvního řádu (angl. *First-order theory*) je vyjádřena v predikátové logice prvního řádu[3]. Teorie definuje konečně mnoho pravidel nad *abstraktními* prvky, tj. aniž by definovala jejich univerzum; postup je opačný — přípustné prvky jsou určeny výhradně jako důsledek pravidel teorie.

**(Predikátová) logika prvního řádu** (angl. *First-order logic, FOL*). Hlavní rozdíl oproti Booleově algebře (resp. výrokové logice) je ten, že termy formulí mohou být hodnoty libovolné domény [5].

Formule FOL se skládají z proměnných a konstant, predikátů, funkcí, logických operací a kvantifikátorů. Termy jsou konstanty, proměnné a funkce. Predikáty jsou funkce, které nabývají jen logických hodnot. Literál je logická proměnná či konstanta, predikát, nebo jejich negace.

Interpretace formule přiřazuje elementy, funkce a predikáty nad nějakou konkrétní doménou symbolům konstant či proměnných, funkcí a predikátů formule. Formule je nazývána jako splnitelná, pokud existuje interpretace, v níž je formule vyhodnocena jako pravdivá. Splnitelnost je primárním rozhodovacím problémem ve FOL.

Formální jazyk FOL je definován jako množina správně formovaných formulí, které jsou splnitelné. Jazyk je *rozhodnutelný*, pokud existuje konečný algoritmus, který korektně rozhoduje, zda libovolné slovo patří či nepatří do jazyka.

FOL obecně není rozhodnutelná, některé teorie však ano. Důležité u teorií (či alespoň některých jejich podmnožin) je zejména to, aby byly rozhodnutelné efektivně, a ne nutně obecně, ale v praktických případech. Díky rozhodnutelnosti lze pak formule automatizovaně analyzovat.

**Definice.** Teorie je definována *značením* a množinou *axiomů*. Značení je množina symbolů konstant, funkcí a predikátů bez konkrétního významu. Axiom je uzavřená FOL formule obsahující pouze prvky ze značení teorie. Formule teorie mohou proti axiomům navíc obsahovat proměnné, logické operace a kvantifikátory.

*Fragment* teorie je její podmnožina přípustných formulí. Častým fragmentem teorií je fragment bez kvantifikátorů[4]. Fragmenty jsou užitečné zejména

[3] Vyšší řády povolují predikáty uvnitř predikátů či funkcí apod.

[4] Tyto formule však stále implicitně obsahují univerzální kvantifikátory pro všechny proměnné.

v případech, kdy jsou lépe rozhodnutelné. Obecně lze říci, že čím limitovanější teorie je, tím má blíže k rozhodnutelnosti[5].

Součástí každé formule teorie jsou implicitně také všechny její axiomy. Proto je nutné vždy uvést, jaká teorie má být použita. Příklady teorií jsou teorie celých či reálných čísel a teorie různých datových struktur (pole, seznam, bitový vektor, fronta, hash tabulka, ...) apod. Základní příklady jsou rozvedeny dále podle [5].

**Teorie rovnosti.** Kromě symbolů konstant, funkcí a predikátů obsahuje jen jediný interpretovaný binární predikát $=$, jehož chování je definováno axiomy:

1. *Reflexivita*: $\forall x : x = x$
2. *Symetrie*: $\forall x, y : x = y \Rightarrow y = x$
3. *Tranzitivita*: $\forall x, y, z : x = y \wedge y = z \Rightarrow x = z$
4. *Funkční kongruence*: $\forall \boldsymbol{x}, \boldsymbol{y} : (\forall i = 1, \ldots, n : x_i = y_i) \Rightarrow f(\boldsymbol{x}) = f(\boldsymbol{y})$ pro všechna kladná přirozená čísla $n$ a $n$-ární funkce $f$.
5. *Predikátová kongruence*: $\forall \boldsymbol{x}, \boldsymbol{y} : (\forall i = 1, \ldots, n : x_i = y_i) \Rightarrow p(\boldsymbol{x}) \Leftrightarrow p(\boldsymbol{y})$ pro všechna kladná přirozená čísla $n$ a $n$-ární predikáty $p$.

Teorie rovnosti je nerozhodnutelná stejně jako FOL, protože povoluje všechna značení (obsahující $=$). Nicméně, fragment bez kvantifikátorů už je efektivně rozhodnutelný.

**Teorie celých čísel.** Existují tři základní teorie celých čísel:

**Peanova aritmetika** má značení $\{0, 1, +, \cdot, =\}$ ($0, 1$ jsou konstanty; $+, \cdot$ binární funkce; a $=$ binární predikát) a následující axiomy:

1. $\forall x : \neg(x + 1 = 0)$
2. $\forall x, y : x + 1 = y + 1 \Rightarrow x = y$
3. $F(0) \wedge (\forall x : F(x) \Rightarrow F(x + 1)) \Rightarrow \forall x : F(x)$
4. $\forall x : x + 0 = x$
5. $\forall x, y : x + (y + 1) = (x + y) + 1$
6. $\forall x : x \cdot 0 = 0$
7. $\forall x, y : x \cdot (y + 1) = x \cdot y + x$

Tyto axiomy definují sčítání, násobení a rovnost přirozených čísel a také *indukci* (axiom 3). Tato teorie bohužel není rozhodnutelná (ani bez kvantifikatorů; na vině je operace násobení) a dokonce není ani úplná[6].

**Presburgerova aritmetika** vychází z Peanovy, ale vynechává operaci násobení, a tedy i axiomy 6 a 7. Tato teorie je již rozhodnutelná, a to dokonce

[5] FOL je též teorie, ale nijak limitovaná — bez axiomů.

[6] Tj. existují v ní formule, které nelze dokázat.

i s kvantifikátory. Operace odčítání a nerovností je možné modelovat[7], a je tedy možné vyjádřit celou teorii celých čísel bez násobení.

**Teorie celých čísel** má stejné vyjadřovací schopnosti jako Presburgerova aritmetika, ale má přirozenější a přívětivější značení: obsahuje všechna celá čísla jako konstanty, operaci odčítání a predikáty nerovností. Také obsahuje unární funkce umožňující používat celočíselné násobky proměnných.

Nadále budou používány dva pojmy: *lineární*, resp. *nelineární* teorie celých čísel jako teorie celých čísel bez násobení, resp. s násobením.

**Teorie reálných čísel.** I zde se teorie dělí na *lineární* a *nelineární* s ohledem na použití operace násobení.

**Nelineární teorie reálných čísel** bývá také označována jednoduše jako teorie reálných čísel. Má značení $\{0, 1, +, -, \cdot, =, \geq\}$ a obsahuje komplexní axiomatizaci zahrnující všechny axiomy:

1. *tělesa* nad $(+, \cdot)$ (tj. axiomy Abelovské grupy nad $(+)$ a okruhu nad $(\cdot)$),
2. úplného uspořádání $\geq$,
3. uspořádaného tělesa (navíc uspořádanost sčítání a násobení),
4. existence kvadratického kořene pro všechny elementy,
5. existence alespoň jednoho kořene všech polynomů lichého stupně.

Tato teorie je rozhodnutelná i s násobením, nicméně asymptotická složitost rozhodovací procedury je dvojnásobně exponenciální.

**Lineární teorie reálných čísel**, také označována jako teorie racionálních čísel[8], omezuje nelineární teorii reálných čísel vyjmutím operace násobení a s tím i axiomů pro násobení a existenci kořenů; k tomu přidává axiom, že neutrální prvek (0) je jediným prvkem s konečným řádem v Abelovské grupě nad $(+)$; a axiom o dělitelnosti prvků (každý prvek je sumou jiného prvku). Horní asymptotická složitost se u této teorie sice nezměnila, ale v průměru je tato teorie efektivně rozhodnutelná, zejména její fragment bez kvantifikátorů.

Teorie mohou být navzájem *kombinovány* (např. teorie polí společně s teorií celých čísel) při splněných určitých podmínek, např. jejich značení by měla být, až na výjimku predikátu $=$, disjunktní (jinak je nutné společné symboly zavést nově). Tato možnost je poměrně důležitá, jinak by bylo zavádění kombinace teorií jako explicitní nové teorie komplikované.

### 1.1.3 Ordinary differential equation (ODE)

Diferenciální rovnice je rovnice pro nějakou *neznámou* funkci a obsahující její derivace, což je běžné pro fyzikální vztahy reálného světa. *Obyčejná* diferenciální

[7] Odčítání převedením na druhou stranu rovnosti, a nerovnosti přičtením nové konstanty do rovnosti.

[8] Důvod je ten, že každá interpretace teorie, vzhledem k jejím axiomům, je ekvivalentní s použitím jak domény reálných, tak racionálních čísel.

rovnice (*ODE*) obsahuje derivace vztažené pouze k *jediné nezávislé proměnné*, což je zpravidla čas [6]. Řešení tohoto speciálního případu je obecně mnohem jednodušší, přesto však stále není obecně možné nalézt analytické řešení, a proto se používají numerické metody [7].

Kromě omezení na ODE dále vymezuji následující vlastnosti: diferenciální rovnice je *prvního řádu*[9], má pevné počáteční podmínky — *Initial value problem (IVP)*, a je *explicitní*[10].

Existuje několik možných formulací tohoto problému, zde je definován jako hledání řešení soustavy rovnic $n$ neznámých funkcí (s výše uvedenými vlastnostmi):

$$\begin{aligned} \dot{\boldsymbol{y}}(t) &= \boldsymbol{f}(t, \boldsymbol{y}(t)) \\ \boldsymbol{y}(t_0) &= \boldsymbol{y_0} \end{aligned} \tag{1.2}$$

kde $t \in \mathcal{R}$ je nezávislá proměnná a $\mathcal{R}$ je množina reálných čísel; $\forall i = 1, \ldots, n$ : $y^i \in \boldsymbol{y} : \mathcal{R} \to \mathcal{R}$ je neznámá diferencovatelná funkce $t$, $\dot{y}^i \in \dot{\boldsymbol{y}}$ je derivace $y^i$ podle $t$, a $f^i \in \boldsymbol{f} : \mathcal{R}^{n+1} \to \mathcal{R}$ je funkce Lipschitz-spojitá v $\boldsymbol{y}$ [11]; $t_0 \in \mathcal{R}$ je počáteční hodnota nezávislé proměnné $t$, která společně s $\boldsymbol{y_0} \in \mathcal{R}^n$ určuje počáteční podmínky. Pro jednoduchost nejsou uvažovány případy, kdy některá funkce není definována na celé $\mathcal{R}$.

Vztahy (1.2) lze přepsat do ekvivalentního tvaru s integrálem [9]:

$$\boldsymbol{y}(t) = \boldsymbol{y_0} + \int_{t_0}^{t} \boldsymbol{f}(\tau, \boldsymbol{y}(\tau))\ \mathrm{d}\tau \tag{1.3}$$

proto bývají někdy numerická řešení ODE nazývány jako *numerická integrace*.

## 1.2 Hybridní systémy

(Dynamický) systém se nazývá *hybridním*, pokud vykazuje jak diskrétní, tak spojité změny. Diskrétní změny je charakterizováno *skoky* (angl. *jumps*), spojité *toky* (angl. *flows*). Skoky jsou obvykle popsány *konečným automatem*, toky pomocí soustav ODE.

Diskrétní systémy se používají pro jejich snadný návrh a analýzu; spojité zejména proto, že popisují procesy z reálného světa (fyzikální, chemické, biologické, ...), jelikož čas je spojitý. Všechny digitální počítače jsou diskrétními systémy s omezenou přesností, je na nich však možné spojité jevy aproximovat.

Hybridní systémy musejí interagovat s vnějším světem, často v reálném čase. Jedná se tedy o *reaktivní* systémy. Obvyklými požadavky na tyto systémy

[9] Rovnice obsahuje pouze první derivace, což však není omezující, neboť každá rovnice vyšších řádů lze přepsat na soustavu rovnic prvního řádu [8][9].

[10] Derivace funkce je řešena explicitně, tj. nevyskytuje se jako argument jiné funkce.

[11] Tento předpoklad podle Picard–Lindelöfova teorému zaručuje, že řešení takové ODE existuje právě jedno; viz. [9].

jsou (kromě jiných) spolehlivost a bezpečnost[12]. Aby mohly být tyto vlastnosti do vysoké míry zaručeny, je nutné využít matematický aparát.

**Hybridní automat.** Hybridní systém lze jako celek matematicky modelovat jako *hybridní automat*. *Stav* hybridního automatu je definován diskrétním řídícím *módem* a spojitými *proměnnými*. Diskrétní změna stavu odpovídá skoku (v konečném automatu), spojitá změna pak toku (průběhu ODE). Módy mohou mít také definovány *invarianty*.

Analýza hybridního systému pak spočívá v rozhodování o množině stavů, zda je dosažitelná či naopak a za jakých podmínek.

Existuje několik nástrojů analyzujících hybridní systémy modelované jako hybridní automaty, ale většinou nejsou založeny na problému SMT. */* Proč? Proč je to lepší/horší? */*

Obě domény samostatně se dnes používají standardně pro modelování systémů a jejich analýzu; k tomu jsou hojně využívány SAT či SMT řešiče pro diskrétní a ODE řešiče pro spojité systémy. Výzvou této práce je obě domény efektivně kombinovat a zároveň využít nástrojů vycházejících z fenoménu problému SAT.

[12]V tomto případě je míněna bezpečnost z hlediska spolehlivého selhání neohrožující majetek či lidi (angl. safety). Bezpečnost ve významu zabezpečení vůči neautorizovanému přístupu (angl. security) je často také důležitá.

Kapitola 2

# Možnosti řešení problematiky

V této kapitole rozebírám možnosti řešení problémů uvedených v kapitole 1 a provádím rešerši existujících řešičů. Uvedené řešiče jsou jak izolované (jen SMT či ODE), tak hybridní (kombinují oba problémy), ale s odlišným typem ODE řešiče, než na jaký jsme cílili. Pro úplnost také uvádím sekci ohledně řešení problému SAT.

U řešičů zmiňuji více či méně podrobně jejich vlastnosti a zdůvodňuji proč jsem je použil či nepoužil.

## 2.1 SAT řešiče

Ač zde uvádím tuto kategorii řešičů, používal jsem je jen nepřímo, neboť jsou součástí SMT řešičů.

Z důvodů implementačních a konvence je většinou vstup do řešičů uváděn v konjuktivní normální formě (angl. CNF), neboli jako konjunkce klauzulí, kde klauzule je disjunkce literálů. Standardně se používá DIMACS-CNF formát.

Většina dnešních SAT řešičů využívá v základu algoritmus Davis–Putnam–Logemann–Loveland (DPLL), který používá několik hlavních operací [10]:

- základní simplifikace klauzulí,
- *substituce* — přiřazení hodnoty proměnné,
- *propagace* — aplikace deduktivních pravidel, zejména pravidla jednotkové klauzule[13],
- *návrat* — navrácení do nějakého předchozího bodu substituce při nalezení konfliktních ohodnocení.

Každý lepší řešič také implementuje nějakou formu učení, které spočívá v přidávání dalších klauzulí na základě průběžně nacházených konfliktů.

[13] Klauzule s jediným literálem vynucuje jednoznačné ohodnocení této proměnné, aby mohla být celá CNF formule splněna.

Známými SAT řešiči jsou např. MiniSAT [11][14], PicoSAT a CryptoMiniSAT. Příklady použití SAT řešičů jsou:

- Bounded Model Checking (BMC),
- funkční testování obvodů:
  logický obvod s injektovanou poruchou je převeden do Booleovské formule a je ověřena její splnitelnost,
- statická analýza kódu programu,
- plánování a grafové problémy

a mnoho dalších. Obecně se však většinou jedná o nějakou formu formální verifikace.

## 2.2 Řešení SMT problému

Jak už bylo zmíněno v sekci 1.1.2, zásadní vliv na výpočet má teorie použitá ve vstupní formuli. SMT řešiče typicky ovládají jen některé teorie a jejich fragmenty, nebo některé rozhodují jen s omezenou efektivitou.

Řešič má za úkol nalézt splňující ohodnocení pro všechny termy vstupní formule, které se nazývá *model*. Výstupem pak je zpravidla `sat` a (volitelně) model. Pokud formule není splnitelná, řešiče většinou umožňují vygenerovat *důkaz* jako certifikát dokládající nesplnitelnost. Výstupem pak je zpravidla `unsat` a (volitelně) důkaz. Také se může stát, že o splnitelnosti vstupu není možné rozhodnout (např. pokud řešič nemá implementovány všechny funkcionality nutné pro daný vstup). Výstupem pak může být např. `unknown`.

Existují dva základní přístupy k řešení SMT problémů: *pilný* (angl. *eager*) a *líný* (angl. *lazy*), nebo i jejich kombinace [12].

**Pilný přístup** soustředí většinu výpočtů do *externího* SAT řešiče tak, že se snaží v *jediném kroku* celou SMT formuli zakódovat do SAT formule (např. celá čísla pomocí bitů jako Booleovských proměnných).

Výkonnost tohoto postupu je zcela závislá na použitém SAT řešiči a nevyužívá zřejmých faktů vázaných k dané teorii (např. komutativita) [1]. Na druhou stranu je z hlediska rozhraní a výpočtu v podstatě nezávislý na použitém SAT řešiči. Je flexibilnější než druhý přístup, protože část specifickou pro teorii tvoří „jen“ optimalizovaný překlad formule, samotný výpočet už ne.

**Líný přístup** spočívá v použití SAT řešiče založeném na DPLL a $\mathcal{T}$-řešiče jako dvou více či méně *spolupracujících komponent* (varianty *online* a *offline* [12]), kde $\mathcal{T}$ je nějaká teorie.

Predikáty teorie (resp. omezení, např. lineární nerovnice) jsou překládány $\mathcal{T}$-řešičem na abstraktní Booleovské literály, které je interní SAT řešič schopen

[14] MiniSAT zvítězil ve všech průmyslových kategoriích v soutěži *SAT 2005 competition* a je často integrován pro svůj minimalistický a snadno rozšiřitelný návrh.

pojmout. V případě, že je taková formule splnitelná (nutná podmínka), $\mathcal{T}$-řešič je použit pro ověření vytvořeného modelu, zda je ohodnocení predikátů splnitelné i v dané teorii $\mathcal{T}$. Tento proces probíhá opakovaně dokud není dosaženo konvergence [10]. Tedy, oba řešiče navzájem intenzivně komunikují a formování Booleovské formule probíhá (typicky) inkrementálně[15] s možností návratů. V případě *online* varianty jsou oba řešiče více propojeny a $\mathcal{T}$-řešič využívá funkce SAT řešiče přímo.

$\mathcal{T}$-řešič musí být navržen speciálně pro danou teorii $\mathcal{T}$, typicky *ad hoc*.

Dalším hlediskem je, zda je SMT řešič jako celek *inkrementální*, který umožňuje dynamické formování vstupních formulí a vícenásobné ověřování splnitelnosti nad různými kontexty. Neinkrementální řešič pracuje jen nad jediným statickým kontextem jakožto celým vstupem. Inkrementální umožňuje průběžně přidávat či odebírat omezení, která se typicky ukládají do *zásobníku*. Ověření splnitelnosti pak lze provést kdykoliv nad obsahem vrcholu zásobníku.

SMT řešiče často pracují nad kombinací více teorií kvůli větší expresivitě. V takovém případě je z hlediska výkonu důležité udržovat teorie v hierarchii a v každém kroku použít jen nezbytně nutnou úroveň [12].

### 2.2.1 SMT-LIB standard

SMT-LIB je iniciativa založená pro účely rozvoje výzkumu a vývoje SMT řešičů, jejíž nejvýznamnější činností je standardizace teorií a vstupně-výstupního jazyka pro řešiče [2]. S tím souvisí udržování komunity vývojářů a souboru standardizovaných výkonnostních úloh (benchmarks), ve kterých jednotlivé týmy soutěží např. v rámci *SMT-COMP*, podobně jako tomu je u komunity SAT řešičů.

SMT-LIB jako teorie označuje teorie v základním znění bez dalších omezení. Pro konkrétní fragment teorie, ve kterém je daná vstupní formule vyjádřena, se používá termín *logika*. Tyto logiky se pak navzájem kombinují či se redukují jejich restrikce. Z pohledu řešiče (konformního s tímto standardem) se operuje pouze s logikami; teorie slouží pouze jako teoretický základ.

Pro odlišení prvků pocházejících z různých teorií se používají *druhy* prvků (angl. *sort*), které připomínají datové typy programovacích jazyků. Proměnné ve formuli jsou označovány jako konstanty[16]; pojmy term a predikát nejsou používány — všechny konstrukty formule FOL jsou vyjádřeny pomocí konstant a funkcí, které jsou případně logického druhu.

Momentálně existuje verze 2 standardu, která definuje hierarchii dílčích logik, z důvodu možnosti aplikace efektivnějších výpočtů pro jednodušší formule, a protože pak lze v rámci izolovaných logik efektivněji srovnávat řešiče navzájem.

[15] To také umožňuje dynamicky přídávat a odebírat formule s omezeními.

[16] Proměnná by mohla vyvolávat dojem, že lze do proměnných, podobně jako v programovacích jazycích, dynamicky přiřazovat hodnoty, což nelze.

Logiky povolují jen některé druhy konstant a funkcí (podle použitých teorií) a případně povolují i definici volných druhů.

**Názvosloví logik.** Standard definuje konvence pro pojmenování jednotlivých logik podle použitých teorií, např.:

- `BV` (bit vectors) — teorie bitových vektorů omezené šířky,
- `IA` (integer arithmetic) — teorie celých čísel,
- `RA` (reals arithmetic) — teorie reálných čísel,
- `IRA` — kombinace `IA` a `RA`,

a jejich fragmentů jako předpony:

1. `QF_` (quantifier-free) — fragment bez kvantifikátorů,
2. `UF` (uninterpreted functions) — fragment povolující použití volných druhů prvků a neinterpretovaných funkcí,
3. `L`, resp. `N` (linear, resp. non-linear) — lineární, resp. nelineární fragment aritmetické logiky.

Příklady některých logik: `BV`, `UF`, `QF_LRA`, `QF_UFNRA`, `UFNIA`, ...

Základní příkazy jazyka jsou deklarace či definice konstant a funkcí, nových druhů, a přidávání formulí do *asercí*. neboli podmínek, které musí být splněny. Ověření splnitelnosti pak spočívá v hledání ohodnocení všech konstant a funkcí splňující konjunkci všech asercí, podobně jako v SAT řešičích.

Jazyk lze použít i pro inkrementální řešiče, pro které lze využít operací přidávání a odebírání asercí ze zásobníku. Ověření splnitelnosti pak vždy probíhá nad vrcholem zásobníku.

Základní vlastnosti standardu verze 2 uvádí např. tento tutoriál [13]. Podrobný popis SMT-LIB standardu verze 2.6 je k nalezení v referenčním dokumentu [14].

### 2.2.2 SMT řešiče

*/* Všechny zde uvedené SMT řešiče používají líný přístup /* jakto? */. */*
Použití SMT řešičů se do značné míry kryje se SAT řešiči, často je nahradily, resp. rozšířily.

**OpenSMT** (konkrétně jeho druhá verze) je inkrementální open-source SMT řešič napsán v jazyce C++, který podporuje standardní iniciativu SMT-LIB [15][16]. Je postaven nad SAT řešičem MiniSAT2. Nástroj byl implementován s důrazem na snadnou rozšiřitelnost o nové $\mathcal{T}$-řešiče, současně však zůstává efektivní[17].

[17] V letech 2008 a 2009 byl oceněn v soutěži *SMT-COMP* jako nejrychlejší open-source SMT řešič ve čtyřech logikách ze SMT-LIB.

OpenSMT používá líný přístup. Jeho architektura je dekomponována do tří hlavních bloků: preprocesor a SAT a $\mathcal{T}$-řešič. $\mathcal{T}$-řešiče mají standardizované rozhraní, které slouží ke komunikaci se SAT řešičem a také vzájemné, je-li použita kombinace více logik, a tedy $\mathcal{T}$-řešičů. $\mathcal{T}$-řešiče lze také přizpůsobovat konkrétním problémům, v případě že je lze řešit efektivněji než v obecném případě.

Řešič lze v aplikacích používat také odděleně jako *černou skříňku* (angl. *black box*), a to buď prostřednictvím volání funkcí API, nebo zpracováním formule jako textového vstupu (např. ve formátu SMT-LIB).

*/* Vyjmenovat logiky */*

**CVC4** je open-source SMT řešič [17]. Stejně jako OpenSMT je navržen pro snadné rozšiřování a poskytuje rozhraní v C++ a také rozhraní textové přes vstupní jazyk, tzn. že nástroj lze použít jak jako knihovnu, tak samostatně.

CVC4 přijímá vlastní vstupní jazyk, nebo standard SMT-LIB verze 1 nebo 2, kde však neposkytuje plnou functionalitu (nelineární aritmetiky nejsou zatím adekvátně podporovány). Má již vestavěno několik základních teorií, např. číselné aritmetiky, bitvektory, řetězce ... Podporuje kvantifikátory a je schopen generovat modely.

*/* 1. v SMT-COMP 2017 */ /* složitější než OpenSMT */ /* Vyjmenovat logiky */ /* [18] */*

## 2.3 Numerické metody řešení ODE

Tyto metody numericky *aproximují* průběh ODE. Obecně používají nějaký *krok*, který určuje vzdálenost mezi sousedními vypočítávanými body. Krok může být fixní či variabilní. Některé metody používají více kroků, což kromě různých vzdáleností také znamená, že hodnota bodů závisí na více než jednom předchozím bodu. Obecně platí, že čím menší je zvolený krok, tím je menší odchylka od exaktního řešení, ale současně vzrůstá výpočetní složitost.

**Značení.** $t_n$ je hodnota nezávislé proměnné $t$ v $n$-tém kroku; $h_n = t_{n+1} - t_n$ je vzdálenost mezi kroky v $n$-tém kroku ($h$ pokud je délka konstantní); $y_n \sim y(t_n)$; $f_n \sim f(t_n, y_n)$.

O metodě je volně řečeno, že je (má přesnost) *řádu* $p$, pokud její odchylka aproximace od exaktního řešení konverguje k $\mathcal{O}(h^{\mathcal{O}(p)})$ [18].

Metody řešení ODE se dělí na *explicitní* a *implicitní*. Explicitní metody počítají každý bod explicitně pouze na základě dosud zjištěných hodnot, např.

$$y_{n+1} = y_n + hf_n \tag{2.1}$$

[18] Jedná se o intiutivní definici, přesné definice řádu metody, lokální a globální chyby a dalších termínů nalezne čtenář např. zde [19].

kdežto implicitní získávají každý bod implicitně z řešení rovnice, která obsahuje i dosud neznámé hodnoty, např.

$$y_{n+1} = y_n + hf_{n+1} \tag{2.2}$$

Explicitní metody jsou časově efektivnější, ale nehodí se pro řešení *rovnic se silným tlumením* (angl. *stiff*)[19], u kterých je výpočet *nestabilní*. Implicitní metody jsou mnohem pomalejší, ale obecně stabilnější a právě vhodné pro tyto těžké rovnice [19]. Míra nestability se odvíjí od nutnosti nastavit co nejmenší krok tak, aby byla odchylka od exaktního řešení přijatelná.

Základní metodou pro řešení ODE je *Eulerova metoda*, která má jak explicitní (vztah (2.1)), tak implicitní (vztah (2.2)) variantu, kde $h$ může a nemusí být konstantní. Metoda vychází ze standardní derivační aproximace [21]:

$$\dot{y}(t) \approx \frac{y(t+h) - y(t)}{h} \tag{2.3}$$

neboli posunu po tečně ke křivce funkce. Tato metoda je poměrně nepřesná, ale je snadno pochopitelná a většina numerických metod z ní vychází [19].

Existují dvě známé rodiny metod pro numerické řešení ODE: *lineární vícekrokové metody* a *metody Runge–Kutta*. Obě skupiny souhrnně nazývám *klasickými numerickými metodami* a jsou uvedeny v následující podsekci. Již existující hybridní řešiče však používají jiné metody než tyto, případně jejich nadstavby. Liší se tím, že na rozdíl od klasických ODE řešičů garantují rozsah chyby aproximace, ale jsou příliš pomalé. Jejich princip je popsán v další podsekci.

### 2.3.1 Klasické numerické metody

Použití těchto metod bylo hlavním cílem této práce, neboť jsou rychlejší než metody použité ve stávajících hybridních řešičích.

Lineární vícekrokové i Runge–Kutta metody sdílejí několik společných rysů:

- řeší IVP ODE prvního řádu,
- průběh funkce počítají na základě jedné a více předchozích hodnot,
- vyskytují se v nich jak explicitní, tak implicitní metody,
- spadá do nich Eulerova metoda,
- garantují konvergenci aproximační chyby ve vztahu k velikosti kroku $h$ a k řádu $p$ [21], nikoliv však její přesný rozsah,
- výstupem jsou páry $(t_n, y_n)$.

[19] Pro rovnice se silným tlumením neexistuje přesná definice. Jsou volně definovány jako takové, kde je pro explicitní metody buď nutné nastavit velmi malou velikost kroku, nebo je řešení nestabilní [20], na rozdíl od implicitních metod, které mohou naopak být stabilní i pro jakoukoli zvolenou velikost kroku [19]. Tyto rovnice často obsahují funkce s několika časovými škálami s rozdílnými granularitami.

Obě skupiny spadají do *obecných lineárních metod* jako speciální případy [19], toto zobecnění ale nebude v tomto dokumentu diskutováno.

#### 2.3.1.1 Lineární vícekrokové metody

Jedná se o obvyklou variantu (obecných) vícekrokových metod. Vícekrokové metody při výpočtech využívají hodnoty několika předchozích kroků, které se uchovávají a mohou být použity i vícekrát. Lineární varianta používá *lineární kombinaci* těchto hodnot [22]:

$$\sum_{j=0}^{k} \alpha_j y_{n+j} = h \sum_{j=0}^{k} \beta_j f_{n+j} \tag{2.4}$$

kde $k$ je počet zpětně sledovaných kroků, $\alpha_j \in \mathcal{R}$ a $\beta_j \in \mathcal{R}$ jsou konstanty, přičemž $\alpha_k \neq 0$ a $\alpha_0 \neq 0 \vee \beta_0 \neq 0$. Pro $\beta_k = 0$ je metoda explicitní, jinak je implicitní. Podle $k$ je konkrétní metoda nazývána jako $k$-kroková.

Funkce $f$ je vyčíslována v pravidelně rozložených bodech (vyskytuje se vždy ve formě $f_n$), což umožňuje zpětné používaní těchto hodnot při větším počtu kroků. Je to hlavní důvod, proč je počet vyhodnocení $f$ obecně menší, než u Runge–Kutta metod, které hodnoty předchozích mezikroků nevyužívají. Pokud je vyčíslení $f$ náročné, pak významně závisí na jejím počtu — v těchto případech jsou tyto metody většinou efektivnější než metody Runge–Kutta v rámci požadované přesnosti. Nevýhodou těchto metod však je, že je nutné prvních $k-1$ kroků spočítat jinou metodou (kromě počátečních podmínek nejsou známy) [21].

Následují příklady těchto metod podle [22] a [21].

**Eulerova metoda** (vztahy (2.1) a (2.2)). Získáme ji dosazením $k = 1$, $\alpha_1 = 1$, $\alpha_0 = -1$ a $\beta_1 = 0$, $\beta_0 = 1$ pro explicitní, resp. $\beta_1 = 1$, $\beta_0 = 0$ pro implicitní variantu, do (2.4). Jedná se o *jednokrokovou* metodu řádu $p = 1$.

**Lichoběžníková metoda:**

$$y_{n+1} - y_n = \frac{h}{2}\left(f_{n+1} + f_n\right) \tag{2.5}$$

($k = 1$, $\alpha_1 = 1$, $\alpha_0 = -1$, $\beta_1 = \beta_0 = \frac{1}{2}$) je *implicitní* a *jednokroková* metoda řádu $p = 2$.

**Adams–Bashforthovy metody** jsou tvaru $\alpha_k = 1$, $\alpha_{k-1} = -1$, $\alpha_{k-2} = \cdots = \alpha_0 = \beta_k = 0$, a $\forall_{j \neq k}\, \beta_j$ jsou zvolena jednoznačně pomocí interpolace polynomem[20] stupně $q$ funkcí $f$ v bodech $t_{n+k-1}, \ldots, t_n$ tak, aby $q + 1 = p = k$. Spadá sem tedy i explicitní Eulerova metoda ($q = 0$, $k = p = 1$, vztah (2.1)).

[20] Jedná se o aproximaci průběhu funkce $y$ pomocí polynomu $P$ tak, aby $y(x_i) = P(x_i)$ pro $x_0, \ldots, x_n$.

Příklady dalších metod:

$$q=1,\ k\ =p=2:\ y_{n+2}-y_{n+1}=\frac{h}{2}\,(\qquad\qquad 3f_{n+1}-\ f_n)$$
$$q=2,\ k\ =p=3:\ y_{n+3}-y_{n+2}=\frac{h}{12}(\qquad 23f_{n+2}-16f_{n+1}+5f_n)$$
$$q=3,\ k\ =p=4:\ y_{n+4}-y_{n+3}=\frac{h}{24}(55f_{n+3}-59f_{n+2}+37f_{n+1}-9f_n)$$

Jsou to efektivní *explicitní* $k$-krokové metody, často používané pro rovnice bez silného tlumení.

**Adams–Moultonovy metody** mají shodný tvar s Adams–Bashforthovými metodami s těmi rozdíly, že jsou *implicitní*, tj. $\beta_k \neq 0$, a $q+1=p=k+1$ s výjimkou pro $q=0$, kde $k=1$ (implicitní Eulerova metoda, vztah (2.2)). Spadá sem i lichoběžníková metoda ($q=k=1, p=2$, vztah (2.5)).

Další příklady:

$$q=k\ =2,\ p=3:\quad y_{n+2}-y_{n+1}=\frac{h}{12}(\qquad 5f_{n+2}+8f_{n+1}-f_n)$$
$$q=k\ =3,\ p=4:\quad y_{n+3}-y_{n+2}=\frac{h}{24}(9f_{n+3}+19f_{n+2}-5f_{n+1}+f_n)$$

**Backward differentiation formula (BDF)** jsou *implicitní* $k$-krokové metody často používané pro rovnice se silným tlumením pro jejich vlastnosti stability (ač jen pro $k \leq 6$).

Jejich tvar je $\beta_{k-1}=\ldots=\beta_0=0$, ostatní koeficienty ($\beta_k$ a $\forall\alpha_j$) jsou zvoleny tak, aby $q=p=k$ (opět pomocí interpolace, tentokrát ale pro funkce $y$). Spadá sem opět implicitní Eulerova metoda ($q=k=p=1$ [21]).

Příklady:

$$q=k\ =p=2:\qquad 3y_{n+2}-4y_{n+1}+\ y_n=2hf_{n+2}$$
$$q=k\ =p=3:\quad 11y_{n+3}-18y_{n+2}+9y_{n+1}-2y_n=6hf_{n+3}$$

#### 2.3.1.2 Runge–Kutta metody

Tyto iterativní metody vycházejí z aproximace pomocí rozvoje Taylorova polynomu, které jsou ale pomalé z důvodu požadavku na výpočet derivací vyšších řádů [21]. Runge–Kutta metody toto obchází vícenásobným vyčíslováním funkce

[21] Pro vztah (2.2) jakožto Adams–Moultonovy metody platilo $q=0$, tentokrát se však jedná o polynom na levé straně rovnosti (2.4), tedy $q=1$.

$f$ v několika bodech (mezikrocích) z intervalu $[t_n, t_{n+1}]$. Tím je dosaženo vyšších řádů přesnosti $p$.

Přesto se jedná o metodu *jednokrokovou*, kde každý krok sestává z několika mezikroků, *fází*. Jedná se o to, že hodnoty mezikroků jsou obecně různé a nemohou být znovu využívány tak, jak tomu je u vícekrokových metod, obecně totiž platí, že po každém kroku Runge–Kutta metod jsou všechny mezikroky zapomenuty.

To může činit potíže, pokud je vyčíslení funkce $f$ náročné, neboť je nutné ji počítat často. Nicméně, tyto metody mají odlišné vlastnosti stability od vícekrokových metod a jejich použití může být mnohdy výhodnější, zejména u rovnic se silným tlumením. Dále tyto metody umožňují lepší průběžné řízení chyby aproximace či adaptaci na ni a mohou být použity jako základ řešičů s garancí rozsahu chyby [8].

Obecná $s$-fázová Runge–Kutta metoda je definována podle [21][8] vztahy

$$\begin{aligned} y_{n+1} &= y_n + h \sum_{i=1}^{s} b_i k_i \\ k_i &= f(t_n + c_i h, z_i) \\ z_i &= y_n + h \sum_{j=1}^{S_i} a_{i,j} k_j \\ c_i &= \sum_{j=1}^{i-1} a_{i,j} \end{aligned} \tag{2.6}$$

kde $\forall b_i, c_i, a_{i,j}$ jsou konstanty plně charakterizující konkrétní Runge–Kutta metodu, uspořádané do tzv. *Butcherovy tabulky*:

$$\begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\ c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} \tag{2.7}$$

Jejich hodnoty jsou hledány podle Taylorova rozvoje a podle požadovaného řádu metody $p$. Není znám přesný vztah pro $p$ a $s$, ale obecně platí $p \leq s$.

Hodnota $S_i$ ve vztahu pro $z_i$ rozlišuje typ metody:

- *explicitní*: $S_i := i - 1$, tj. $\forall_{i,j}\, i \leq j : \; a_{i,j} = 0$; $c_1 = 0$; $\forall k_i$ závisí pouze na $k_j,\ j < i$; Butcherova tabulka je v striktně dolním trojúhelníkovém tvaru s nulovou diagonálou; z toho vyplývá $z_1 = y_n, k_1 = f_n$;
- *implicitní*: $S_i := s$; $\exists_{i,j}\, i \leq j : \; a_{i,j} \neq 0$.

Příklady Runge–Kutta metod:

**Eulerova metoda** (vztahy (2.1) a (2.2)) — $s = p = 1$, $b_1 = 1$ a $c_1 = a_{1,1} = 0$ pro explicitní; $c_1 = a_{1,1} = 1$ pro implicitní metodu:

$$\begin{aligned} y_{n+1} &= y_n + hk \\ k &= f(t_n + h, y_n + hk\ ) = f(t_{n+1}, y_{n+1}) \\ \Rightarrow y_{n+1} &= y_n + hf_{n+1} \end{aligned}$$

Jejich Butcherovy tabulky:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \qquad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

**Lichoběžníková metoda** (vztah (2.5)) — $s = p = 2$, je implicitní.

Tabulka:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

**Klasická Runge–Kutta metoda (RK4):**

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= f_n \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned} \tag{2.8}$$

je *explicitní* metoda s parametry $s = p = 4$.

Tabulka:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Existují také *adaptivní* Runge–Kutta metody, které v každém kroce počítají odhad chyby aproximace, podle níž se dynamicky mění délka kroku $h_n$.

#### 2.3.1.3 ODE řešiče

Následují řešiče, které některé z uvedených klasických metod implementují.

**SUNDIALS** (*SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers*) je nástroj napsán v jazyce ANSI C [23]. Pracuje s třídami datových vektorů, nad kterými lze vytvářet uživatelsky definované datové struktury a operace v rámci aplikačního rozhraní. Má implementovány výchozí struktury s předdefinovanými operacemi jak pro sériové prostředí, tak i paralelní se sdílenou (*OpenMP*) nebo distribuovanou (*MPI*) pamětí. Veškerá paralelizace je obsažena pouze v rámci specifických operací nad danými vektory (a lze ji rovněž uživatelsky implementovat), a tudíž není rozlišováno mezi sériovým a paralelním kódem aplikace.

SUNDIALS sestává z více komponent. Diskutuji pouze jedinou z nich — *CVODE*, který je určen pro řešení ODE. Další komponenty řeší také diferenciální algebraické rovnice (*DAE*) — *IDA* — a nelineární algebraické systémy — *KINSOL*. CVODE je výsledkem přepsání řešiče *VODE* z Fortranu do C, počínaje už rokem 1993 [24].

*/* CVODES — includes forward and disjoint sensitivity analysis */*

**odeint** je flexibilní C++ knihovna pro numerické řešení ODE [25]. Je navržena v duchu šablonového metaprogramování (*TMP*) — veškeré její numerické algoritmy jsou nezávislé na použitých datových kontejnerech. Také díky komfortnímu aplikačnímu rozhraní je implementace numerických simulací rychlá a snadná.

Knihovna je součástí rodiny C++ knihoven *Boost* [26]. Obsahuje pouze hlavičkové soubory a umožňuje změnit chování numerických operací — tímto způsobem lze např. použít odeint se SIMD operacemi [7].

*/* gsl */*

### 2.3.2 Garantovaná řešení

Tato řešení využívají *intervalovou aritmetiku*[22], což umožňuje jednak specifikovat počáteční podmínky s nějakou nejistotou jakožto intervalový rozsah, druhak obalení úseků výsledku do intervalových uzávěrů, tzn. že meze odchylky aproximace od exaktního řešení jsou přesně známy. Taková řešení se nazývají jako *garantovaná* (angl. *guaranteed* nebo *validated*). Některé zdroje tento problém formulují jako problém s intervalovými počátečními podmínkami (*Interval* IVP, *IIVP*) [8], některé setrvávají u názvu IVP. Garance rozsahu řešení je klíčovým rozdílem těchto metod od klasických řešičů ODE ze sekce 2.3.1.

[22] Nejsou nám známy metody s garantovaným rozsahem aproximační chyby založená na jiném principu, než je intervalová aritmetika.

**Značení.** $[t_n]$ je interval $[t_n, t_{n+1}]$ délky $h_n$; $[y_n]$ je uzávěr $[y_n^{min}, y_n^{max}]$ v bodě $t_n$ (tj. garantované meze řešení v $t_n$); $[\tilde{y}_n]$ je uzávěr $[\tilde{y}_n^{min}, \tilde{y}_n^{max}]$ pro celé $[t_n]$.

Celý výpočet je rozdělen do kroků v bodech $t_n$. Je-li metoda jednokroková, je její průběh následující [8]:

1. v každém kroku se operuje nad intervalem $[t_n]$,
2. nejprve je hledán volnější uzávěr $[\tilde{y}_n]$,
3. $\forall t \in [t_n]$ je garantována existence $\forall y_n \in [y_n]$, přičemž $[y_n] \subseteq [\tilde{y}_n]$,
4. délka $h_n$ je největší možná v souladu s garantovaným řešením,
5. výpočet $[y_{n+1}]$.

Výstupem jsou trojice $(t_n, [y_n], [\tilde{y}_n])$ obsahující uzávěry (namísto diskrétních hodnot).

Tyto metody používají interní ODE řešiče stávajících hybridních řešičů, které jsou uvedeny v následující sekci.

## 2.4 Hybridní řešiče

Řešiče, které umí analyzovat modely hybridních systémů, již existují, nicméně nezachází dobře s praktickými úlohami z reálného světa. A to proto, že používají řešiče pro ODE s intervalovou aritmetikou, která je zbytečně přesná a ve výsledku pomalá, neboť je exaktní ve smyslu zaručení rozsahu chyby aproximace (viz. sekce 2.3.2).

Našim cílem bylo sestrojit řešič, který nemusí být tolik přesný, ale dokáže analyzovat modely i rozsáhlých systémů v únosně krátké době, a tím by byl použitelný i v praxi, použitím klasických numerických metod (viz. sekce 2.3.1).

Zkoumané hybridní řešiče nám však posloužily jako zdroj cenných informací pramenících z kombinace diskrétního a spojitého modelu a také nás inspirovaly ve volbě vstupního jazyka. S těmito řešiči budou také srovnány výsledky našeho produktu.

*/* [27] */*

**iSAT-ODE** „kombinuje *iSAT*, který řeší rozsáhlé Boolovské kombinace aritmetických omezení, s rovnicemi ODE“ [28][29].

K řešení ODE používá interní nástroj *VNODE-LP* [30], který se nejprve pokouší dokázat, že existuje jediné řešení problému, a poté hledá meze, do kterých toto řešení spadá.

Podobně jako v našem případě kombinuje oba nástroje odděleně, tj. oba jsou samostatně použitelné na svou podmnožinu úloh.

Projekt nezveřejnil zdrojové kódy, přístupný je jen dynamicky linkovaný binární soubor s externími závislostmi a je stále ve stádiu vývoje.

*/* Zatím jsem nebyl schopen spustit kvůli chybějícím starým knihovnám. */*

**dReal** je nástroj napsán v jazyce C++, který rozhoduje, zda je vstupní formule nesplnitelná (`unsat`), nebo $\delta$-splnitelná ($\delta$-`sat`) [31]. Nesplnitelnost je rozhodnuta exaktně a doložena důkazem; $\delta$-splnitelnost je numericky aproximována (resp. exaktně rozhodnuta na zjednodušené formuli) s přesností $\delta$ (racionální číslo). */* Přesněji: řeší exaktně 'delta-perturbated' formule, nevím zda je to přepsáno správně. */*

dReal je postaven nad některými existujícími nástroji, zejména OpenSMT2 [15] a MiniSAT [11] a na straně diferenciálních rovnic pak CAPD (viz. [32]), který počítá intervalové uzávěry ODE. Zpracovává nelineární logiky reálných čísel, zejména nad polynomy, trigonometrickými či exponenciálními funkcemi, rozšířené o ODE.

Integrace nástrojů je v implementaci řešena interně. Základ tvoří lineární $\mathcal{T}$-řešič v OpenSMT, který je doplněn o nelineární logiku a ODE. Konkrétně rozšiřuje logiku `QF_LRA` na `QF_NRA` a ještě o diferenciální rovnice, nazvanou `QF_NRA_ODE`. Z hlediska jazyka spočívá rozšíření v přidání několika málo příkazů pro definice ODE, nastavení invariant, propojení diskrétních stavů s ODE apod. Nad tímto vstupem operuje řešič přímo, tj. SMT a ODE část se nespouští zvlášť v nezávislých komponentách. Tento vstupní jazyk [32] byl hlavním zdrojem naší inspirace při návrhu vlastního vstupního jazyka.

Program také používá vlastní specifikační jazyk, ze kterého se generují nástrojem dReach SMT formule podle zvoleného počtu kroků[23]. Tento předstupeň je lépe lidsky čitelný a navíc brání chybám vzniklým z ručního vytváření rozsáhlých SMT formulí.

V projektu je zahrnuto několik výkonnostních úloh ve vstupních jazycích dReal (specifikace i SMT formule). */* např. dron se očividně počítá hodně dlouho; v testech (ctest) několik úloh timeoutuje */*

*/* [33] */*

[23] dReal tento proces nazývá přímo jako BMC.

Kapitola 3

# Návrh zvoleného řešení

V této kapitole rozebírám teoretický návrh řešení problému zvolený před vlastní implementací. Zatím neuvádím implementační detaily a konkrétní používané nástroje a programovací jazyky. Nejprve popíši specifikaci celého nástroje a jeho vstupy (vstupní jazyk, v němž budou přijímány textové vstupy) a výstupy. Následně uvedu softwarový model celého řešiče.

## 3.1 Specifikace nástroje

Nástroj má fungovat jako řešič kombinace dvou problémů: problému SMT podle přístupů diskutovaných v sekci 2.2, a problému numerického řešení diferenciálních rovnic s použitím klasických numerických metod, jak bylo diskutováno v sekci 2.3.1. Tvar počátečních podmínek není omezen, avšak není-li množina podmínek konečná (tj. např. intervaly), doba výpočtu není definována a zpravidla neterminuje. Intervaly však lze aproximovat výčtem hodnot z intervalu.

Nástroj má sloužit jako *prototyp* odlišného přístupu ke zkoumání hybridních modelů systémů než dosavadní řešiče uvedené v sekci 2.4. Hlavním účelem práce je *srovnání* s některým stávajícím řešičem z hlediska efektivity i jiných vlastností. Neformálním požadavkem pak je, aby pro alespoň některou podmnožinu úloh, s nižšími požadavky na přesnost, byl náš prototyp rychlejší než stávající řešiče, a to i přesto, že by nebyl příliš optimalizován, jelikož náš přístup klade menší požadavky na přesnost a měl by být výpočetně výrazně méně náročný.

Nástroj bude přijímat jako vstup textový soubor spadající do námi specifikovaného vstupního jazyka. Výstupem bude zejména příznak úspěchu a volitelně také nějaká forma výsledných dat. Vstupy a výstupy jsou podrobněji popsány ve vlastních podsekcích.

Jedním z případů užití nástroje je BMC (viz. sekce 1.1.1), u kterého se velikost vstupu a délka výpočtu odvíjí od zvoleného počtu kroků ověřování modelu. Kroky jsou v tomto případě odděleny skoky — změnami diskrétního stavu systému — ke kterým dochází při porušení nějakého *invariantu* vázaného

k aktuálnímu stavu. Jedná se o systém *řízený událostmi* (angl. *event trigerred*). Nemusí tedy být předem zřejmé, ve kterých časových okamžicích bude docházet k integraci a ve kterém výpočet skončí. Tento způsob používá např. nástroj dReal (viz. sekce 2.4).

Náš řešič ale s takovými kroky nepracuje (alespoň ne v této verzi), nýbrž pracuje s předem specifikovanými časovými okamžiky, mezi kterými dochází k integraci po předem danou dobu, a poté se mění stav systému závisle na výstupech integrace pomocí ověření splnitelnosti SMT řešičem. Jedná se tedy o systém *řízený časem* (angl. *time trigerred*)[24]. Tyto úseky výpočtu budu označovat jako *fáze*. Náš řešič kriticky závisí na zvoleném rozložení fází, jak z hlediska přesnosti, tak z hlediska délky výpočtu. Pokud změny stavů modelu závisí na invariantech integrovaných funkcí, je nutné zvolit délku fází co nejnižší, aby bylo porušení invariant detekováno co nejdříve. Invarianty totiž nejsou kontrolovány v průběhu integrace. V opačném případě, nebo pokud není porušení invariant kritické, postačují vyšší délky fází, čímž se urychluje výpočet.

Nástroj není koncipován jako konečný produkt, není příliš uživatelsky přívetivý a může obsahovat řadu chyb. Nástroj má sloužit pro účely experimentování s navrženým způsobem řešení úloh, a dále buď jako zdroj inspirace pro vývojáře průmyslového nástroje, anebo přímo jako postupně se vyvíjející projekt na bázi stávajících otevřených zdrojových kódů.

### 3.1.1 Vstupní jazyk

Vstupním jazykem je záhodno postihnout použití SMT formulí a současně umožnit definovat ODE a propojit je s diskrétními SMT stavy. Vycházel jsem z jazyka SMT-LIB (viz. sekce 2.2.1) a ze vstupního jazyka nástroje dReal (viz. sekce 2.4). Použité názvosloví vychází z SMT-LIB.

Ač je náš vstupní jazyk podobný na ty referované, není s nimi kompatibilní, zejména *není konformní* se standardem SMT-LIB, který je relativně robustní, umožňuje nastavovat parametry pro řešič, podporuje inkrementální operace, apod. Také definuje výstupní jazyk. Náš řešič sice interně tento jazyk hojně využívá, ale svůj vstup omezuje jen na některé části. Vstupní specifikace modelů musí spadat do některé *teorie reálných čísel* (konkrétní logiky jsou uvedeny dále). Tyto logiky jsou pochopitelně rozšířeny o ODE. Nad těmito vstupy lze provést pouze neinkrementální ověření splnitelnosti. Většina zodpovědnosti pojená se vstupní specifikací SMT části vstupu je přímo delegována na SMT řešič, včetně kontroly validity vstupu.

Jazyk používá plně uzávorkovanou prefixovou notaci[25]. Podmnožina příkazů týkající se jen specifikace SMT formulí je převzata z SMT-LIB a k nim jsou

[24] V našem případě však (pochopitelně) odpadá výhoda systémů řízených časem oproti těm řízených událostmi, že je porucha detekována na straně přijímače, jelikož zde pracujeme s „bezporuchovým“ řešičem.

[25] Tato syntaxe je známa zejména z jazyka *Lisp*.

ortogonálně doplněny příkazy týkající se nadstavby o ODE. Obě skupiny jsou popsány ve zvláštních podsekcích.

Vstupní jazyky jsou obecně definovány tak, aby byly pokud možno nezávislé na konkrétně použitých řešičích.

**Značení.** Znaky `<>` nejsou součástí syntaxe a ohraničují či seskupují argumenty (nejsou-li již nějak ohraničeny); `*` značí, že dotčený řetězec se může opakovat vícekrát nebo být prázdný; `+` je jako `*`, ale zakazuje prázdný řetězec; `|` značí více možností pro jednu pozici argumentu.

#### 3.1.1.1 Syntaxe jazyka

Vstupní jazyk je sekvence *tokenů*, *výrazů*, bílých znaků a komentářů. U znaků abecedy se rozlišuje velikost písmen.

**Bílé znaky.** Povolenými bílými znaky jsou:

| Název | mezera | tabulátor | nová řádka |
|---|---|---|---|
| ASCII | 32 | 9 | (13 + )10 |

S výjimkou oddělení dvojic tokenů jsou bílé znaky ignorovány.

**Komentář.** Jako komentář je interpretován každý úsek řádku začínající znakem `;` až po konec řádku. Jejich obsah je ignorován.

**Token** je sekvence znaků závisle na typu tokenu, vždy však bez bílých znaků, které slouží jako jejich oddělovače. Tokeny se dělí na *identifikátory* a *literály.*

**Identifikátory** sestávají z alfanumerických znaků a znaků

`+ - * / ^ = < > _ . ?`

Musí být předem deklarovány, definovány nebo rezervovány a nesmí začínat číslicí[26]. Reprezentují buď *příkazy* (pak se vždy jedná o rezervovaný token; jsou interpretovány výhradně interně v řešiči), nebo *funkce*, které mohou být i uživatelsky definované, či *druh* prvků, výrazů apod. (angl. *sort*). Speciálním případem funkce je *konstanta*, která nemá žádné argumenty. Pojem proměnných se nepoužívá, neboť hodnotám identifikátorů nelze dynamicky přiřazovat nové hodnoty, stejně jako v SMT-LIB.

**Literály** jsou bezejmenné konstanty nějakého druhu. Numerické obsahují číslice a případně desetinnou tečku (`.`) nebo záporné znaménko (`-`)[27]; na začátku nejsou povoleny přídavné `0` a kladné znaménko; desetinná čísla musí

[26] Vzhledem k jen minimálním restrikcím na název identifikátoru je vhodné, aby se uživatel vyvaroval zavádějících názvů, např. obsahujících symboly operátorů (`a<b` apod.), a aby důsledně odděloval tokeny bílými znaky nebo do výrazů.

[27] Záporné literály v SMT-LIB povoleny nejsou, proto je nutné provést transformaci na výraz s unární funkcí `-` a kladným literálem.

obsahovat číslici před i po desetinné tečce; není podporován semilogaritmický tvar. Booleovské literály jsou `true` a `false`.

**Výraz** je vždy uzavřen v závorkách:

```
( <<token> | <expr>>* )
```

kde `<expr>` je vnořený výraz a `<token>` token. Pokud je výraz umístěn v kořenové úrovni vstupu, pak se musí jednat o příkaz. V příkazu musí být prvním elementem výrazu token s názvem příkazu. Obecné výrazy toto omezení nemají, ale pokud se jedná o funkci, platí pro ni totéž co pro příkazy. Příkazy nemusí mít druh návratové hodnoty, funkce ano.

(Bílé znaky a komentáře nejsou v sekvenci zahrnuty; při vyhodnocení je jejich obsah ignorován.)

#### 3.1.1.2 SMT konstrukty

Je použita pouze podmnožina konstruktů týkajících se povolených teorií reálných čísel. Jedná se o druhy výrazů a o rezervované příkazy a funkce.

**Druhy prvků:**

- `Bool` — logický typ,
- `Real` — typ reálných čísel.

Celočíselný druh není akceptován; pro diskrétní konstanty je nutno využít výhradně druh `Bool`, typicky pro diskrétní stav systému, který je konečný.

**Rezervované funkce.** Zahrnuty jsou následující funkce (resp. operátory) se standardní sémantikou:

- Unární: `not`
- Binární: `/`
- $n$-ární:
  - levá asociativita:
    - $n \geq 1$: `-` `and` `or`
    - $n \geq 2$: `+` `*`
  - pravá asociativita, $n \geq 2$: `=>`
  - se zřetězením, $n \geq 2$: `=` `<` `>` `<=` `>=`

a dále tyto funkce:

- `distinct` — $n$-ární funkce s $n \geq 2$, která vrací nerovnost všech dvojic prvků,
- `ite` — ternární funkce s prvním argumentem druhu `Bool`, který když je pravdivý, vrací se druhý argument, jinak třetí argument.

**`set-logic`** nastavuje logiku použitou v SMT řešiči:

```
(set-logic <logic_name>)
```

kde `<logic_name>` je jedna z následujících logik teorií reálných čísel s volnými funkčními symboly, podle SMT-LIB:

- `QF_UFLRA` — lineární bez kvantifikátorů,
- `QF_UFNRA` — nelineární bez kvantifikátorů,
- `UFLRA` — lineární s kvantifikátory.

Příkaz smí být volán nejvýše jednou a musí předcházet všem ostatním uvedeným příkazům. Není-li příkaz uveden, je jako výchozí logika zvolena `QF_UFLRA`.

Pokud to implementace umožňuje, smí být také podporovány zmíněné logiky bez volných funkčních symbolů (názvy jsou bez znaků `UF`). Pak je jako výchozí logika volena `QF_LRA`.

**`declare-fun`** slouží k deklaraci nové funkce (resp. konstanty) bez její interpretace. Je tvaru

```
(declare-fun <fun_name> (<arg_sort>*) <sort>)
```

Argumenty příkazu:

- `<fun_name>` — název identifikátoru funkce,
- `<arg_sort>*` — výčet identifikátorů druhů argumentů funkce (prázdné v případě konstanty),
- `<sort>` — identifikátor druhu návratové hodnoty.

Příklady:

```
(declare-fun y (Real) Real)
(declare-fun empty? () Bool)
```

Funkce a konstanty jsou deklarovány globálně a mohou být za místem deklarace libovolně používány uvnitř dalších funkcí.

**`define-fun`** rozšiřuje[28] `declare-fun` o definici funkce:

```
(define-fun <fun_name> ((<arg> <arg_sort>)*) <sort> <expr>)
```

se shodnými argumenty kromě:

- `(<arg> <arg_sort>)*` — výčet párů identifikátorů názvu argumentů funkce a jejich druhů,
- `<expr>` — výraz nebo token definující chování funkce s druhem návratové hodnoty `<sort>` a (ne nutně) obsahující jednotlivé argumenty `<arg>`.

Příklad:

```
(define-fun v ((s Real) (t Real)) Real (/ s t))
```

[28] Každá funkce je buď jen deklarována, nebo definována, ne obojí.

**assert** zavádí formule modelu, které musejí být splněny:

```
(assert <expr>)
```

kde <expr> je výraz nebo token s druhem návratové hodnoty Bool. Příklad:

```
(assert (or (= mode a) (= mode b) ))
```

#### 3.1.1.3 Konstrukty ODE

V kontextu ODE lze zjednodušovat syntaxi příkazů s následujícími pravidly:

- název nezávislé proměnné v ODE je vždy t,
- druhy funkcí i jejich derivací a nezávislé proměnné je Real.

Tyto skutečnosti nebudou nadále zmiňovány.

Vstup může obsahovat vícero závislých či nezávislých diferenciálních rovnic, označovaných jako ode. Každá ode dále sestává z jedné či více variant derivací, z nichž v každé fázi je pro každou ode platná právě jedna varianta. Varianty derivací budou označovány jako dt. dt umožňují volit různé předpisy pro derivace neznámých funkcí závisle na aktuálním stavu celého systému.

**Nové druhy prvků:**

- Dt — druh určující zvolenou variantu derivace dt.

**Nové rezervované funkce.** SMT logiky neumějí dobře zacházet s některými nelineárními reálnými funkcemi, k nimž lze využít ODE řešič a namísto s funkcemi pracovat s konstantami, jimž jsou přiřazeny výsledky integrování. Přidány jsou následující unární funkce:

```
abs  sqrt  cbrt  sin  cos  tan  exp  ln
```

a binární funkce (resp. operátor): ^
Tyto však mohou být využity pouze uvnitř příkazu define-dt (viz. dále).

**define-dt** slouží k definici dt, tj. výrazu popisujícího variantu derivace funkce, a současně k deklaraci ode neznámé funkce u první zmíněné varianty dt. Všechny dt musí v rámci ode sdílet stejnou signaturu (viz. dále).

Tvar příkazu:

```
(define-dt <fun_name> <dt_name> (<arg>*) <expr>)
```

s argumenty:

- <fun_name> — název ode, tj. identifikátoru neznámé funkce nezávislé proměnné t obsahující všechny varianty dt,
- <dt_name> — název dt, tj. identifikátoru varianty derivace funkce <fun_name>,
- <arg>* — identifikátory argumentů výrazu, které jsou druhu Real a musí být *shodné* u všech variant dt; *neobsahují* funkci <fun_name> a

nezávislou proměnnou `t`, které jsou zahrnuty implicitně; mohou obsahovat i identifikátory jiných `ode`, více viz. níže,

- `<expr>` — výraz nebo token popisující tvar derivace funkce, který může obsahovat funkci `<fun_name>` a nezávislou proměnnou `t` [29] a jednotlivé argumenty `<arg>`. Na rozdíl od SMT funkcí `<expr>` nepřijímá globální funkce a konstanty.

Příklady:

```
(define-dt x dx () 1)
(define-dt y dy_on () (- (* (/ 3 t) y) 2))
(define-dt z dz_a (k) (+ (/ 1 z) k))
```

Identifikátory `<fun_name>` je nutné používat výhradně uvnitř `int-ode` příkazů. Identifikátory `<dt_name>` jsou zavedeny jako konstanty druhu `Dt`, které lze používat uvnitř příkazů `assert` pro účely propojení konstant druhu `Dt` se stavem systému.

Argumenty `<arg>` vstupují do výrazu `<expr>` jako konstanty druhu `Real` jakožto počáteční podmínky integrace a také jako vstupní hodnoty v každé fázi integrace. Pokud `<arg>` pochází z konstanty, její hodnota se nemění; pokud se však jedná o identifikátor některé `ode` (pocházející z `<fun_name>`), její hodnota se průběžně mění, jelikož jsou všechna ODE integrována synchronně. Tímto způsobem se definují soustavy více ODE[30], př.:

```
(define-dt x dx (y) (+ x y))
(define-dt y dy (x) (- x y))
```

`int-ode` obaluje výraz integrace neznámé funkce v konkrétní fázi a dosazuje do `ode` hodnoty či konstanty. Návratová hodnota výrazu je druhu `Real` a lze jej používat uvnitř příkazů `assert`.

Tento příkaz se chová jako funkce a musí být umístěn v místě, kde jsou funkce povoleny, což např. není vrcholová úroveň vstupu.

Na rozdíl od příkazu `define-dt` pracuje tento výhradně s identifikátory, ne s obecnými klíči. Slouží k tomu, aby dosazoval do rovnic definovaných příkazem `define-dt` konkrétní počáteční a koncové hodnoty a vybíral některou variantu `dt`.

Tvar příkazu:

```
(int-ode <fun_name> <dt> (<init> <t_1> <t_2>) (<arg_val>*))
```

s argumenty:

- `<fun_name>` — název `ode`, tj. identifikátor neznámé derivované funkce zavedený příkazy `define-dt`,

[29] Funkce se uvádí bez závislosti na `t`, tj. jako konstanta.

[30] Pokud chcete z nějakého důvodu použít funkci některé `ode` jako *konstantní* vstup pro jinou `ode`, zvolte pro tento argument `<arg>` název odlišný od `<fun_name>`; vstupní konstanty pro příkaz `int-ode` zůstávají stejné.

- `<dt>` — konstanta druhu `Dt`, která určuje některou z variant derivací `dt` definovaných příkazy `define-dt`[31],
- `<init>` — počáteční hodnota funkce `<fun_name>` v bodě `<t_1>`,
- `<t_1> <t_2>` — počáteční a koncová hodnota nezávislé proměnné `t`,
- `<arg_val>*` — počáteční hodnoty argumentů druhu `Real` předané výrazu zvolené varianty derivace.

Názvy všech vstupních konstant mohou být libovolné identifikátory, jejich struktura a návaznosti jsou zodpovědností uživatele. Příklady:

```
(int-ode x dx_0 (x_0 t_0 t_1) ())
(int-ode x dx_1 (x_1 t_1 t_2) ())
(int-ode y dy_1 (y_1 t_1 t_2) ())
(int-ode y der_25 (var-3 tt_5 xy56) ())
(int-ode z dz_1 (z_1 t_1 t_2) (k))
(int-ode z dz_2 (z_1 t_1 t_3) (k))
```

**`define-ode-step`** definuje (počáteční) velikost kroku v interním ODE řešiči:

```
(define-ode-step <h>)
```

kde `<h>` je konstanta druhu `Real`.

#### 3.1.1.4 Struktura a použití jazyka

V této podsekci je uveden tvar doporučené struktury vstupu, který by validně popisoval model hybridního systému a umožňoval jeho analýzu našim nástrojem.

Vzhledem k tomu, že se v jazyce nevyskytují žádné proměnné, není možné, aby se průběh stavu systému v rámci jednoho ověření splnitelnosti dynamicky měnil — výsledkem je vždy statické ohodnocení. Jelikož je vstup statický, je pro modelování průběhu nutné použít mnoho konstant o předem známém počtu.

Rozložení fází je určeno obecně podle navazujících časových mezí příkazů `int-ode` (hodnoty konstant nezávislé proměnné `t`).

Následují jednotlivé sekce, které by se měly objevit ve vstupech. Kromě těchto smí uživatel používat i další SMT konstrukty tohoto jazyka.

**Deklarace a inicializace konstant.** Všechny konstanty musejí být deklarovány a konstanty počátečních podmínek musí být i definovány. Každé konstantě se typicky dává jako přípona číslo fáze, ale řešič na to nebere žádný ohled.

[31] Nejedná se o konstanty pocházející z příkazu `define-dt`, ale o pomocné konstanty, které jsou ohodnoceny SMT řešičem na základě asercí se stavem modelu.

Je vhodné deklarovat konstanty nezávislé proměnné `t`, průběhů neznámých funkcí, diskrétních stavů a konstant voleb variant derivací `dt`[32].

Intervalové počáteční podmínky lze aproximovat pomocí logického součtu několika rovností.

Příklad:

```
;; Literals definition
(define-fun t0   () Real 0)
(define-fun y0_0 () Real 1) (define-fun y0_1 () Real 2)
(define-fun run0 () Bool false)
;; Constants declaration
(declare-fun t_0    () Real) (declare-fun t_1    () Real)
(declare-fun y_0    () Real) (declare-fun y_1    () Real)
(declare-fun run_0  () Bool) (declare-fun run_1  () Bool)
(declare-fun dy_0   () Dt)
;; Initial conditions
(assert (and (= t_0 t0) (= run_0 run0)
             (or (= y_0 y0_0) (= y_0 y0_1))
))
```

**Definice derivací funkcí** se provádí pomocí příkazů `define-dt`. Příklad:

```
(define-dt y dy_run  ()    1 )
(define-dt y dy_idle () (- 1))
```

Identifikátory variant derivací `dt` *nesmí* kolidovat s konstantami volených variant v jednotlivých fázích, př.:

```
(declare-fun dy_0   () Dt) (declare-fun dy_1   () Dt)
;; ...
(define-dt y dy_1   () 1)  ;; conflict !!
```

**Invarianty** znamenají zavedení podmínek, které musí být splněny *mezi* všemi fázemi, ale mohou být porušeny v průběhu integrace. Mohou a nemusí být závislé na aktuálním stavu systému. Pro stanovení podmínek pro konstanty závisle na konkrétním diskrétním stavu systému doporučujeme použít až část definice skoků uvedenou dále.

Pokud to implementace *explicitně* sama neprovádí, je nutné omezit všechny konstanty jednotlivých fází druhu `Dt` pouze na výčet možných variant derivací z příkazů `define-dt`.

Doporučujeme zkonstruovat pomocnou funkci `invariant`, př.:

[32] Pozor na konflikt identifikátorů konstant `dt` s identifikátory variant derivací z příkazu `define-dt`.

```
(define-fun invariant ((dy Dt) (y Real)) Bool
    (and (or (= dy dy_run) (= dy dy_idle))
         (<= y ymax)
))
```

Příklad s použitím funkce `invariant`:

```
(assert (and (invariant dy_0 y_0)
             (invariant dy_1 y_1)
))
```

**Nastavení voleb variant derivací** se provádí pomocí příkazu `assert`, ve kterém se kombinuje libovolný stav systému a konstanty druhu `Dt`. Tím dochází k propojení diskrétní a spojité domény modelu.

Doporučujeme zkonstruovat pomocnou funkci `connect`, př.:

```
(define-fun connect ((dy Dt) (run Bool)) Bool
    (and (=>      run  (= dy dy_run ))
         (=> (not run) (= dy dy_idle))
))
```

Příklad s použitím funkce `connect`:

```
(assert (and (connect dy_0 run_0)
             (connect dy_1 run_1)
))
```

**Definice skoků.** Skoky, tj. změny diskrétního stavu, lze definovat též pomocí asercí mezi sousedními stavy a dalšími konstantami.

Doporučujeme zkonstruovat pomocnou funkci `jump`, př.:

```
(define-fun jump ((run1 Bool) (run2 Bool) (y2 Real)) Bool
    (and (=> (and      run1  (<  y2 bound_1) )      run2  )
         (=> (and      run1  (>= y2 bound_1) ) (not run2) )
         (=> (and (not run1) (>  y2 bound_2) ) (not run2) )
         (=> (and (not run1) (<= y2 bound_2) )      run2  )
))
```

Příklad s použitím funkce `jump`:

```
(assert (and (jump run_0 run_1 y_1)
             (jump run_1 run_2 y_2)
))
```

Pokud je požadavek na libovolnou změnu spojitého stavu modelu při některém skoku (např. reset časovače), je nutné tyto konstanty v jednotlivých fázích zdvojit, kde první značí např. hodnotu na začátku fáze a druhá na konci fáze. Příklad:

```
(declare-fun tau_0_0  () Real) (declare-fun tau_0_1  () Real)
(declare-fun tau_t_0  () Real) (declare-fun tau_t_1  () Real)
;; ...
(define-fun jump ( (run1 Bool)  (run2 Bool)
                   (tau1t Real) (tau20 Real)
                 ) Bool
    (and (=> (and      run1  (<  tau1t 5) )
             (and      run2  (=  tau20 tau1t) ))
         (=> (and      run1  (>= tau1t 5) )
             (and (not run2) (=  tau20 0) ))
         ;; ...
))
(assert (and (jump run_0 run_1 tau_t_0 tau_0_1)
             (jump run_1 run_2 tau_t_1 tau_0_2)
))
```

**Nastavení fází** znamená definovat časové okamžiky mezi integracemi, tj. např. hodnotami konstant `t_i`. Nejjednodušším způsobem je zavedení konstantní periody `T`, např.:

```
(define-fun T () Real 1)
(assert (and (= t_1 (+ t_0 T)) (= t_2 (+ t_1 T)) ))
```

**Integrace** se provádí příkazy `int-ode`. Dochází tím k propojení konkrétních vstupních a výstupních konstant druhu `Real` a konstant druhu `Dt`. Podle argumentů mezí nezávislé proměnné `t` těchto příkazů je určeno rozložení fází výpočtu. Všechna ODE ve stejných časových mezích jsou integrována synchronně.

Příklad:

```
(assert (and (= y_1 (int-ode y dy_0 (y_0 t_0 t_1) ()))
             (= y_2 (int-ode y dy_1 (y_1 t_1 t_2) ()))
))
```

a pro případ zdvojených konstant fází (viz. definice skoků):

```
(assert (and (= y_t_0 (int-ode y dy_0 (y_0_0 t_0 t_1) ()))
             (= y_t_1 (int-ode y dy_1 (y_0_1 t_1 t_2) ()))
))
```

#### 3.1.1.5 Předzpracování vstupu

Předzpracování vstupu znamená jeho úpravu na úrovni substitucí textu, před samotným zpracováním, bez sémantické analýzy. Základní funkcí předzpracování vstupu je odstranění komentářů.

Protože vstupy zpravidla obsahují velké množství opakujícího se kódu plynoucí z rozdělení výpočtu do fází, byla do možností předzpracování vstupního jazyka přidána *makra*, která umožňují parametrizované generování textového kódu. Princip se podobá makrům jazyka C. Rozlišují se *příkazová* a *uživatelská* makra. Příkazová makra slouží jako direktivy pro předzpracovač vstupu a lze pomocí nich zavést uživatelská makra. Uživatelská makra umožňují parametrizovanou textovou substituci.

Název každého makra musí být určen jediným tokenem, který začíná znakem `#`. Pokud je makro parametrizováno, musí být token následován výrazem s parametry. Pokud makro parametrizováno není, token může a nemusí být následován prázdným výrazem `()`, doporučujeme však prázdný výraz používat, čímž se zamezuje případné chybné interpretaci následujícího výrazu, který není (nemá být) seznamem parametrů. Parametry makra jsou v jeho těle použita jako dočasná uživatelská makra. Pokud název parametru koliduje s dříve definovaným uživatelským makrem, má parametr přednost.

Makra mohou obsahovat vnořená makra. Jsou-li makra expandována, vyhodnocení je provedeno rekurzivně a není kontrolováno, zda je rekurze konečná.

Makra umí pracovat s numerickými literály (včetně celočíselných), ale ne s literály `true` a `false` druhu `Bool`.

**Příkazová makra** se nesmí nacházet uvnitř vstupních výrazů, ale mohou se nacházet uvnitř jiných maker. Mezi tato makra patří:

- 1. `#if <cond> <body> #endif`
  2. `#if <cond> <body1> #else <body2> #endif`

  Podmíněně *expanduje* text `<body>`, pokud je *literál* `<cond>` vyhodnocen jako pravdivý. Ve variantě 2 je navíc při nesplnění podmínky `<cond>` expandována část `<body2>`. Výsledkem může být i prázdný text.

- 1. `#def <name> <|(<arg>*)> <body> #enddef`
  2. `#define <name> <|(<arg>*)> <body>`

  Zavádí *globální uživatelské makro* s názvem `<name>`, s parametry, nebo bez nich, s obsahem `<body>`. Je povolena nejvýše jedna definice globálního makra `<name>`. Makro `<name>` *není expandováno* v místě definice, ale až v místě volání. `<body>` tedy může obsahovat i libovolná vnořená makra včetně dalších definic; korektnost závisí až na kontextu místa volání makra. Makro smí být definováno i rekurzivně, ale je nutné ohlídat koncové podmínky.

  Ve variantě 1 může být tělo i víceřádkové; varianta 2 je zakončena koncem řádku.

- `#let <name> <<body>|(<body>)> <scope> <|#endlet <name>>`
  Zavádí *lokální uživatelské makro* v rámci `<scope>` s názvem `<name>`

bez parametrů. Smí být definováno i několik lokálních maker `<name>`, platné je to naposledy definované. `<body>` je expandováno už v místě definice, výsledkem expanze smí být i prázdný text. Neuzávorkované `<body>` je interpretováno jako jediný token. `#endlet` ukončuje platnost aktuálního makra.

- 1. `#for (<var> <init> <end>) <body> #endfor`
  2. `#for (<var> <init> (<cond>) (<step>)) <body> #endfor`
  3. `#for (<var> (<list>)) <body> #endfor`

  Expanduje text `<body>`, který může záviset na `<var>` jakožto lokálním uživatelském makru. Text je opakovaně expandován s měnící se hodnotou `<var>` závisle na uvedených podmínkách.

  V těle smí být obsažena vnořená makra `#for`.

  Varianta 1 generuje `<var>` s celočíselnými hodnotami od `<init>` do `<end>` včetně, s jednotkovým krokem. Varianta 2 generuje `<var>` s počáteční hodnotou `<init>` a následujícími hodnotami odpovídající vyhodnocení výrazu `(<step>)`, dokud je výraz `(<cond>)` vyhodnocován jako pravdivý. Varianta 3 generuje `<var>` postupně se všemi hodnotami uvedenými ve výčtu `(<list>)`.

Příklady příkazových maker:

```
#define TO_BOOL(cond) #if #cond true #else false

#define N() 5
#def SUM_SQR() (+
   #for (i 0 #N)  ;; equiv. to: #for (i 0 (<= #i #N) (+ #i 1))
                  ;; equiv. to: #for (i (0 1 2 3 4 5))
   #let j ((+ #i 1))
       (* #i #j)
   #endlet j
   #endfor
) #enddef
```

**Uživatelská makra** musejí být před použitím alespoň jednou definována pomocí příkazových maker `#def`, `#define` nebo `#let`, nebo jako parametry makra. Parametry maker jsou interně zavedeny pomocí mechanismu lokálních uživatelských maker (s tělem odpovídajícím hodnotě parametru v místě volání), a proto mezi nimi nebude nadále rozlišováno. Názvy lokálních a globálních maker se mohou navzájem překrývat, přednost má vždy naposledy definované lokální makro.

V každém místě volání jsou makra nahrazena za definovaný text, který může být závislý na parametrech nebo i na samotném makru — pak hovoříme o **rekurzi**. Rekurze mají velkou vyjadřovací schopnost, jelikož lze používat

vnořených podmíněných maker `#if`. Pomocí těchto rekurzí lze např. zavést makra `#for`. Průběh rekurzivních expanzí ale není nijak kontrolován.

Expanze maker je prováděna i uvnitř tokenů. Každý token je rozdělen na části podle znaků `#` a každá část je vyhodnocena zvlášť. Je-li token složen z více než jedné takové části, jsou všechny expandované části *složeny do jediného tokenu*, a to i v případě, že těla maker obsahují více než jeden token; nesmí však obsahovat výrazy. Vnitřní makra také nemohou mít žádné parametry, jelikož parametry maker nikdy nejsou součástí tokenu, protože se uvádí ve výrazech. Pouze poslední část tokenu smí obsahovat parametry umístěné v následujícím výrazu.

Nejen pro účely oddělení částí maker a textu v rámci jednoho tokenu jsou zavedena dvě rezervovaná makra s prázdným názvem a s názvem `#` (tj. volají se jako `#` a `##`). `#` je z textu smazáno[33]; `##` je expandováno na prázdný token. Obě makra lze použít na vynucení složení expanze makra do jednoho tokenu.

Znak `#` je možné použít jako escape sekvenci: `\#`, čímž se zamezí expanzi makra (nebo je odložena) a znak `#` je ponechán nedotčen. Toto je užitečné pro účely předávání tokenů, které obsahují lokální uživatelská makra, jako parametrů globálního uživatelského makra, pokud je žádoucí, aby bylo lokální makro expandováno až uvnitř těla na základě lokální definice. (Příkazová makra `#def`* svá těla neexpandují, proto v nich není potřeba escape sekvence používat.) Bez použití escape sekvencí není možné docílit toho, aby makro vygenerovalo znak `#`, tj. jakékoli neexpandované makro.

Doporučujeme používat tuto konvenci pro názvy uživatelských maker: velká písmena pro globální makra (`MACRO`) a malá písmena pro lokální makra (`macro`).

**Aritmetická expanze** je dalším nástrojem v rámci předzpracování vstupu, který slouží k nahrazení vstupního výrazu rezervované funkce za jeho aritmetické vyhodnocení. Výraz nesmí obsahovat nepřímé argumenty, jinak dojde k chybě při vyhodnocení. (Ve fázi předzpracování lze používat pouze literály a makra, ne konstanty a funkce.)

Expanze se provede předřazením tokenu `$` před výraz (bez `#`). (Vnořené výrazy už před sebou mít token `$` nemusí.) Výchozím typem argumentů vyhodnocovaných výrazů jsou reálná čísla (resp. čísla s plovoucí řádovou čárkou). Typ lze také určit explicitně přidáním znaku do tokenu s `$`:

- `<d | i>` → celočíselný typ,
- `f` → reálný typ.

[33] `#` může být použito pouze na konci tokenu, jinak je interpretováno jako makro s názvem, který následuje za `#`.

**Příklady obecného použití maker:**

```
#define STEPS() 10
#define STEPS-1() $d (- #STEPS 1)
#def INT_ODE(f)
#for (i 0 #STEPS-1)
#let j $d(+ #i 1)
    (= #f##_#j (int-ode #f d#f##_#i (#f##_#i t_#i t_#j) ()))
#endlet j
#endfor
#enddef
(assert (and
    #INT_ODE(x) ;; (= x_1 (int-ode x dx_0 (x_0 t_0 t_1) ()))
                ;; (= x_2 (int-ode x dx_1 (x_1 t_1 t_2) ())) ...
    #INT_ODE(y) ;; (= y_1 (int-ode y dy_0 (y_0 t_0 t_1) ())) ...
))

#def FACT(n)
#if $(= #n 0)  *
#else          #FACT( $d(- #n 1) ) #n
#endif
#enddef
(assert (= $(#FACT(5)) 120) ) ;; $(* 1 2 3 4 5) == 120  => true
#define SEQ() 1 2 3
( #SEQ )  ;; (1 2 3)
( #SEQ# ) ;; (123)
( #FACT#(3) )  ;; error: missing parameters for 'FACT' !
( #FACT(3)# )  ;; (* 1 2 3)
( ##FACT(3) )  ;; (FACT(3))
( ###FACT(3) ) ;; (*123)
```

**Použití maker ve vstupním jazyce:**

- Definice numerických literálů, které lze použít i v příkazech `define-dt`:

```
#define K() 5
(define-dt x dx () (* #K t))
```

- Nastavení fází výpočtu:

```
#define T()     0.5
#define T_MIN()   0
#define T_MAX()  10
#define STEPS() $d(+ $f(/ (- #T_MAX #T_MIN) #T) )
;; ...
#def T_STEPS()
#for (j 1 #STEPS)
#let i $d(- #j 1)
    (= t_#j  (+ t_#i  #T))
#endlet j
#endfor
#enddef
```

- Deklarace všech konstant fází:

```
#def DECL_CONSTS(const type)
#for (i 0 #STEPS)
    (declare-fun #const##_#i () #type)
#endfor
#enddef
#DECL_CONSTS(t  Real)
;; ...
```

  Stejný způsob lze použít pro invarianty, volby variant derivací, skoky a integrace.

- Aproximace intervalových počátečních podmínek:

```
#def INIT_INTERVAL(var min max step) (or
#for (i #min (<= #i #max) (+ #i #step))
   (= #var #i)
#endfor
) #enddef
```

- Kaskádní kompozice systémů: každý systém reprezentovat vlastní sadou konstant s nějakou číselnou příponou a vstup generovat pomocí maker `#for` nebo rekurzivních maker. Krajní systémy generovat zvlášť, nebo pomocí makra `#if`, a ostatní propojit pomocí parametrických uživatelských maker, např. s parametrem `#i` a `#let j $d(+ #i 1)`.

### 3.1.2 Výstupy

Výstupem řešiče je především příznak splnitelnosti. Je-li vstup splnitelný, pak volitelně také ohodnocení diferencovaných konstant v jednotlivých fázích. Pokud je navíc zadán výstupní soubor, jsou do něj zapsány celé trajektorie všech diferencovaných funkcí a z těchto dat je vykreslen společný graf.

V případě konečné množiny počátečních podmínek je výstupem `sat` nebo `unsat` v případě splnitelného nebo nesplnitelného vstupu. S výjimkou zanedbání aproximačních chyb ODE řešiče jsou tyto výstupy exaktní, jelikož řešič buď nalezne splňující ohodnocení, nebo prozkoumá všechny možnosti a ověří, že žádná není splnitelná.

V případě nekonečné množiny počátečních podmínek je možný výstup `sat`, ale pokud je vstup nesplnitelný, výpočet pravděpodobně nikdy neskončí.

Výstup `unknown` *není* navržen, ač by byl v některých případech vhodný, např. pro nesplnitelné intervalové počáteční podmínky.

## 3.2 Softwarová architektura

V následující sekci popíši abstraktní návrh modelu komponent celého řešiče, jejich vztahů a rozhraní a rozdělení zodpovědností. Poté rozeberu interní návrh jednotlivých komponent.

### 3.2.1 Model komponent

Stěžejními komponentami jsou SMT a ODE řešič. Úkolem je zajistit jejich vzájemnou komunikaci a řídit centrální algoritmus celého procesu od přijmutí vstupu ve vstupním jazyce po výpis výsledků.

Průběh výpočtu se řídí zejména SMT řešičem. První otázkou bylo, zda celý systém řídit z SMT řešiče, nebo jej použít jako externí komponentu a centrální bod umístit mimo něj. Pokud by byl SMT řešič centrálním bodem, mohly by být maximálně využity všechny jeho funkce. Nevýhodou by však byla nutnost zvolit konkrétní řešič a komunikaci s ODE řešičem implementovat uvnitř, což by vyžadovalo podrobnější obeznámení se s fungováním SMT řešiče. Je však tento postup nutný?

Pokud by byl SMT řešič implementován jako černá skříňka, bylo by použití celého systému mnohem flexibilnější. SMT řešiče jsou stále v aktivním vývoji a možnost použít libovolnou verzi by bylo velkým přínosem. Proto bylo snahou nalézt řešení využívající tohoto postupu.

#### 3.2.1.1 SMT řešič jako nezávislá komponenta

SMT řešič lze použít jako téměř nezávislou komponentu, pokud splňuje tyto požadavky:

- je inkrementální,

- je konformní s SMT-LIB standardem.

Každý takový řešič pak lze používat a komunikovat s ním výhradně prostřednictvím SMT-LIB jazyka, tj. přes soubory nebo standardní vstup a výstup.

Vzhledem k tomu, že SMT-LIB obsahuje operace se zásobníkem asercí (viz. sekce 2.2.1), je možné dynamicky přidávat a odebírat klauzule, což lze použít pro přijímání hodnot z ODE řešiče. Druhá možnost je přidávání podmíněných a konfliktních klauzulí s vypočtenými hodnotami z ODE řešiče. Tyto postupy jsou možné z toho důvodu, že ODE řešič v našem pojetí pracuje výhradně s pevnými počátečními podmínkami a má tedy i jednoznačný výstup. Hodnoty konstant jsou získávány postupně od počátečních podmínek a přidávány jako nové aserce. Při porušení některých invariant se provede návrat. Podrobnější algoritmus je popsán v centrální komponentě.

Svým způsobem lze tento postup použít i jako opakovaně generované celistvé statické vstupy pro neinkrementální SMT řešič, ale předpokládá se, že inkrementální řešič si bude počínat efektivněji.

Tímto je získáno velké flexibility ze strany SMT řešiče, vzhledem k tomu, že SMT-LIB standard podporuje většina řešičů. Jediné potenciální riziko je neefektivní počínání řešičů v inkrementálním módu, tj. pokud by obecně operace ověření splnitelnosti byla výpočetně náročná, vzhledem k tomu, že tato operace bude prováděna často. Předpokládá se však, že doba výpočtu by měla být výrazně nižší v následujících fázích, které přidávají jen malé množství nových asercí, oproti první fázi, který řeší celý počáteční vstup.

Používání textového rozhraní pomocí SMT-LIB by mělo mít zanedbatelný vliv na výkon oproti použití programového rozhraní, v porovnání s dobou samotných výpočtů SMT a ODE řešičů. Implementace si však musí poradit s korektními konverzemi čísel s plovoucí řádovou čárkou z textu či do textu, protože SMT řešič pracuje s exaktními hodnotami.

#### 3.2.1.2 ODE řešič

ODE řešič postačuje použít jako samostatnou komponentu, jelikož má fungovat jako filtr — pro každý vstup vrátí odpovídající výstup. Výjimkou je jen jeho inicializace, kdy se musí nastavit tvary diferenciálních rovnic. Konkrétní rozhraní nehraje důležitou roli, jen je důležité dávat pozor na nastavení přesnosti čísel s pohyblivou řádovou čárkou, protože SMT řešič pracuje s racionálními čísly, které jsou exaktní. To může činit potíže zejména při komunikaci prostřednictvím znakových řetězců.

Důležitým požadavkem je však to, aby byl řešič schopen přijímat specifikace diferenciálních rovnic dynamicky jako text, protože tak jsou reprezentovány ve vstupním jazyce. Např. řešiče odeint a SUNDIALS přijímají specifikace jako kompilované funkce přímo v programovacím jazyce, což je efektivní, ale pro tento účel nevhodná varianta. Řešiče, které to umějí, existují (např. GNU Plotutils, GNU Octave, SageMath; nemluvě o komerčních nástrojích), ale problém je např. v tom, že (pochopitelně) nemají navzájem nijak standardizován

vstupní formát, jako tomu je třeba u SMT řešičů s SMT-LIB standardem. Pokud bychom zvolili některý z nich, mohlo by být následně poměrně obtížné umožnit nasazení jiného řešiče.

Komponentu s ODE řešičem by bylo možné navrhnout tak, aby nějakým způsobem obalovala obecné funkcionality řešiče bez ohledu na konkrétní použitý nástroj. Toho lze docílit navržením komponenty jako abstraktní třídy obstarávající veřejné rozhraní a konkrétní implementaci přenechat na odvozených třídách. Pokud by byl zvolený ODE řešič implementován v jazyce C++ nebo C, měla by být jeho implementace uvnitř odvozené třídy snadná, a navíc by takové řešení mělo být efektivní.

Po řešiči je (prozatím) vyžadována pouze integrace na základě exaktních počátečních podmínek a koncových podmínek určených předem danou délkou integrace, bez řešení jakýchkoli invariant.

#### 3.2.1.3 Zpracování vstupu

Vstupní jazyk je navržen podobně jako jazyk SMT-LIB standardu, což značně usnadňuje zpracování vstupu, jelikož stačí zpracovat jen přidané ODE konstrukty a celý zbytek vstupu delegovat s jen minimálními změnami na SMT řešič jako inicializaci. ODE řešič je nutné inicializovat definicemi všech diferenciálních rovnic a jejich argumentů.

Přidanou hodnotou je umožnění použití maker, pomocí nichž lze vstupy parametrizovat a ke generování není zapotřebí dalšího nástroje. Předzpracování vstupu, tak jak je navrženo, je zcela nezávislé na sémantice vstupního jazyka a mělo by být implementováno jako samostatná komponenta.

Komponenta zpracování vstupu bude s konstantami a funkcemi pracovat výhradně na úrovni jejich identifikátorů a nebude řešit jejich možné hodnoty; to bude zodpovědnost centrální komponenty a SMT řešiče.

#### 3.2.1.4 Centrální komponenta

Zodpovědností centrálního bodu je na základě zpracovaného vstupu nastavit diferenciální rovnice a ty související sjednotit, inicializovat oba řešiče a určit fáze výpočtu. Následně pak řídit průběh výpočtu a komunikaci mezi oběma řešiči.

Díky několika možným rozhraním u obou řešičů je návrh poměrně volný a důraz je kladen hlavně na zvolený řídící algoritmus, který musí korektně ověřit všechny možnosti ohodnocení vstupních konstant a funkcí a přiměřeně efektivně zacházet s inkrementálním SMT řešičem, zejména s návraty. Návrh algoritmu je uveden v samostatné sekci.

### 3.2.2 Návrh ODE řešiče

Problém s dynamickými textovými specifikacemi diferenciálních rovnic a sestavení odpovídajících funkcí jsem se rozhodl řešit formou stromových struktur

výrazů a jejich transformací na funkce s argumenty. Tyto struktury pak lze použít jako vstupní specifikace při inicializaci a vytvořené funkce volat ve fázích integrace.

Nejprve popíši návrh avizované struktury a poté návrh abstraktního řešiče.

#### 3.2.2.1 Výrazy a jejich vyhodnocení

Tato datová struktura sestává z *výrazů* a z jejich i několika *vyhodnocení*.

**Výraz** je obecná stromová struktura sestavená z prefixových textových výrazů. Každý výraz obsahuje spojový *seznam* potomků, z nichž každý je buď další podvýraz, nebo token s textovou hodnotou. Seznam je použit proto, že se předpokládá sekvenční průchod jeho strukturou, a aby bylo možné efektivně odkudkoli odebírat či přidávat prvky.

Výraz ve výchozí formě nemá určen žádný datový typ a používá pouze znaky. Může být tedy použit pro libovolné účely vyžadující vytvoření hierarchické struktury z (ne zcela nutně) prefixového vstupu, např. i pro účely syntaktického rozboru textového vstupu, který ani není výrazem, ale používá prefixovou notaci.

Po této struktuře je vyžadováno, aby co nejvíce zpřístupnila sekvenční čtení i zápis, což budou velmi časté operace.

**Vyhodnocení** se vždy vztahuje k jedinému výrazu, ale výraz může mít přidružených i několik vyhodnocení. Úloha vyhodnocení je vytvořit z obecné textové struktury výrazu strom konkrétních funkcí s přímými či nepřímými argumenty konkrétního aritmetického typu, který lze v inicializaci volit různě. Tato struktura musí umožňovat volání jako funkce, případně i s parametry, pokud výraz obsahuje nepřímé argumenty.

Když je výraz transformován na vyhodnocení, musí být jeho první prvek token s názvem nějaké funkce, typicky aritmetickým operátorem. Následující prvky nemají žádná omezení (kromě toho zmíněného pro podvýrazy) a platí pro ně následující:

- je-li prvek další podvýraz, vytváří se další vyhodnocení,
- je-li prvek token, provede se konverze na datový typ; pokud konverze selže (token nereprezentuje hodnotu daného typu), je token považován za nepřímý argument.

Nepřímé argumenty se později dosadí jako parametry při volání vyhodnocení a textové hodnoty tokenů jsou uloženy jako klíče argumentů. Nepřímých argumentů se stejným klíčem může být ve výrazu obsaženo více.

#### 3.2.2.2 Abstraktní řešič

Řešič bude implementován abstraktní třídou tak, aby umožňoval snadné odvození na konkrétní ODE řešič s implementací různých metod integrace

funkcí. Odvozené třídy by měly řešit pouze implementaci konkrétních metod, ale veřejné i neveřejné rozhraní by měla řešit abstraktní třída, včetně stanovení použitých datových struktur.

Řešič bude umožňovat inicializaci specifikací diferenciálních rovnic primárně pomocí datových struktur výrazů uvedených v sekci 3.2.2.1. K těmto výrazům si řešič interně sestaví odpovídající vyhodnocení tak, jak jsou uvedeny v sekci 3.2.2.1. Uživatel bude pracovat pouze s výrazy, od vyhodnocení bude odstíněn. Naopak implementace odvozených řešičů budou pracovat pouze se sestavenými vyhodnoceními. Vyhodnocení budou vždy obsahovat i nepřímé argumenty, implicitně alespoň argument integrované funkce a volitelně také argument nezávislé proměnné `t` (viz. vztah (1.2)).

Specifikované diferenciální rovnice bude možné počítat opakovaně s různými vstupními argumenty podle konstant aktuálních příkazů `int-ode`. Při výpočtech budou v implementaci interně volána sestavená vyhodnocení v každém kroce integrace, což vyžaduje, aby bylo volání vyhodnocení přiměřeně efektivní, jelikož kroků integrace bude řádově stovky až statisíce (podle délky fází výpočtu).

Třída by se však měla pokud možno chovat jako obecný ODE řešič bez užších vazeb na problém SMT. Tuto zodpovědnost by měla řešit centrální komponenta.

Řešič bude také podporovat ukládání průběhu integrací všech ODE a jejich výpis.

Takový řešič bude možné používat jako filtr (s výjimkou inicializace) — na každý vstup odpoví výstupními hodnotami.

### 3.2.3 Návrh zpracování vstupu

Úkolem komponenty pro zpracování vstupu je nalézt příkazy ODE vstupního jazyka (viz. 3.1.1.3) a částečně také SMT konstrukty (viz. 3.1.1.2), zpracovat je a nahradit je za konstrukce výhradně SMT-LIB standardu, nebo je zcela vyřadit. Postup je následující:

1. Nastavení SMT vstupu: zvolení logiky, definice druhu `Dt`, ad.
2. Zpracování definic diferenciálních rovnic: načtení definic diferenciálních rovnic do výrazů a načtení seznamů klíčů nepřímých argumentů pro ODE řešič z příkazů `define-dt`; definice konstant variant derivací druhu `Dt` s názvy podle identifikátorů z `define-dt`.
3. Substituce příkazů integrací `int-ode` za pomocné konstanty nebo funkce druhu `Real`; sekvenční uložení identifikátorů argumentů příkazů `int-ode` a roztřízení podle jednotlivých ODE a podle unikátních párů identifikátorů nezávislých proměnných `t`. (Nastavení fází je zodpovědnost centrální komponenty.)
4. (Volitelné) nastavení počáteční délky kroku integrací z příkazu `define-ode-step` a jeho smazání.
5. Transformace zbylých konstruktů nekompatibilních s SMT-LIB standardem, které jsou povoleny ve vstupním jazyce (např. záporné numerické

literály).

Ke zpracování vstupu lze s výhodou použít struktury výrazů ze sekce 3.2.2.1, jelikož vstupní jazyk používá prefixovou notaci.

### 3.2.4 Návrh předzpracování vstupu

Tato komponenta má fungovat samostatně pro libovolný prefixový vstup, tak jak je definována v sekci 3.1.1, ale bez ohledu na sémantiku tokenů a výrazů, s výjimkou komentářů a maker. Návrh maker je uveden v podsekci 3.1.1.5.

V první řadě se provedou nejjednodušší substituce textu na úrovni řádků, bez ohledu na strukturu výrazů i maker. V této fázi dojde ke smazání komentářů a k nahrazení řádkových maker `#define` za jejich uzavřený ekvivalent ve tvaru makra `#def`. Poté je vstup nezávislý na řádcích a je závislý výhradně na struktuře výrazů a maker.

Následné zpracování textu bude (opět) založeno na třídách výrazů ze sekce 3.2.2.1. Výrazy a makra budou procházeny rekurzivně a každý token obsahující makro bude náležitě zpracován, což bude vyžadovat operace vkládání a odstraňování potomků výrazů.

### 3.2.5 Řídící algoritmus

Úlohou algoritmu je dospět v SMT řešiči k ohodnocení všech konstant na základě výsledků rovnic z ODE řešiče. Hlavní výzvou je problém s návraty, kdy v průběhu výpočtu dochází k tomu, že vstup v aktuální podobě není splnitelný. Návraty má efektivně implementován SMT řešič, ale je otázkou, jakým způsobem je na něj delegovat.

Složitost roste s počtem fází výpočtu a s počtem všech možných voleb derivací. V nejhorším případě se musí projít všechny možnosti, tomu se nelze vyhnout, kromě použití nějakých heuristik, které ale nebudou uvažovány. Rovněž nebudou uvažovány možnosti paralelizace.

**Diskuze redukce ověření splnitelností.** Důležitým aspektem je poměr výpočetních náročností operací ověření splnitelnosti SMT řešičem a výpočtu diferenciálních rovnic ODE řešičem. Vzhledem k tomu, že celý výpočet je rozdělen do mnoha relativně málo vzdálených fází, je délka integrací poměrně malá oproti běžným případům užití. Navíc integrace počítá jen s malým množstvím vstupních hodnot, oproti SMT řešiči, který musí v každé fázi ověřit splnitelnost kompletně celého vstupu, ač v inkrementálním módu. Dá se tedy očekávat, že ODE řešič bude rychlejší než SMT řešič, a efektivní algoritmus by měl redukovat počet operací ověření splnitelnosti a částečně do nich delegovat návraty.

Návraty na SMT řešič bohužel není v rozumné míře možné delegovat zcela, protože takový postup by vyžadoval spočítat úplně všechny možnosti průchodů. Důvod je ten, že každá dílčí integrace závisí na konkrétních vstupních

hodnotách, a tyto zase tranzitivně závisí na všech předešlých. Tudíž není možné mít v každé fázi pokryty všechny možnosti, např. pomocí podmíněných klauzulí, aniž by složitost rostla exponenciálně.

Částečná redukce ověření splnitelnosti je možná pomocí způsobu, kdy se v každé fázi vyřeší kromě SMT řešičem zvolené varianty derivací navíc také všechny ostatní kombinace voleb variant v rámci aktuální fáze a přidají se jako podmíněné klauzule. Tím by se pokrylo lokální okolí aktuální fáze a počet nutných ověření splnitelnosti by se redukovalo o jednu úroveň stromu prohledávaného prostoru, a pomocí podmíněných klauzulí by se částí návratů zabýval SMT řešič.

Počet všech kombinací variant derivací závisí na produktu počtu variant derivací každé ODE, kterých je omezený počet a nezávisí na velikosti vstupu (počtu fází), ale výhradně na obecné specifikaci modelu. Tento počet by tedy neměl být velký a pokud by byl ODE řešič výrazně rychlejší než SMT řešič, měla by redukce počtu ověření splnitelnosti převážit nad nadbytečným výpočtem diferenciálních rovnic. Takový algoritmus by si však v případě návratů musel nějakým způsobem pamatovat, které vypočtené varianty už procházel a které ještě ne.

Postup řešením více variant derivací lze dále modifikovat — řešit jich více či méně, o více fází napřed, apod. Efektivita zvoleného řešení by závisela na empirickém měření složitosti, analyticky ji lze těžko předpovědět.

**Základní algoritmus.** Pro účely prototypu navrhnu zatím alespoň základní algoritmus, který postupuje jen po jednotlivých cestách ve stromu prohledávaného prostoru, výhradně na základě aktuálního ohodnocení konstant, bez předběžných výpočtů jiných variant. Takový postup je jednodušší implementovat, ale v každé fázi vyžaduje ověření splnitelnosti, a tedy významně závisí na její výkonnosti.

V této variantě ztrácí smysl přidávat nové hodnoty jako podmíněné klauzule, namísto toho stačí aserce jen vkládat do zásobníku asercí. Důvod, proč samotné podmíněné klauzule nefungují, je ten, že nově vypočtené hodnoty jsou podmíněny jejich vstupními podmínkami, ale SMT řešič nic nenutí tyto vstupní podmínky zvolit a smí si zvolit i jiné varianty derivací, které ale ještě nejsou spočtené, a tudíž si za výsledek integrace smí dosadit libovolnou hodnotu. Tudíž by bylo nutné kromě podmíněné klauzule navíc explicitně přidat klauzule, které vyžadují vstupní hodnoty v předpokladech. Tím ale podmíněné klauzule ztrácí smysl a stačí jen rovnou přidat předpoklady i výsledky do asercí bez podmínek.

Bez použití podmíněných klauzulí je však nutné při návratu přidané aserce odebrat, což umožňují operace se zásobníkem asercí. Je nutné přidávat konfliktní klauzule, tím se definitivně uzavírají větve ve stromu prohledávaného prostoru a algoritmus tak konverguje k výsledku. Se vzrůstajícím množstvím konfliktních klauzulí však roste složitost dílčích operací ověření splnitelnosti.

Postup je následující:

1. Překlad vstupní formule:

i. Uložení definic diferenciálních rovnic ze zpracovaného vstupu, určení rozložení fází výpočtu[34].

ii. Sloučení všech ODE, které obsahují společné klíče nepřímých argumentů, do soustav ODE.

iii. Inicializace SMT řešiče, tj. zaslání modifikovaného vstupu bez specifikací diferenciálních rovnic.

iv. Inicializace ODE řešiče, tj. zaslání specifikací (soustav) diferenciálních rovnic.

2. Nastav počáteční číslo fáze na 0: `s := 0`.
3. Ověření splnitelnosti SMT formule:
   - Je-li splnitelná, získej model, tj. ohodnocení všech konstant.
   - Není-li splnitelná, proveď návrat:
     i. Pokud `s = 0`, vstup není splnitelný. Konec.
     ii. Odeber vrchní úroveň zásobníku asercí: `(pop 1)`.
     iii. Přidej konfliktní klauzuli (aserci) znemožňující vstupní (nikoli výstupní[35]) ohodnocení předchozí fáze.
     iv. Vrať se do předchozí fáze: `s--`, jdi na bod 3.
4. Pokud je dosaženo celkového počtu fází, jdi na bod 9.
5. Vyber ohodnocené konstanty, které do fáze vstupují jako vstupní argumenty.
6. Proveď výpočet všech diferenciálních rovnic v rámci aktuální fáze a ulož výstupy.
7. Přidej výstupní i vstupní hodnoty této fáze jako aserce do nové úrovně zásobníku asercí: `(push 1)`.
8. Přejdi do další fáze: `s++`, jdi na bod 3.
9. Vypiš získaný model. Konec.

Tento postup se díky konfliktním klauzulím podobá algoritmu DPLL.

[34] Tato operace není diskutována, nicméně ve zcela obecném případě se může jednat o poměrně náročnou úlohu.

[35] Přidání výstupních hodnot do konfliktní klauzule by umožnilo řešiči za ně dosadit jiné hodnoty a tím zneplatnit celou klauzuli.

Kapitola 4

# Realizace

V implementaci řešiče jsem postupoval po jednotlivých softwarových komponentách podle jejich návrhu. V některých případech, zejména u centrální komponenty, je realizace oproti návrhu zjednodušena, což je v odpovídající sekci explicitně zmíněno. Výsledný prototyp lze použít pro účely experimentování s různými modely hybridních systémů a pro účely srovnání s řešičem pracujícím s intervalovou metrikou, konkrétně dReal (viz. sekce 2.4).

Nejprve popíši projekt jako celek, poté rozeberu implementaci jednotlivých komponent, centrální komponentou konče. Na závěr uvedu výčet některých nedostatků a dosud chybějících funkcionalit jako seznam úkolů do budoucna.

## 4.1 Struktura a vlastnosti projektu

Projekt jsem nazval *SMT+ODE solver (SOS)*. Řešič je koncipován jako soubor knihoven, appletů a hlavní aplikace, implementovaných převážně v jazyce C++ (není-li řečeno jinak). Přestože se jedná o prototyp, je projekt strukturně koncipován tak, aby byla jeho případná rozšíření a další vývoj možná provést snadno přímo v něm. Projekt je zamýšlen jako soubor knihoven umožňující použití různých SMT a ODE řešičů jak jako samostatných aplikací, tak jako C++ knihoven. Projekt používá verzovací systém *git*, má otevřené zdrojové kódy a je veřejně dostupný včetně tohoto textu práce na adrese `https://github.com/Tomaqa/sos`.

Zdrojové kódy napsané v C++ používají standard C++14 (nejsou zpětně kompatibilní se staršími standardy) a jsou umístěny ve jmenném prostoru `SOS`. Moduly, které zprostředkovávají některý z řešičů, jsou izolovány od dalších zodpovědností. Tyto moduly jsou pochopitelně závislé na knihovnách třetích stran. Kromě těchto (vyměnitelných) modulů je však celý projekt se základními funkcemi nezávislý od externích knihoven a využívá výhradně vlastní a standardní knihovny STL a POSIX. C++ zdrojové kódy jsou zapsány

ve stylu Stroustrup[36].

Hlavní aplikace je umístěna v souboru `bin/sos_odeint`. Je implementována v jazyce C++ a podrobněji rozebrána ve vlastní sekci. Původně byla vyvíjena ve skriptovacím jazyce Bash v souboru `bin/prototype.sh`, který již v aktuální verzi není přítomen, ale je zpětně dohledatelný.

Před použitím nástroje je nutné jej sestavit. Návod je umístěn v příloze */* link *\/*. Všechny uvedené soubory aplikací či appletů jsou přístupné až po tomto sestavení.

Nástroj byl vyvíjen a testován pouze v OS Linux.

## 4.2 Realizace výrazů a jejich vyhodnocení

Vzhledem k tomu, že tyto struktury využívá většina komponent projektu, jsou implementovány jako samostatná knihovna a uvedeny ve vlastní sekci.

Výrazy reprezentuje třída `Expr` a jejich vyhodnocení šablonová třída `Expr::Eval<Arg>`.

**`Expr`** je odvozená třída od abstraktní třídy `Expr_place`. `Expr` využívá polymorfismu — obsahuje stromovou strukturu jako seznam ukazatelů na `Expr_place`, které představují argumenty výrazu. Argumenty jsou objekty tříd odvozených od `Expr_place`: buď uzly jako další podvýrazy (`Expr`), nebo koncové listy, které reprezentují nějakou hodnotu. Ty jsou implementovány šablonovou třídou `Expr_value<Arg>`, která může jako šablonový parametr obsahovat i netextový typ (což je vlastně v rozporu s návrhem). Jako její speciální odvozená třída je zavedena třída `Expr_token`, která obsahuje textovou hodnotu. Pokud uživatel nepoužívá objekty `Expr_value<Arg>`, pak je struktura nezávislá od interpretace.

`Expr_token` umožňuje šablonovou interpretaci svého textového obsahu jako aritmetického typu pomocí `get_value`, a také nastavení podle aritmetické hodnoty pomocí `set_value`. Tyto konverze však mohou být nepřesné, např. v případě čísel s plovoucí řádovou čárkou. V takovém případě může být vhodnější použít `Expr_value<Arg>`.

Zjištění typu ukazatele je doporučeno provádět pomocí funkcí `is_evalue`, `is_etoken` a `is_expr`. Přetypování ukazatelů na nekterý odvozený typ je rovněž doporučeno provádět pomocí explicitních funkcí (`ptr_to_expr`, apod.).

`Expr` se typicky sestavuje z textového vstupu: `std::string` nebo `std::istream`. Jedinými speciálními znaky textového vstupu jsou znaky kulatých závorek `(` a `)`, které interpretují argument jako (pod)výraz. Všechny ostatní znaky jsou interpretovány jako tokeny oddělené bílými znaky, jež nikdy nejsou součástí tokenů. Pro listy jsou použity pouze objekty textového typu — `Expr_token`.

[36]Podle tvůrce jazyka C++: Bjarne Stroustrup.

Každý objekt třídy obsahuje interní iterátor indikující pozici v seznamu potomků a s ním související funkce umožňující sekvenční čtení i zápis, případně spojené i s přetypováním, např. `peek`, `get`, `extract`, `get_token`, `extract_expr`, `add_new_expr_at_pos`, `erase_at_pos`, ...

Dále jsou vřazeny šablonové funkce, které provedou konverzi potomka libovolného typu na aritmetický typ (`ptr_to_value`, `get_value` apod.).

Nad sestavenými výrazy lze provést některé základní operace:

- `simplify` — všechny (pod)výrazy (včetně kořenového), které obsahují jen jediný argument typu token, jsou převedeny na token.
- `to_binary` — výraz je transformován tak, aby každý (pod)výraz (včetně kořenového) obsahoval nejvýše tři argumenty, z nichž první musí být token s libovolným názvem funkce bez ohledu na její interpretaci.
- `flatten` — všechny vnořené tokeny jsou přesunuty do kořenového výrazu a podvýrazy jsou smazány.
- `transform_to_args<Arg>` — výraz, který obsahuje výhradně tokeny, je transformován na pole prvků typu `Arg`.
- `get_eval<Arg>` — provede se `to_binary` a vrátí se objekt typu `Expr::Eval<Arg>` sestavený z položek výrazu.

Vyhodnocení výrazu je vždy externím objektem, není obsaženo jako členská proměnná. Dokud není použita operace `get_eval`, je objekt zcela nezávislý na svém vyhodnocení a implementace třídy `Expr::Eval<Arg>` ani nemusí být přítomna. Třídu `Expr` je tedy možné použít i pro libovolné účely vytvoření hierarchické struktury z prefixového textového vstupu bez jakékoli spojitosti s aritmetickým vyhodnocením.

**`Expr::Eval<Arg>`.** Nové objekty vyhodnocení se konstruují z objektů třídy `Expr`, které musí být v binárním či unárním tvaru, či v jejich kombinaci. Vytvoření objektu vyhodnocení s přímými a nepřímými argumenty se děje podle návrhu uvedeném v sekci 3.2.2.1.

Klíče nepřímých argumentů jsou ukládány dynamicky bez duplikací v pořadí prefixového průchodu výrazem. Objekty třídy `Expr::Eval<Arg>` mají přetížen operátor volání funkce, tj. `()`, s pozičními parametry s hodnotami pro nepřímé argumenty v pořadí, v jakém byly uloženy jejich klíče.

Aby bylo pořadí parametrů vyhodnocení jednoznačné, je možné mu je explicitně přiřadit při konstrukci. Pokud výraz obsahuje další klíče, které dosud nejsou obsaženy, jsou umístěny na konec seznamu klíčů. Mohou být obsaženy i redundantní klíče, které ve výrazu obsaženy nejsou, ale hodnota jim při volání přiřazena být musí (ač libovolná).

V unárních a binárních funkcích jsou přítomny všechny rezervované funkce specifikované ve vstupním jazyce v sekcích 3.1.1.2 a 3.1.1.3.

`Expr::Eval<Arg>` obsahuje stromovou strukturu objektů třídy `Eval::Oper` reprezentující hierarchii vyhodnocení výrazů `Expr`, dále pole klíčů nepřímých

argumentů a pole jejich hodnot. Pole hodnot je nastaveno při volání celého vyhodnocení jako funkce.

`Eval::Oper` představuje binární nebo unární funkci s argumenty tří možných typů:

- přímý argument — hodnota,
- nepřímý argument — ukazatel do pole hodnot klíčů,
- podvýraz — ukazatel na další objekt typu `Eval::Oper`.

Argumenty je nutné vyhodnotit až v momentě volání funkce, proto je použit princip *líného vyhodnocení* (angl. *lazy evaluation*) — argumenty jsou uloženy jako nulární funkce, které jsou volány společně s voláním vyhodnocení objektu `Eval::Oper`. Celý výraz `Expr::Eval<Arg>` je pak vyhodnocen voláním kořenového objektu `Eval::Oper`.

Obě třídy jsou dostupné jako knihovny. Také je možné je využít pomocí aplikace `bin/applet/eval`, která vyhodnocuje vstupní výrazy, které mohou obsahovat i nepřímé argumenty.

## 4.3 Implementace adaptéru SMT řešiče

SMT řešič je vhodné mít implementován flexibilně tak, aby byla snadná jeho výměna, jak bylo diskutováno v sekci 3.2.1.1. V našem prototypu je řešič použit jako samostatná aplikace s textovým rozhraním podle SMT-LIB standardu verze 2. Náš nástroj byl testován s řešiči CVC4 a z3 (viz. sekce 2.2.2 */* a z3 */*).

SMT řešič reprezentuje třída `SMT::Solver`, která momentálně zahrnuje jak potřebné rozhraní, tak implementaci související s propojením textového rozhraní se synovským procesem SMT řešiče. Výhledově by bylo vhodné rozhraní a implementaci oddělit, tj. třídu realizovat jako abstraktní, která poskytuje operace obecně potřebné k řešení hybridních modelů bez ohledu na konkrétní implementaci. K této třídě by byla poskytnuta jako základní implementace odvozená třída s názvem např. `SMT::Smtlib`, která by operace delegovala přes textové rozhraní. Bylo by však možné zvolit libovolnou jinou třídu, která by např. operovala přímo s programovým rozhraním konkrétního SMT řešiče jako knihovny[37].

Komponenta zahrnuje i část zodpovědností, které nesouvisí výhradně jen s SMT řešičem, ale jsou částečně spjaty s kombinováním řešiče s diferenciálními rovnicemi. Úlohou komponenty je zprostředkování SMT řešiče pro účely tohoto nástroje, ne implementace nezávislého řešiče.

[37]Oba zmíněné SMT řešiče jsou implementovány v jazyce C++. Pokud by byly vyšší požadavky na výkon nástroje, bylo by možné je použít jako externí knihovnu s C++ rozhraním.

**Identifikátory a hodnoty** vstupních konstant jsou uloženy podle příkazů `int-ode` po řádcích reprezentovaných strukturou `Const_ids_rows`, resp. `Const_values_rows` (pole struktur `Const_ids_row`, resp. `Const_values_row`) pro každou ODE zvlášť. Klíčem každého řádku identifikátorů je unikátní dvojice konstant počáteční a koncové hodnoty nezávislého parametru `t` (`Time_const_ids`) a hodnotou je `Const_ids_entries` (pole struktur `Const_ids_entry`), tj. pole identifikátorů voleb derivací, počátečních hodnot a vstupních parametrů každého jednotlivého systému, který používá danou ODE v daném okamžiku. Použití pole umožňuje, aby definovanou ODE mohlo současně používat více systémů nezávisle, např. v případě kaskádní kompozice. Struktury hodnot konstant jsou analogické.

`SMT::Solver` komunikuje s SMT řešičem pomocí operací definovaných SMT-LIB standardem (viz. sekce 2.2.1), nicméně jedná se o obecný koncept operací použitelný pro různé implementace. Každá uvedená operace je později rozvedena včetně konkrétních použitých funkcí. Realizace komunikace s SMT řešičem je uvedena až v další části.

Kromě inicializace se jedná o tyto operace:

- `(check_sat)` — ověření splnitelnosti aktuálních asercí; výstupem je `sat` nebo `unsat` (nebo `unknown`, což je považováno za chybu),
- `(get_value)` — získání hodnot konkrétních konstant, čemuž musí předcházet `check_sat`; výstupem jsou exaktní racionální čísla zpravidla ve tvaru zlomků,
- `(assert)` — přidání hodnot konstant spjatých s aktuální fází výpočtu, jako podmínky nebo jako konfliktu, do vrcholové úrovně zásobníku asercí,
- `(push)` a `(pop)` — přidání či odebrání úrovně zásobníku asercí.

Získávání hodnot se vždy vztahuje pouze ke vstupním konstantám dané fáze. Pokud je výstup reprezentován výrazem, je vyhodnocen pomocí objektu třídy `Expr::Eval`. Tyto výrazy mohou mít teoreticky neomezenou přesnost, což by vyžadovalo použití dynamických struktur s možností rozšiřující přesnosti. Implementovány jsou ale jen statické typy čísel s plovoucí řádovou čárkou. Problém nastává v momentě, kdy je potřeba takové číslo vypsat zpět ve tvaru textu, aby hodnota zůstala stejná. Výpis řeším zjednodušeně pomocí fixního počtu desetinných míst a ořezáním výsledných hodnot integrací na ještě menší počet, abych ponechal určitý prostor pro případné navýšení desetinných míst z navazujících výpočtů hodnot konstant v SMT řešiči. Jedná se o poměrně náchylné řešení, lepším způsobem by bylo pamatování si načítaných textových reprezentací hodnot konstant a jejich opětovné použití při výpise.

Hodnoty dané fáze lze získat s různou granularitou od nejvyšší po nejnižší se strukturami (či jejími částmi) `Const_*_row` pomocí funkcí `get_step_time_values` počínaje a `get_step_row_values` konče. Vždy však jen v rámci jediné fáze i ODE.

Aserce lze přidávat obecně pomocí funkce `assert`, ale praktičtější je použití funkce `assert_step_row`, která vytvoří formule s veškerým obsahem struktur `Const_ids_row`, `Const_values_row` a výsledků integrace ODE. Současně se provede operace `push`. Vložené aserce jsou interně ukládány do zásobníku, aby bylo možné v případě konfliktu provést i více návratů v řadě. Návrat provede prostřednictvím funkce `assert_last_step_row_conflict` operaci `pop` a přidá negaci všech formulí se vstupními konstantami fáze ze zásobníku (tj. nevyžaduje žádné argumenty).

Rozhraní všech dosud zmíněných funkcí by mělo být nezávislé na konkrétní implementaci adaptéru SMT řešiče.

**Komunikace s SMT řešičem** je zprostředkována pomocí dvojice nepojmenovaných *rour* (angl. *unnamed* či *anonymous pipes*) standardu POSIX. Každá roura je jednosměrná a je realizována v paměti, tj. mimo souborový systém. Cílovou platformou tohoto řešení jsou systémy z rodiny Unix, v rámci nichž by mělo fungovat standardně.

Nastavení komunikace provádí funkce `fork_solver` technikou fork-exec:

1. Vytvoření dvou rour: funkce `pipe`.
2. Vytvoření synovského procesu SMT řešiče: funkce `fork`.
3. Synovský proces přesměruje standardní vstup a výstup do rour: funkce `dup2`.
4. Synovský proces se nahradí procesem SMT řešiče: funkce `execlp`.

Poté rodičovský proces komunikuje prostřednictvím rour přes jejich získané deskriptory. K tomu slouží standardní funkce `write` a `read`, které pracují na úrovni (binárních) bytů. V takové komunikaci je nutné mít dohodnutý nějaký protokol. V našem případě stačí jako zprávy přijímat buď celistvé výrazy, pokud zpráva začíná závorkou, nebo řádky.

Přijímání zpráv je implementováno sekvenčně po jednom znaku, aby bylo možné detekovat konec zprávy a nečíst žádné znaky navíc. To je potenciálně neefektivní — lepším řešením by bylo vyhradit samostatné vlákno pro přijímaní bloků dat poskytovaných jako jednotlivé zprávy. Takové řešení by bylo náročnější na implementaci a vyžadovalo by synchronizaci vláken.

## 4.4 Implementace adaptéru ODE řešiče

V sekci 3.2.1.2 jsem rozebral a oddůvodnil návrh adaptéru pro ODE řešiče pomocí abstraktní třídy a odvozených tříd. Podrobnější návrh komponenty ODE řešiče jsem uvedl v sekci 3.2.2. Jako primární řešiče jsem zvolil odeint a SUNDIALS, které je možné použít jako externí knihovny uvnitř odvozených tříd bez nutnosti dalších rozhraní, jelikož jsou napsány v jazyce C++, resp. v C. SUNDIALS jsem ale nakonec neimplementoval, jelikož nepodporuje některé funkcionality jazyka C++, např. objektové zapouzdření, přetěžování operátorů,

šablonové programování. Jeho nasazení by vyžadovalo ve srovnání s řešičem odeint zavádění dodatečných tříd, které by poskytovaly nějakou abstrakci nad strukturami a funkcemi. Proto jsem zvolil pouze řešič odeint.

Realizaci dynamických specifikací diferenciálních rovnic z textových řetězců a jejich vyhodnocení jsem provedl prostřednictvím vlastních tříd pro výrazy a jejich vyhodnocení podle avizovaného návrhu v sekci 3.2.2.1. Realizace těchto tříd byla popsána v sekci 4.2.

Nejprve popíši abstraktní třídu ODE řešiče, a poté odvozené třídy s konkrétními implementacemi řešení ODE.

### 4.4.1 Abstraktní třída řešiče

Abstraktní třída `ODE::Solver` poskytuje většinu potřebných funkcionalit pro obecný ODE řešič, který přijímá vstupní specifikace diferenciálních rovnic prostřednictvím objektů třídy `Expr` a jejich vyhodnocení provádí interně pomocí objektů třídy `Expr::Eval`. Externí ODE řešič má na starosti pouze samotné řešení rovnic na základě konkrétně definovaných výrazů, ostatní operace deleguje na abstraktní třídu.

Třída není závislá na navrženém vstupním jazyce a celkově na kombinování s SMT řešičem. Měla by být použitelná jako obecný ODE řešič, jen s tím rozdílem, že je možné pro každou ODE nastavit více variant derivací.

#### 4.4.1.1 Specifikace rovnic

Základní forma inicializace řešiče se provádí z páru objektů typu `ODE::Odes_spec` a `ODE::Param_keyss`.

`Odes_spec` reprezentuje množinu struktur typu `Ode_spec`, které reprezentují všechny varianty derivací pro danou funkci — objekty typu `Dt_spec` (alias pro `Expr`). Pro každý objekt `Dt_spec` je sestaveno vyhodnocení typu `Dt_eval` (alias pro `Expr::Eval`).

`Param_keyss` obsahuje množinu struktur `Param_keys`, které reprezentují pole klíčů pro korespondující vyhodnocení `Dt_eval`. Pole klíčů je vždy shodné pro všechny `Dt_spec` (resp. `Dt_eval`) v rámci `Ode_spec`, tj. ODE.

Standardně tedy každá dvojice z množin `Odes_spec` a `Param_keyss` představuje jednu ODE. Každá rovnice je pak řešena nezávisle na ostatních a také klíče má nezávislé. Tento stav nazývám jako *nezávislý*.

Nezávislost rovnic může být výhodná, ale také ne. Pokud kterákoliv rovnice obsahuje jako argument funkci některé ODE (a má za ni být považována), výsledek výpočtu bude chybný, neboť tato funkce bude předložena jako konstanta a nebude se měnit současně s právě řešenou funkcí ODE, protože každá ODE je řešena nezávisle.

**Sjednocení klíčů.** Pokud je potřeba všechny ODE řešit synchronně (myšleno synchronizovaně po krocích integrace), rovnice se na sobě stávají závislými.

Stačí jediná taková závislost, aby byl jako závislý automaticky chápán celý vstup. Takový stav nazývám jako *sjednocený*. Důvodem je, že zejména z implementačních důvodů je vhodné mít pro všechny ODE sjednocené pole klíčů a každá ODE při svém vyhodnocení dosazuje aktuální hodnoty závisle na integracích ostatních rovnic.

Podle tvaru vstupu je možné implicitně rozpoznat, zda se jedná o sjednocený stav, konkrétně podle `Param_keyss` — pokud obsahuje jen jedinou položku `Param_keys`, nebo pokud jsou všechny položky identické. Pak je první položka interpretována jako sjednocené klíče pro všechny ODE.

Sjednocení klíčů lze také explicitně vynutit, ale to je možné pouze při konstrukci řešiče. V tomto případě se vytvoří mapování nesjednocených klíčů na vytvořené sjednocené pomocí číselných indexů, které je přístupné z funkce `cunif_param_keyss_ids`. Druhá strana si tak může pomocí tohoto přeorganizovat své specifikace rovnic a nadále používat pouze sjednocených klíčů. To je sice efektivní varianta, ale relativně nepohodlná. Proto je umožněno i nadále při řešení rovnic poskytovat vstupní hodnoty v nesjednoceném tvaru, které řešič interně sjednotí sám aplikací mapování.

Každé pole klíčů musí splňovat následující pravidla:

- Pokud je přítomen klíč nezávislé proměnné `t`, musí být umístěn na poslední pozici.
- Musí být obsažen alespoň jeden klíč různý od `t`, který je interpretován jako klíč integrované funkce. Jeho pozice musí být následující:
  - nezávislý stav → první pozice,
  - sjednocený stav → pozice na diagonále, tj. pozice klíče odpovídá pozici `Ode_spec` v rámci `Odes_spec`.

Jinou formou inicializace řešiče je pomocí textového vstupu (`std::string` nebo `std::istream`), které jsou delegovány na konstrukci pomocí `Expr`. Vstupní řetězec je formátován jako dvojice výrazů, které mají strukturně shodný tvar s dvojicí `ODE::Odes_spec` a `ODE::Param_keyss`, s jedinou výjimkou: pokud je explicitní požadavek na sjednocení klíčů, je navíc mezi dvojici výrazů nutné vložit token `*`.

#### 4.4.1.2 Řešení rovnic

Podobně jako u specifikací rovnic je možné provést výpočet rovnic buď přímo s parametry požadovaných typů, nebo z textového vstupu. Základní vstup tvoří dvojice objektů typu `ODE::Dt_ids` a `Solver::Contexts`.

`Dt_ids` reprezentuje pole indexů zvolených variant derivací `Dt_spec` pro toto řešení. Tento parametr není závislý na stavu řešiče, zda je či není sjednocený.

`Contexts` je pole objektů typu `Context`. Pokud je voláno řešení rovnic jako sjednocených, je nutné předat jen jediný `Context`, jelikož mají všechny ODE sjednocené klíče a tedy i hodnoty jim přiřazené. Lze však použít i obecnou

funkci řešení rovnic (přijímající `Contexts`), která, v případě, že se řešič nachází ve sjednoceném stavu, se pokusí aplikovat interní mapování nesjednocených pozic na sjednocené (jak bylo popsáno v části o sjednocených klíčích). (Nebo lze předat `Contexts` se shodnými položkami.)

`Context` je třída obalující počáteční a koncové podmínky řešení ODE. Počátečními podmínkami jsou počáteční hodnoty všech parametrů ODE a nezávislé proměnné `t` (viz. vztah (1.2)). Jako koncové podmínky se (zatím) fixně považuje jen koncová hodnota nezávislé proměnné `t`.

Počáteční hodnota `t` se vždy uvádí odděleně od všech ostatních parametrů a nesmí být v nich duplicitně obsažena. V případě, že je daná ODE závislá na `t`, je její hodnota přidávána automaticky uvnitř řešiče `Solver`.

Řešení rovnic se provádí různými členskými funkcemi `Solver::solve`* závisle na předávaných parametrech a na požadavek sjednocení klíčů. Pro řešení jen jediné ODE slouží `solve_ode`. Pro řešení všech rovnic slouží `solve_odes`, která samostatně detekuje, zda zvolit sjednocený výpočet. Pro explicitní požadavek na sjednocený výpočet slouží `solve_unif_odes`, která selže pokud klíče rovnic nejsou sjednoceny. Poslední možností je funkce `solve`, která přijímá textový vstup ve tvaru dvojice výrazů se stejnou strukturou jako mají typy `Dt_ids` a `Context(s)`. Pokud je v druhém výrazu specifikován jen jeden `Context`, jsou rovnice řešeny sjednoceně.

#### 4.4.1.3 Další operace

Dalšími operacemi obsaženými ve veřejném rozhraní třídy jsou:

- `set_step_size` — nastaví (počáteční) velikost kroku integrací,
- `add_ode_spec` — přidá specifikaci další ODE s klíči parametrů,
- `is_unified` — vrátí příznak, zda je řešič ve sjednoceném stavu; pokud není a dosud to nebylo ověřeno, je ověřeno, zda skutečně sjednocený není,
- `cparam_keyss` — zkonstruuje objekt typu `ODE::Param_keyss` s klíči parametrů všech ODE zvlášť,
- `cunif_param_keys` — vrátí referenci na sjednocené klíče všech ODE typu `ODE::Param_keys`; selže, pokud `is_unified` není pravdivé,
- `ctrajects`, `cunif_traject` — vrátí referenci na objekt typu `Solver::Traject(s)` (viz. dále),
- lidsky čitelný výpis všech obsažených rovnic řešiče.

`Traject` je třída, která shromažďuje průběh (*trajektorii*) integrace jedné ODE, tj. obsahuje pole hodnot nezávislé proměnné `t` a hodnot všech parametrů ODE. Tyto jsou platné jen v rámci jednoho řešení rovnic — hodnoty jsou při každém volání funkce `solve`* z kapacitních důvodů resetovány.

`Trajects` je pole objektů `Traject` o velikosti počtu ODE řešiče.

Třída `Solver` (a její odvozené třídy) je dostupná jako knihovna. Také je možné ji využít jako aplikaci, k čemuž poskytuje šablonovou třídu `Solver::Run<S>`, kde `S` je konkrétní odvozená třída s implementací integrace. Aplikace používá inicializaci a řešení rovnic s textovými vstupy a chová se jako filtr: na inicializaci odpoví výpisem `cparam_keyss` nebo `cunif_param_keys` a na každé řešení (`solve`) výpisem výsledku. Je-li specifikován výstupní soubor, jsou do něj průběžně ukládány výpisy objektů `Traject`(`s`).

### 4.4.2 Odvozené třídy

Třídy odvozené od `ODE::Solver` musí dodat implementace integrace rovnic. `Solver` obsahuje tři virtuální neveřejné metody: `eval_ode`, `eval_odes` a `eval_unif_odes`. `eval_odes` ve výchozím tvaru jen vyplní pole výsledků pomocí jednotlivých volání `eval_ode`; zbylé dvě funkce nejsou implementovány. Odvozená třída tedy musí definovat jak nezávislé integrování jednotlivých rovnic, tak synchronní integraci všech rovnic, mají-li sjednocené klíče.

**`Euler`** poskytuje triviální implementaci integrace pomocí explicitní Eulerovy metody (viz. vztah (2.1)). Tato třída slouží zejména k demonstračním a testovacím účelům, jelikož je Eulerova metoda nepřesná. Třída není závislá na externích knihovnách.

Spustitelná aplikace třídy je umístěna v souboru `bin/applet/euler`.

**`Odeint`** využívá některých funkcí ODE řešiče odeint (viz. sekce 2.3.1.3). Odeint je realizován výhradně uvnitř hlavičkových souborů v rámci C++ knihoven Boost, které třída `Odeint` částečně zahrnuje. Zatím je použita pouze výchozí funkce `odeint::integrate`, které jsou z třídy `Solver` v každém kroce integrace poskytovány vypočtené hodnoty z objektů typu `Dt_eval`. Funkce používá metodu Dormand–Prince 5, což je explicitní adaptivní Runge–Kutta metoda (viz. sekce 2.3.1.2).

Implementace třídy `Odeint` je triviální, neboť řešič odeint nevyžaduje žádnou inicializaci a pouze se volá funkce `integrate` s počátečními a koncovými hodnotami a s funkčními objekty.

Spustitelná aplikace třídy je umístěna v souboru `bin/applet/odeint`.

## 4.5 Implementace zpracování vstupu

Návrh zpracování vstupu je uveden v sekci 3.2.3. Pro tyto účely byla vytvořena třída `Parser`. V první řadě je vstup předzpracován (viz. další sekce). Poté je ke zpracování vstupu použit objekt třídy `Expr`, který by měl jako přímé potomky obsahovat pouze další výrazy `Expr`, jelikož v kořenové úrovni nejsou tokeny povoleny (po předzpracování). Výrazy v první úrovni představují příkazy,

které jsou procházeny rekurzivně, pokud se nějak dotýkají ODE řešiče. Ostatní výrazy, zejména ty týkající se výhradně SMT řešiče, jsou ponechány částečně nebo zcela nezpracovány.

Hlavním úkolem je zpracování příkazů `define-dt` a `int-ode`, z nichž je nutné shromáždit specifikace všech ODE a názvy konstant vstupující do `int-ode` jako vstupní argumenty. K tomu slouží struktura `Odes`, což je pole struktur `Ode`. `Ode` je pětice těchto struktur:

1. `Ode_key` — klíč (identifikátor) ODE,
2. `Dt_keys` — pole klíčů (identifikátorů) variant derivací,
3. `Ode_spec` — pole specifikací rovnic derivací (viz. sekce 4.4.1.1) ve stejném pořadí, jako `Dt_keys`,
4. `Param_keys` — společné klíče nepřímých argumentů pro všechny rovnice v `Ode_spec`; není kontrolováno, zda jsou klíče napříč všemi rovnicemi dané ODE shodné,
5. `Const_ids_rows` — pole identifikátorů vstupních konstant jednoho příkazu `int-ode`, jak bylo diskutováno v sekci 4.3.

S výjimkou `Const_ids_rows` pochází všechny ostatní hodnoty z příkazů `define-dt`. Položky `Const_ids_row` jsou v rámci příslušného klíče `Ode_key` ukládány v pořadí, v jakém jsou ve vstupu čteny příkazy `int-ode`. (Určení fází podle konstant názávislých proměnných `t` není zodpovědností zpracování vstupu.)

Hodnoty klíčů, včetně těch z `Const_ids_rows`, jsou také duplicitně ukládány do vyhledávacích stromových struktur, aby byly rychle dohledatelné.

Příkazy, které jsou zpracovány, jsou:

- `set-logic` — povoleny jsou všechny logiky zmíněné v sekci 3.1.1.2, a to *včetně* logik bez volných funkčních symbolů,
- `define-dt` — uloží se specifikace varianty derivace dané ODE; při prvním výskytu klíče `Ode_key` dojde k deklaraci ODE, tj. uložení klíče `Ode_key` a klíčů nepřímých argumentů `Param_keys` společných pro všechny varianty derivací,
- `define-ode-step`,
- `int-ode` — ze vstupních argumentů příkazu se vytvoří jedna položka `Const_ids_row` pro odpovídající klíč `Ode_key`, která je vložena na konec pole `Const_ids_rows`; příkaz je do SMT vstupu transformován jako *konstanta* (čemuž předchází její deklarace); vstupní argumenty příkazu *musí být globální identifikátory.*

Dále je zpracován každý token. Dosud se provádí jen transformace záporných numerických literálů na výrazy.

Komponentu je možné použít jako knihovnu, nebo pomocí textové aplikace `bin/applet/parser`, která na standardní výstup vypíše vstup pro SMT řešič a na chybový výstup vstup pro ODE řešič. Pomocí přepínače `-E` lze také provést pouze předzpracování vstupu a výsledek vypsat na standardní výstup.

## 4.6 Implementace předzpracování vstupu

Realizace předzpracování vstupu navazuje na návrh uvedený v sekci 3.2.4. Výsledkem je třída `Preprocess`, která je implementována uvnitř třídy `Expr` (viz. sekce 4.2), jelikož lze předzpracování použít pro libovolný textový prefixový vstup obsahující komentáře a makra stejné se vstupním jazykem (viz. sekce 3.1.1), a protože je celé zpracování silně vázáno na interní objekt třídy `Expr`. Samotná třída `Expr` je však nezávislá na implementaci třídy `Preprocess`.

Globální a lokální makra jsou uložena zvlášť ve vyhledávacích stromech `Macros_map` a `Lets_map`. `Macros_map` obsahuje jako hodnoty dvojice `Macro_param_keys` (názvy parametrů makra) a `Macro_body` (alias pro `Expr`). `Lets_map` obsahuje jako hodnoty zásobníky objektů `Let_body` (alias pro `Macro_body`).

Příkazové makro `#for` je zatím implementováno jen ve formě celočíselného vzestupného rozsahu řídící proměnné s jednotkovým krokem (varianta 1). Pro ostatní případy lze využít rekurzivních volání uživatelských maker.

**Aritmetické expanze** výrazů jsou implementovány pomocí třídy `Expr::Eval` (viz. sekce 4.2). Ve skutečnosti se jedná o expanzi tokenu, jelikož výrazy jsou předcházeny tokenem `$`. Makra `#if` a `#for` používají ke svému vyhodnocení aritmetickou interpretaci tokenů, k čemuž je použita buď aritmetická expanze výrazu, nebo, v případě, že se jedná o literál, získání aritmetické hodnoty z objektu třídy `Expr_token` pomocí `get_value` (viz. sekce 4.2).

Vzhledem k tomu, že C++ není dynamicky typovaný jazyk, a požadovaný typ argumentů je zjišťován dynamicky z textového vstupu, bylo nutné programově odlišit případy použití reálných a celých čísel. K reprezentaci hodnoty jsem použil konstrukt `union`, který umožňuje paměťovou oblast interpretovat jako různé typy (ačkoli pouze staticky).

**Zpracování maker v rámci tokenu** je provedeno tak, že se token rozdělí na části podle znaků `#`[38]. Pokud jsou obsaženy alespoň dvě takové části, přidá se každá do výrazu jako nový token a zpracuje se zvlášť, a poté je výsledek všech expanzí spojen do právě jednoho tokenu, který může být i prázdný. Rezervovaná makra volaná jako `#` a `##` jsou implementována takto:

- je přidáno globální makro s prázdným názvem,
- při rozdělení tokenu na části je pro každou nalezenou část (kromě té poslední), která je rovna jedinému znaku `#`, odmazán první znak z následující části, který nutně musí být roven `#`. Tím je simulováno volání makra `##`.

**Escape sekvence** `\#` je implementována tak, že se do části tokenu, která začíná znakem `#`, a která následuje za částí, která končí znakem `\`, přidá další znak `#` na začátek. Pokud je následně při zpracování expanze makra toto detekováno, odmaže se první znak a expanze není provedena.

[38] Každá část může obsahovat nejvýše jeden znak `#` právě na první pozici.

Komponentu lze použít jako knihovnu nebo pomocí aplikace `bin/applet/-parser` (viz. sekce 4.5).

## 4.7 Realizace řídící komponenty

Úlohou centrální komponenty je řídit SMT a ODE řešič (sekce 4.3 a 4.4) a průběh kombinovaného výpočtu. Postupoval jsem dle návrhu komponenty uvedeném v sekci 3.2.1.4 a návrhu řídícího algoritmu ze sekce 3.2.5. Komponenta je však realizována jako *prototyp*, některé funkcionality chybí nebo jsou zjednodušeny.

Řídící komponenta je umístěna v šablonové třídě `Solver<OSolver>`, která komunikuje s oběma řešiči výhradně pomocí programového rozhraní vlastních tříd `SMT::Solver` a `ODE::Solver`; teprve na nich leží zodpovědnost konkrétní realizace propojení s řešiči třetích stran. `ODE::Solver` slouží jako rozhraní, konkrétní implementaci obsahuje zvolená odvozená třída `OSolver`. Výhledově by bylo lépe použít také třídu SMT řešiče jako rozhraní s volitelnou implementací (jak bylo uvedeno v sekci 4.3) jako šablonového parametru.

Oproti navrženému základnímu řídícímu algoritmu byla aplikována následující zjednodušení:

- Fáze výpočtu jsou brány v pořadí, ve kterém se nachází příkazy `int-ode`; je zodpovědnost uživatele, aby hodnoty vstupních konstant nezávislé proměnné `t` tvořily neklesající posloupnost. Počet příkazů `int-ode` na jednu fázi, včetně shodných ODE, však není omezen.
- Rozložení všech fází musí navíc být pro všechny ODE *stejné*. To prakticky znamená to, že všechny příkazy `int-ode` musí používat pouze jednu společnou sadu konstant nezávislé proměnné `t`.
- Všechny odlišné ODE jsou vkládány do jediné společné soustavy; vícenásobný výskyt ODE v rámci fází je považován za nový nezávislý systém */* !!! tohle je asi blbě ... Kaskádní kompozice asi nebude fungovat - bude kolize na klíči ODE v ODE řešiči. Aby se počítaly synchronně, musí klíč odpovídat názvu ODE, takže jen vložení jiné vstupní konstanty nepomůže. Mělo by ale fungovat vícenásobná definice více ODE s odlišnými názvy - ale se stejným tělem .. */*. Řešič `ODE::Solver` je tedy vždy používán ve sjednoceném stavu.

Výpočet se provádí funkcí `solve`, která zavolá funkci `do_step` pro počáteční fázi a počítá se dokud není dosaženo poslední fáze (pak bylo nalezeno splňující ohodnocení vstupu), nebo dokud není proveden návrat z počáteční fáze (pak vstup není splnitelný).

Interakce s oběma řešiči spočívá pouze v sestavení či přeuspořádání požadovaných vstupních struktur.

**Inicializace ODE řešiče** spočívá v zaslání specifikací rovnic a klíčů nepřímých argumentů v nezávislém tvaru. S výhodou je využito toho, že `ODE::Solver` umí explicitně vynutit sjednocení klíčů a také později sám provádět mapování pozic hodnot nezávislých kontextů do sjednoceného. Řídící komponenta tedy kromě příznaku v konstruktoru řešiče, který vyžaduje sjednocení, je zcela oproštěna od této skutečnosti a pracuje s rovnicemi jako s nezávislými.

*/* řešení ode - jak je to s entries? viz. hore */*

Celý nástroj lze použít jako knihovnu, nebo jako aplikaci. Aktuální podoba aplikace je umístěna v souboru `bin/sos_odeint`, která používá odeint pro řešení ODE (třída `ODE::Odeint`).

## 4.8 Seznam dalších úkolů

- Podrobnější dokumentace zdrojových kódů.
- Implementace řídícího makra `#for` ve všech variantách.
- Kontrola příkazů `define-dt`, zda mají všechny v rámci dané ODE shodné klíče nepřímých argumentů.
- Lepší zacházení s výstupními textovými hodnotami z SMT řešiče, pokud se do něj následně posílají zpět jako podmínky.
- Oddělení rozhraní od implementace ve třídě `SMT::Solver`; odvozenou třídu zavést jako druhý šablonový parametr třídy `Solver`.
- Vyhrazení samostatného vlákna pro příjem odchozích zpráv SMT řešiče po blocích.
- Implementace odvozené třídy pro implementaci ODE řešiče SUNDIALS.
- Umožnit obecnější nastavení fází výpočtu v řídící komponentě.
- Umožnit obecnější koncové podmínky diferenciálních rovnic, např. podle koncové hodnoty funkce.
- Implementace efektivnějšího algoritmu redukujícího počet operací ověření splnitelnosti vstupu.

Kapitola 5

# Experimentální část

# Závěr

# Literatura


[1] Wikipedia: SAT Modulo Theories. [online], [cit. 2017-08-16]. Dostupné z: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[2] Barrett, C.; Fontaine, P.; Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). [online], 2016, [cit. 2017-08-17]. Dostupné z: http://www.SMT-LIB.org

[3] Cook, S. A.: The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, New York, NY, USA: ACM, 1971, s. 151–158, doi:10.1145/800157.805047, [cit. 2018-03-20]. Dostupné z: http://doi.acm.org/10.1145/800157.805047

[4] Biere, A.: Bounded Model Checking. In *Handbook of Satisfiability*, editace A. Biere; M. Heule; H. van Maaren; T. Walsh, kapitola 14, IOS Press, 2009, s. 457–481, doi:10.3233/978-1-58603-929-5-457, [cit. 2018-03-07].

[5] Bradley, A. R.; Manna, Z.: *The Calculus of Computation*. Springer-Verlag Berlin Heidelberg, 2007, ISBN 978-3-540-74112-1, 366 s., [cit. 2018-03-12].

[6] Wikipedia: Ordinary differential equation. [online], [cit. 2018-03-07]. Dostupné z: https://en.wikipedia.org/wiki/Ordinary_differential_equation

[7] Ahnert, K.; Mulansky, M.: Odeint – Solving Ordinary Differential Equations in C++. *AIP Conf. Proc. 1389*, 2011: s. 1586–1589, doi:10.1063/1.3637934, [cit. 2017-08-17].

[8] Alexandre dit Sandretto, J.; Chapoutot, A.: Validated Explicit and Implicit Runge-Kutta Methods. *Reliable Computing electronic edition*, ročník 22, Červenec 2016, [cit. 2018-03-07]. Dostupné z: https://hal.archives-ouvertes.fr/hal-01243053

[9] Peter Philip: Ordinary Differential Equations. [online], 2017, lecture notes [cit. 2018-03-07]. Dostupné z: http://www.math.lmu.de/~philip/publications/lectureNotes/ODE.pdf

[10] de Moura, L.; Bjørner, N.: Satisfiability modulo theories: introduction and applications. *Commun. ACM*, ročník 54, č. 9, 2011: s. 69–77, [cit. 2018-03-06].

[11] Eén, N.; Sörensson, N.: MiniSAT. [online], 2008, [cit. 2017-08-23]. Dostupné z: http://minisat.se

[12] Biere, A.; Heule, M.; van Maaren, H.; aj.: Satisfiability Modulo Theories. In *Handbook of Satisfiability*, editace C. Barrett; R. Sebastiani; S. A. Seshia; C. Tinelli, kapitola 12, IOS Press, 2008, s. 737–797, [cit. 2017-11-21].

[13] Cok, D. R.: The SMT-LIBv2 Language and Tools: A Tutorial. [online], 2013, [cit. 2017-11-21]. Dostupné z: http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf

[14] Barrett, C.; Fontaine, P.; Tinelli, C.: The SMT-LIB Standard, Version 2.6. 2017, [cit. 2017-11-21].

[15] Sharygina, N.: OpenSMT. [online], 2012, [cit. 2017-08-16]. Dostupné z: http://verify.inf.usi.ch/opensmt

[16] Bruttomesso, R.; Pek, E.; Sharygina, N.; aj.: The OpenSMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ročník 6015, Springer, Paphos, Cyprus: Springer, 2010, s. 150–153, doi:10.1007/978-3-642-12002-2_12, [cit. 2018-03-07].

[17] Kshitij Bansal, F. B., Clark Barrett: CVC4. [online], 2017, [cit. 2017-10-18]. Dostupné z: http://cvc4.cs.stanford.edu/web/

[18] Barrett, C.; Conway, C. L.; Deters, M.; aj.: CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, *Lecture Notes in Computer Science*, ročník 6806, editace G. Gopalakrishnan; S. Qadeer, Springer, Červenec 2011, s. 171–177, snowbird, Utah. Dostupné z: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf

[19] Ernst Hairer and Christian Lubich: Numerical solution of ordinary differential equations. [online], 2015, introductory text [cit. 2018-03-09]. Dostupné z: https://na.uni-tuebingen.de/~lubich/pcam-ode.pdf

[20] Wikipedia: Numerical methods for ordinary differential equations. [online], [cit. 2018-03-08]. Dostupné z: https://en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations

[21] Atkinson, K.; Han, W.; Jay, L.; aj.: *Numerical Solution of Ordinary Differential Equations*. John Wiley & Sons, Inc., 2 2009, ISBN 978-0-470-04294-6, 272 s., [cit. 2018-03-09].

[22] Endre Süli: Numerical Solution of Ordinary Differential Equations. [online], 2014, lecture notes [cit. 2018-03-09]. Dostupné z: `https://people.maths.ox.ac.uk/suli/nsodes.pdf`

[23] Woodward, C. S.: SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers. [online], 2005, [cit. 2017-08-16]. Dostupné z: `https://computation.llnl.gov/projects/sundials`

[24] Hindmarsh, A. C.; Brown, P. N.; Grant, K. E.; aj.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, ročník 31, č. 3, 2005: s. 363–396, [cit. 2017-08-17].

[25] Ahnert, K.; Mulansky, M.: Odeint. [online], 2012, [cit. 2017-08-16]. Dostupné z: `http://headmyshoulder.github.io/odeint-v2`

[26] Dawes, B.; Abrahams, D.: Boost Library Documentation. [online], [cit. 2017-08-26]. Dostupné z: `http://www.boost.org/doc/libs`

[27] Ishii, D.; Ueda, K.; Hosobe, H.: An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, ročník 13, č. 5, 2011: s. 449–461, [cit. 2018-03-11].

[28] Niehaus, J.: iSAT-ODE. [online], 2010, [cit. 2017-08-16]. Dostupné z: `http://www.avacs.org/tools/isatode`

[29] Eggers, A.; Ramdani, N.; Nedialkov, N.; aj.: Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. *International Conference on Software Engineering and Formal Methods (SEFM)*, ročník 9, 2011, [cit. 2018-03-11].

[30] Nedialkov, N.: VNODE-LP–a validated solver for initial value problems in ordinary differential equations. [online], 2006, [cit. 2017-08-20]. Dostupné z: `http://www.cas.mcmaster.ca/~nedialk/vnodelp`

[31] Jekyll; Bones, S.: dReal. [online], 2016, [cit. 2017-08-16]. Dostupné z: `http://dreal.github.io`

[32] Bae, K.; Kong, S.; Gao, S.: SMT Encoding of Hybrid Systems in dReal. In *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, *EPiC Series in Computing*, ročník 34, editace G. Frehse; M. Althoff, EasyChair, 2015, ISSN

2398-7340, s. 188–195, doi:10.29007/s3b9, [cit. 2018-03-06]. Dostupné z: https://easychair.org/publications/paper/4Qr

[33] Gao, S.; Kong, S.; Clarke, E.: Satisfiability Modulo ODEs. *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, [cit. 2018-03-11].

Příloha **A**

# Seznam použitých symbolů a zkratek

**ANSI** American National Standards Institute. 21

**API** Application programming interface. 15

**ASCII** American Standard Code for Information Interchange. 27

**BDF** Backward differentiation formula. 18

**BMC** Bounded Model Checking. 4, 12, 23, 25

**CNF** Conjunctive normal form. 11

**CTL** Computation tree logic. 4

**CVC** Cooperating Validity Checker. 15, 52

**DAE** Differential-algebraic equation. 21

**DIMACS** Center for Discrete Mathematics and Theoretical Computer Science. 11

**DPLL** Davis–Putnam–Logemann–Loveland. 11, 12, 48

**FOL** First-order logic. 5, 6, 13

**GNU** GNU's Not Unix!. 42

**IVP** Initial value problem. 8, 16, 21

**LTL** Linear time logic. 4

**MPI** Message Passing Interface. 21

**NP** Nondeterministic polynomial time. 3, 4

**ODE** Ordinary differential equation. vii, ix, 2, 4, 7–9, 11, 15–17, 19, 21–23, 26, 27, 30–32, 35, 41–49, 53–59, 61, 62, 72

**OpenMP** Open Multi-Processing. 21

**OS** Operační systém. 50

**POSIX** Portable Operating System Interface. 49, 54

**QBF** Quantified Boolean formulas. 4

**SAT** Boolean satisfiability problem. vii, ix, 2–4, 9, 11–15, 22, 23

**SIMD** Single instruction multiple data. 21

**SMT** Satisfiability Modulo Theories. vii, ix, 2, 4, 5, 9, 11–15, 23, 25–32, 41–43, 45–49, 52–55, 59, 61, 62, 72

**SOS** SMT+ODE solver. 49

**STL** Standard Template Library. 49

**SUNDIALS** SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers. 21, 42, 54, 62

**TMP** Template Metaprogramming. 21