Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Diplomová práce

SMT řešič s diferenciálními rovnicemi

Bc. Tomáš Kolárik

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

27. března 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 27. března 2018 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Kolárik, Tomáš. *SMT řešič s diferenciálními rovnicemi.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

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

Abstract

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

Obsah

Odkaz na tuto práci vi

Úvod **1**
Cíl práce 2
Požadavky zadání 2
Rešerše 2

1 Teoretická část **3**
1.1 Formulace problémů 3
1.1.1 Problém splnitelnosti: SAT 3
1.1.2 Satisfiability Modulo Theories (SMT) 4
1.1.3 Ordinary differential equation (ODE) 7
1.2 Hybridní systémy 8

2 Možnosti řešení problematiky **11**
2.1 SAT řešiče 11
2.2 Řešení SMT problému 12
2.2.1 SMT-LIB standard 13
2.2.2 SMT řešiče 14
2.3 Numerické metody řešení ODE 15
2.3.1 Klasické numerické metody 16
2.3.2 Garantovaná řešení 21
2.4 Hybridní řešiče 22

3 Návrh zvoleného řešení **25**
3.1 Vstupní jazyk 25
3.1.1 Syntaxe jazyka 26
3.1.2 SMT konstrukty 27
3.1.3 Konstrukty ODE – verze 1 28
3.1.4 Struktura a použití jazyka --verze 1 29
3.1.5 Konstrukty ODE – verze 2 31

3.1.6 Struktura a použití jazyka --verze 2 32

4 Realizace **35**

5 Testy a výsledky **37**

Závěr **39**

Literatura **41**

A Seznam použitých symbolů a zkratek **45**

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

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

[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.

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 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

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

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

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

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 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

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

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

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 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].

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

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

[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].

[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.

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.

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*.

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. 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,

[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.

- `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 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].

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.

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

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.

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

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.

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

[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.

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).

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)).

Příklady dalších metod:

$$q=1,\ k=p=2:\ y_{n+2}-y_{n+1}=\frac{h}{2}\left(\qquad\qquad 3f_{n+1}-\ f_n\right)$$

$$q=2,\ k=p=3:\ y_{n+3}-y_{n+2}=\frac{h}{12}\left(\qquad 23f_{n+2}-16f_{n+1}+5f_n\right)$$

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

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)).

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

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 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].

[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 $q = 1$.

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*: $S_i \coloneqq i - 1$, 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*: $S_i \coloneqq s$; $\exists_{i,j}\, i \leq j :\ a_{i,j} \neq 0$.

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:

0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
	$\frac{1}{2}$	$\frac{1}{2}$

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:

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

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.

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).

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

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 δ-splnitelná (δ`-sat`) [31]. Nesplnitelnost je rozhodnuta exaktně a doložena důkazem; δ-splnitelnost je numericky aproximována (resp. exaktně rozhodnuta na zjednodušené formuli) s přesností δ (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

/ komentář */*

3.1 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 přímo kompatibilní, tzn. že *není konformní* se standardem SMT-LIB, který je relativně robustní, umožňuje nastavovat různé logiky, poskytuje přídavné příkazy pro interaktivní použití v konzoli (např. `check-sat`, `push`, `exit`, ...) a také definuje výstupní jazyk. Náš řešič se omezuje pouze na (vstupní) specifikace modelů v *nelineární teorii reálných čísel bez kvantifikátorů* s možností používat volné funkční symboly (v SMT-LIB tomu odpovídá logika `QF_UFNRA`) společně s ODE, tj. neumožňuje nastavovat různé logiky a interpretovat vstup jinak, než jako ověření jeho splnitelnosti.

Jazyk používá plně uzávorkovanou prefixovou notaci[24]. Podmnožina příkazů týkající se jen specifikace SMT formulí je převzata ze SMT-LIB, a k nim jsou ortogonálně doplněny příkazy týkající se nadstavby o ODE. Obě skupiny jsou popsány ve zvláštních podsekcích.

Jsou tři různé odpovědi na vstupní specifikaci, stejně jako u SMT řešičů (sekce 2.2): `unsat` s důkazem, `sat` s modelem, nebo `unknown` (u nás nejčastěji z důvodu neúplných počátečních podmínek pro ODE).

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

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

Značení. `<>` není součástí syntaxe a ohraničuje či seskupuje argumenty (nejsou-li již nějak ohraničeny); `*` značí, že dotčený ohraničený řetězec se může opakovat vícekrát nebo být prázdný; `+` je stejné s `*` kromě toho, že řetězec se musí vyskytnout alespoň jednou; `|` značí více možností pro jednu pozici argumentu.

3.1.1 Syntaxe jazyka

Vstupní jazyk je sekvence *tokenů*, *výrazů*, bílých znaků a komentářů.

Bílé znaky. Povolenými bílými znaky jsou:

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

S výjimkou oddělení tokenů jsou bílé znaky ignorovány.

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

Token je sekvence znaků závisle na typu tokenu. Dělí se na *identifikátory* a *literály*.

Identifikátory sestávají z alfanumerických znaků a znaků

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

Musí být předem deklarovány nebo rezervovány a nesmí začínat číslicí[25]. 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. */* ala smtlib */*.

Literály jsou bezejmenné konstanty nějakého druhu. Numerické obsahují číslice a případně desetinnou tečku (`.`); na začátku nejsou povoleny přídavné `0` ani znaménko; desetinná čísla musí obsahovat číslici před i po desetinné tečce; není podporován semilogaritmický tvar. Booleovské literály jsou `true` a `false`.

Výraz je sekvence

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

[25] 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.

kde `<expr>*` jsou vnořené výrazy a `<token>` je token povolený v kontextu daného výrazu. První token výrazu odlišuje, zda a o jaký se jedná příkaz, funkci či literál. Příkazy nemusí mít druh návratové hodnoty, funkce ano.

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

3.1.2 SMT konstrukty

Je použita pouze podmnožina konstruktů týkající se zvolené teorie uvedené v sekci 3.1. 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 (operátory) se standardní sémantikou:

```
+  -  *  /  ^  =  <  >  <=  >=  not  and  or  xor  =>
```

/ asociativity, chainability, ite, distinct */ /* ^ v CVC4 jen pro prirozena cisla, v OpenSMT vubec */*

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

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

Argumenty příkazu:

- `<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)
```

`define-fun` rozšiřuje[26] `declare-fun` o definici funkce:

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

se shodnými argumenty kromě:

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

- `(<arg> <arg_sort>)*` — výčet párů identifikátorů názvu argumentů funkce a jejich druhů,
- `<expr>` — výraz 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))
```

`assert` zavádí formule modelu, které musejí být splněny[27]:

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

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

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

3.1.3 Konstrukty ODE – verze 1

V kontextu ODE lze zjednodušovat syntaxi příkazů, jelikož je pravidlem, že název nezávislé proměnné v ODE je vždy `t`, a že druhy funkcí i jejich derivací a nezávislé proměnné je `Real`. Tyto skutečnosti tedy není nutné zmiňovat.

`define-dt` slouží k definici derivace funkce. Jedná se o modifikovanou formu příkazu `define-fun` ze sekce 3.1.2:

```
(define-dt <name> <fun> <expr>)
```

s argumenty:

- `<name>` — název identifikátoru derivace funkce (pro jednu funkci lze definovat více derivací),
- `<fun>` — název identifikátoru derivované funkce nezávislé proměnné `t`, která se může vyskytovat ve výrazu `<expr>`,
- `<expr>` — výraz definující tvar derivace funkce, (ne nutně) obsahující i původní funkci `<fun>` či nezávislou proměnnou `t`; `<fun>` se uvádí bez závislosti na `t`, tj. jako konstanta.

Příklady:

```
(define-dt dx x 1)
(define-dt dy_1 y (- (* (/ 3 t) y) 2))
```

Tyto funkce je nutné používat výhradně uvnitř ODE příkazů.

`dt-int` slouží k deklaraci funkce, která ponese výslednou hodnotu integrace vypočtenou interním ODE řešičem. Hodnoty těchto funkcí jsou vypočítávány postupně, zpočátku nejsou známy. Příkaz má formu

```
(dt-int <dt> <t_init> <t_end> <init>)
```

[27] Je ověřena splnitelnost konjunkce všech těchto formulí, podobně jako v SAT řešičích.

s argumenty:

- `<dt>` — identifikátor pomocné konstanty derivace funkce nesoucí hodnotu některé varianty derivace vzniklé příkazem `define-dt`,
- `<t_init>` — počáteční hodnota nezávislé proměnné `t`,
- `<t_end>` — koncová hodnota `t`,
- `<init>` — počáteční hodnota funkce (tj. v bodě `<t_init>`), která je v `<dt>` derivována.

Příklad:

```
(dt-int dy_1 t_0 t_1 y_0)
```

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

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

3.1.4 Struktura a použití jazyka – verze 1

V této podsekci je uvedena doporučená struktura vstupu, aby 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 dynamicky měnil. Jelikož je vstup statický, je pro modelování průběhu nutné využít několika konstant o předem známém počtu, tzn. že už při modelování systému je nutné určit *počet kroků simulace*[28]. Kroky jsou určeny různými hodnotami nezávislé proměnné `t`, mezi nimiž dochází k integraci neznámých funkcí. V těchto krocích také dochází se skokům: na základě hodnot integrovaných i jiných funkcí se mění diskrétní stav.

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

Deklarace a inicializace konstant. Všechny konstanty použité ve všech krocích simulace musejí být deklarovány a konstanty počátečního kroku musí být i definovány. Každé konstantě je vhodné dát jako příponu číslo kroku, kterého se týká.

Je vhodné deklarovat konstanty pro nezávislou proměnnou `t`, průběh neznámých funkcí, diskrétní stav a také konstanty pro volby derivací neznámých funkcí[29].

Příklad:

```
(define-fun t0 () Real 0) (define-fun y0 () Real 1)
(define-fun m0 () Bool false)
(declare-fun t_0  () Real) (declare-fun t_1  () Real)
```

[28] Pozor, jedná se o jiné kroky než kroky ODE řešiče.

[29] Jedná se o pomocné konstanty odlišující použité derivace z `define-dt`, které jsou jednoznačně určeny na základě aktuálního diskrétního stavu modelu. Derivace se v posledním kroku již nevyužívá, není pro ni tedy nutné deklarovat kontantu.

```
(declare-fun y_0  () Real) (declare-fun y_1  () Real)
(declare-fun m_0  () Bool) (declare-fun m_1  () Bool)
(declare-fun dy_0 () Real)
(assert (and (= t_0 t0) (= y_0 y0) (= m_0 m0) ))
```

Nastavení kroků znamená definovat hodnoty konstant `t_i`. Nejjednodušším způsobem je zavedení konstatní délky kroku `T`, např.:

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

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

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

Propojení derivací s diskrétním stavem. Integrace neznámých funkcí se i s ostatními funkcemi vkládají do asercí společně s diskrétními konstantami, čímž dochází k propojení diskrétní a spojité domény modelu. Jednak se propojují pomocné konstanty derivací s diskrétním stavem, jednak konstanty průběhů neznámých funkcí s jejich integracemi.

Doporučujeme zkonstruovat pomocnou funkci `connect`, př.:

```
(define-fun connect ((m1 Bool) (dy1 Real)
                     (t1 Real)  (t2 Real)
                     (y1 Real)  (y2 Real)) Bool
    (and (or (and      m1  (= dy1 dy_run ))
             (and (not m1) (= dy1 dy_idle)) )
         (= y2 (dt-int dy1 t1 t2 y1))
))
```

Příklad s použitím funkce `connect`:

```
(assert (and (connect m_0 dy_0 t_0 t_1 y_0 y_1)
             (connect m_1 dy_1 t_1 t_2 y_1 y_2)
))
```

Definice skoků. Skoky, tj. změny diskrétního stavu, lze snadno definovat též pomocí asercí mezi sousedními stavy.

Doporučujeme zkonstruovat pomocnou funkci `jump`, př.:

```
(define-fun jump ((m1 Bool) (m2 Bool) (y2 Real)) Bool
    (or (and      m1       m2  (<  y2 bound_1))
        (and      m1  (not m2) (>= y2 bound_1))
        (and (not m1) (not m2) (>  y2 bound_2))
        (and (not m1)      m2  (<= y2 bound_2))
))
```

Příklad s použitím funkce `jump`:

```
(assert (and (jump m_0 m_1 y_1)
             (jump m_1 m_2 y_2)
))
```

3.1.5 Konstrukty ODE – verze 2

V kontextu ODE lze zjednodušovat syntaxi příkazů, jelikož je pravidlem, že název nezávislé proměnné v ODE je vždy `t`, a že druhy funkcí i jejich derivací a nezávislé proměnné je `Real`. Tyto skutečnosti tedy není nutné zmiňovat.

`declare-ode` slouží k deklaraci neznámé funkce a všech jejich variant derivací. Současně s variantami derivací jsou uvedeny názvy diskrétních stavů — módů, se kterými budou propojeny:

```
(declare-ode <fun_name> ((<dt> <mode>)+))
```

Argumenty:

- `<fun_name>` — název identifikátoru neznámé funkce,
- `(<dt> <mode>)+` — výčet párů identifikátorů variant derivací funkce `<fun_name>` (viz. příkaz `define-dt`) a jejich odpovídajících módů, což jsou obyčejné funkce definované příkazem `define-fun` s těmi omezeními, že jsou druhu `Bool`, a že všechny musejí přijímat shodný počet a druhy argumentů.

Příklady:

```
(declare-ode x ((dx true)))
(declare-ode y ((dy_1 mode_on) (dy_2 mode_off)))
```

`define-dt` slouží k definici varianty derivace funkce, která byla deklarována příkazem `declare-ode`:

```
(define-dt <name> <expr>)
```

s argumenty:

- `<name>` — název identifikátoru derivace funkce deklarovaného v příkazu `declare-ode`,
- `<expr>` — výraz definující tvar derivace funkce, (ne nutně) obsahující identifikátor funkce z `declare-ode` a nezávislou proměnnou `t` [30].

Příklady:

```
(define-dt dx 1)
(define-dt dy_1 (- (* (/ 3 t) y) 2))
```

Tyto identifikátory je nutné používat výhradně uvnitř ODE příkazů.

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

`int-ode` slouží k nastavení argumentů jednotlivých kroků integrace:

```
(int-ode <fun> (<t_1> <t_2>) (<fun_1> <fun_2>) (<mode_arg>*))
```

s argumenty:

- `<fun>` — identifikátor neznámé funkce vzniklý příkazem `declare-ode`,
- `(<t_1> <t_2>)` — počáteční a koncová hodnota nezávislé proměnné `t`,
- `(<fun_1> <fun_2>)` — počáteční hodnota funkce `<fun>` (tj. v bodě `<t_1>`) a identifikátor koncové hodnoty funkce, do níž bude umístěn výsledek integrace,
- `(<mode_arg>*)` — argumenty předané módu deklarovaného v `declare-ode`, tj. hodnoty, které rozhodnou o použité variantě derivace.

Příklady:

```
(int-ode x (t_0 t_1) (x_0 x_1) ())
(int-ode y (t_2 t_3) (y_2 y_3) (on_2))
```

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

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

/ Takto globálně, nebo pro každou ODE zvlášť? */*

3.1.6 Struktura a použití jazyka – verze 2

V této podsekci je uvedena doporučená struktura vstupu, aby 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 dynamicky měnil. Jelikož je vstup statický, je pro modelování průběhu nutné využít mnoha konstant o předem známém počtu, tzn. že už při modelování systému je nutné určit *počet kroků simulace*[31]. Kroky jsou určeny různými hodnotami konstant nezávislé proměnné `t`, mezi nimiž dochází k integraci. V těchto krocích také dochází se skokům.

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 použité ve všech krocích simulace musejí být deklarovány a konstanty počátečního kroku musí být i definovány. Každé konstantě je vhodné dát jako příponu číslo kroku, kterého se týká. Je vhodné deklarovat konstanty pro nezávislou proměnnou `t`, průběh neznámých funkcí a diskrétní stav.

Příklad:

```
(define-fun t0 () Real 0) (define-fun y0 () Real 1)
(define-fun run0 () Bool false)
```

[31] Pozor, jedná se o jiné kroky než kroky ODE řešiče.

```
(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)
(assert (and (= t_0 t0) (= y_0 y0) (= run_0 run0) ))
```

Nastavení kroků znamená definovat hodnoty konstant `t_i`. Nejjednodušším způsobem je zavedení konstatní délky kroku `T`, např.:

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

Deklarace a definice derivací funkcí se provádí pomocí příkazů `declare-ode`, `define-dt` a definováním funkcí pro módy. Příklad:

```
(declare-ode y ((dy_run mode_run) (dy_idle mode_idle)))
(define-dt dy_run     1 )
(define-dt dy_idle (- 1))
(define-fun mode_run  ((run Bool)) Bool run)
(define-fun mode_idle ((run Bool)) Bool (not run))
```

Integrace se provádí příkazem `int-ode`. Dochází tím k propojení konkrétních konstant reprezentující diskrétní stav s konstantami průběhu derivace. Výsledek každé integrace je směřován do argumentu příkazu, takže se nepoužívají žádné aserce[32].

Příklad:

```
(int-ode y (t_0 t_1) (y_0 y_1) (run_0))
(int-ode y (t_1 t_2) (y_1 y_2) (run_1))
```

Definice skoků. Skoky, tj. změny diskrétního stavu, lze snadno definovat pomocí asercí mezi sousedními stavy.

Doporučujeme zkonstruovat pomocnou funkci `jump`, př.:

```
(define-fun jump ((m1 Bool) (m2 Bool) (y2 Real)) Bool
    (or (and      m1       m2  (<  y2 bound_1))
        (and      m1  (not m2) (>= y2 bound_1))
        (and (not m1) (not m2) (>  y2 bound_2))
        (and (not m1)      m2  (<= y2 bound_2))
))
```

Příklad s použitím funkce `jump`:

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

[32] Aserce se používají až interně v přeloženém vstupu obsahujícím jen příkazy SMT-LIB.

Kapitola 4

Realizace

Kapitola 5

Testy a výsledky

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. 20

API Application programming interface. 14

ASCII American Standard Code for Information Interchange. 26

BDF Backward differentiation formula. 18

BMC Bounded Model Checking. 4, 12, 23

CNF Conjunctive normal form. 11

CTL Computation tree logic. 4

CVC Cooperating Validity Checker. 14, 15

DAE Differential-algebraic equation. 21

DIMACS Center for Discrete Mathematics and Theoretical Computer Science. 11

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

FOL First-order logic. 5, 6, 13

IVP Initial value problem. 8, 16, 21

LTL Linear time logic. 4

MPI Message Passing Interface. 20

NP Nondeterministic polynomial time. 3, 4

ODE Ordinary differential equation. vii, ix, 2, 4, 7–9, 11, 15–17, 19–23, 25, 28, 29, 31, 32

OpenMP Open Multi-Processing. 20

QBF Quantified Boolean formulas. 4

SAT Boolean satisfiability problem. vii, ix, 2–4, 9, 11–14, 22, 28

SIMD Single instruction multiple data. 21

SMT Satisfiability Modulo Theories. vii, ix, 2, 4, 9, 11–15, 22, 23, 25, 27, 29, 32, 33

SUNDIALS SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers. 20, 21

TMP Template Metaprogramming. 21