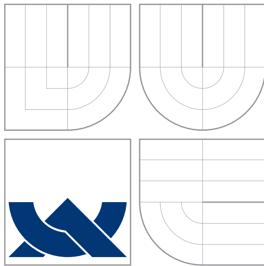
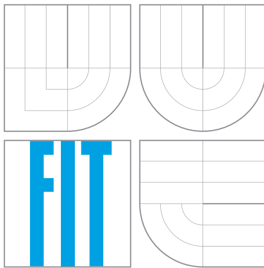


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁVRH A IMPLEMENTACE NÁSTROJE PRO FORMÁLNÍ VERIFIKACI SYSTÉMŮ SPECIFIKOVANÝCH JAZYKEM RT LOGIKY

DESIGN AND IMPLEMENTATION OF A TOOL FOR FORMAL VERIFICATION OF SYSTEMS SPECIFIED IN RT-LOGIC LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. JAN FIEDOR

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. JOSEF STRNADEL, Ph.D.

BRNO 2009

Zadání práce

1. Seznamte se s principy formální specifikace a verifikace; detailněji prostudujte zejména metody verifikace předpokládající specifikaci v jazyce logiky pracující s reálným časem (RT logiky, RTL).

2. Po vzájemné dohodě s Bc. Markem Gachem navrhněte blokové schéma a rozhraní nástroje schopného verifikovat vlastnosti systému specifikovaného jazykem RT logiky.

3. Po dohodě s vedoucím vytvořte v přirozeném jazyce neformální slovní specifikaci několika jednoduchých systémů pracujících v reálném čase.

4. Systémy specifikované v předchozím bodě specifikujte formálně pomocí jazyka RT logiky.

5. Nástroj (blokově navržený v bodě 2) implementujte a jeho funkčnost ověřte verifikací vhodně vybraných vlastností systémů specifikovaných jazykem RT logiky. Zvažte implementaci metod vedoucích k redukci stavového prostoru.

6. Shrňte dosažené výsledky a navrhněte možná rozšíření Vaší práce.

Abstrakt

Protože komplexnost systémů pořád roste a s tím také riziko výskytu chyb, je potřeba tyto chyby efektivně a spolehlivě opravovat. U řady systémů reálného času tato potřeba platí dvojnásob, jelikož byť jediná chyba může způsobit jejich úplné zhroucení, které může mít katastrofální důsledky. Formální verifikace, na rozdíl od jiných metod, umožňuje spolehlivé ověřování požadavků kladených na určitý systém.

Klíčová slova

Formální verifikace, Logika reálného času, RTL, QF_UFIDL, graf omezení, DFS, návrhový vzor, CORBA, ANTLR

Abstract

As systems complexity grows, so grows the risk of errors, that's why it's necessary to effectively and reliably repair those errors. With most of real-time systems this statement pays twice, because a single error can cause complete system crash which may result in catastrophe. Formal verification, contrary to other methods, allows reliable system requirements verification.

Keywords

Formal verification, Real-Time Logic, RTL, QF_UFIDL, constraint graph, DFS, design pattern, CORBA, ANTLR

Citace

Jan Fiedor: Návrh a implementace nástroje pro formální verifikaci systémů specifikovaných jazykem RT logiky, diplomová práce, Brno, FIT VUT v Brně, 2009

Návrh a implementace nástroje pro formální verifikaci systémů specifikovaných jazykem RT logiky

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D.

. . . .
Jan Fiedor
24. května 2009

© Jan Fiedor, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 2

1 Úvod 3

2 Systémy reálného času [5] 4

3 Verifikace 5
3.1 Od testování k verifikaci 5
3.2 Obecné metody verifikace 5
3.2.1 Symbolické logiky [1] 5
3.2.2 Automaty a jazyky 9

4 Verifikace systémů reálného času [2] 11
4.1 Specifikace a bezpečnostní tvrzení 11
4.2 Model typu událost-akce 12
4.3 Logika reálného času (RTL) 12
4.4 Omezené RTL formule 14
4.5 Graf omezení 16
4.6 Ověření nesplnitelnosti 17
4.7 Prohledávací strom 18
4.8 Optimalizace 20

5 Návrh nástroje pro verifikaci 23

6 Realizace nástroje pro verifikaci 25
6.1 Implementační prostředí 25
6.2 Modularita 26
6.3 Verifikátor 26
6.3.1 Jádro 26
6.3.2 Proxy jádra 27
6.3.3 Plánovač jádra 27
6.3.4 Událostní subsystém 27
6.3.5 Subsystém rozšíření 28
6.3.6 Protokolovací subsystém 30
6.3.7 Konfigurační subsystém 31
6.3.8 Komunikační subsystém 32
6.3.9 Modul pro zpracování omezených RTL formulí 33
6.3.10 Modul pro konverzi omezených RTL formulí 37

6.3.11 Modul pro tvorbu grafu omezení 41
6.3.12 Modul pro tvorbu prohledávacího stromu 44
6.4 Grafické uživatelské rozhraní 46
6.4.1 Komponenty 46
6.4.2 Jádro 47
6.4.3 Proxy jádra 48
6.4.4 Plánovač jádra 48
6.4.5 Událostní subsystém 48
6.4.6 Subsystém rozšíření 48
6.4.7 Protokolovací subsystém 49
6.4.8 Konfigurační subsystém 49
6.4.9 Komunikační subsystém 50
6.4.10 Lokalizační subsystém 50
6.5 Komunikace 51
6.5.1 Implementační požadavky 51
6.5.2 Přenos událostí 52
6.5.3 Konkrétní implementace 53

7 Závěr a zhodnocení 55
7.1 Směry dalšího vývoje 55
7.2 Závěr 55

Použitá literatura 56

Kapitola 1

Úvod

Systémy reálného času lze najít ve velkém množství zařízení, od domácích spotřebičů, přes elektroniku a dopravní prostředky, až například po vesmírné sondy. Protože komplexnost těchto systémů pořád roste a s tím také riziko výskytu chyb, je potřeba tyto chyby efektivně a spolehlivě opravovat. Než může být ovšem nějaká chyba opravena, musí být nejdříve vůbec odhalena. Staré způsoby hledání chyb, jako kontrola implementace systému jiným člověkem, jsou při rozsáhlosti dnešních systémů velice neefektivní a je tedy potřeba uplatnit automatizovatelné metody pro hledání a případně i opravu chyb.

Formální verifikace umožňuje spolehlivé ověřování požadavků kladených na určitý systém. Samozřejmě tato vysoká spolehlivost a efektivita z hlediska odhalování chyb je vykoupena řadou problémů. Formální verifikace vyžaduje pro svou činnost určitou formu popisu specifikace systému a ověřovaných vlastností, ať již je to popis pomocí logických formulí, automatů, speciálního programovacího jazyka, petriho sítí nebo třeba procesní algebry. Použitý popis také přímo ovlivňuje možnosti ověřování, určitá forma popisu může být velice vhodná pro ověřování specifické vlastnosti systému, ale zároveň také nepoužitelná pro ověřování vlastnost jiné. Dalším problémem je složitost výpočtu, ať již z hlediska časové či prostorové náročnosti. Používané metody mívají často exponenciální složitost. Často se tedy musí u složitějších systémů přistupovat k abstrakci pro zjednodušení složitosti řešených problémů.

Kapitola 2

Systémy reálného času [5]

Hlavním rozdílem mezi standardními systémy a systémy reálného času je v požadavcích na korektnost těchto systémů. Hlavním požadavkem u standardních systémů je samozřejmě správnost výstupů. U systémů reálného času je ovšem situace složitější, neboť kromě správnosti výstupů je důležitý také čas, kdy byly tyto výstupy vyprodukovány.

Systém reálného času by mohl být tedy definován jako systém, který musí reagovat správně a *včas* na vstupní podněty. Je důležité si uvědomit, že termín *včas* je relativní, nemusí vždy znamenat rychle, vyjadřuje pouze nutnost dodržení určitých časových mezí, aby byla splňena požadovaná doba odezvy daného systému.

Systémy reálného času lze dělit několika způsoby. Spíše než rozdělení podle požadavků na dobu odezvy se ale využívá dělení podle charakteru selhání systému, tedy jaké budou důsledky nedodržení požadovaných časových mezí. Zde se využívá rozdělení do tří kategorií:

1. **Soft** RT systém - nedodržení časového omezení má za následek pouze degradování výkonu systému, ale nezpůsobí jeho selhání.

2. **Firm** RT systém - nedodržení několika časových omezení nevede k úplnému selhání systému, ovšem vetší počet nedodržených časových omezení může vést k selhání systému nebo dokonce katastrofě.

3. **Hard** RT systém - nedodržení jediného časového omezení má za následek kompletní selhání systému, které může vyústit v katastrofu.

Kapitola 3

Verifikace

3.1 Od testování k verifikaci

Testování je pravděpodobně jednou z nejstarších technik pro odhalování chyb v softwaru nebo hardwaru. Jde o spuštění testovaného systému s využitím konečné množiny vstupů a následné ověření, zda získané výstupy nebo chování systému odpovídá jeho specifikaci. Na rozdíl od standardních systémů je u systémů pracujících v reálném čase kromě hodnot vstupů důležitý také čas, kdy byly tyto vstupy systému předloženy. Stejně tak u ověřování je, kromě korektnosti výstupů, důležitý také čas, kdy byly vyprodukovány. Protože u větších a složitějších systémů je počet možných vstupů obrovský, někdy i nekonečný, je tato technika pro celkové ověření vlastností systému prakticky nepoužitelná.

Simulace, narozdíl od testování, pracuje pouze s modelem existujícího systému. Výhodou využití modelu je možnost provádění testů, které by byly u reálného systému příliš nebezpečné, jako například testování řízení jaderné elektrárny, nebo dokonce nemožné. Pro jeden reálný systém lze vytvořit libovolný počet jeho modelů s různými úrovněmi abstrakce. To umožňuje při testování ignorovat irelevantní části systému, které na ověřování testované vlastnosti nemají žádný vliv, a urychlit tím celkovou simulaci. Nevýhodou simulace je, že není možné namodelovat všechny sekvence událostí, které mohou v reálném modelu nastat.

Předchozí dvě techniky jsou dobré pro odhalování chyb v simulovaném nebo reálném systému, ale většinou nemohou garantovat, že systém splňuje požadavky, které jsou na něj kladeny. Verifikace dokáže zjistit, zda je ověřovaná vlastnost vždy platná nebo zda může dojít k jejímu porušení. Nevýhodou verifikace je, že vyžaduje pro svou činnost popis systému a požadavků kladených na tento systém pomocí jazyka, vhodného pro verifikaci.

3.2 Obecné metody verifikace

3.2.1 Symbolické logiky [1]

Symbolická logika je kolekce jazyků, které používají symboly pro reprezentaci faktů, událostí a akcí, a poskytují pravidla umožnující usuzování nad těmito symboly. Pokud je k dispozici specifikace systému a jeho požadovaných vlastností ve formě logických formulí, je možné dokázat, že tyto vlastnosti jsou logickým důsledkem specifikace tohoto systému. Mezi nejpoužívanější logiky pro popis a následnou verifikaci systémů patří výroková logika a predikátová logika.

3.2.1.1 Výroková logika

Základními prvky výrokové logiky jsou pravdivostní symboly $true$ a $false$ a výrokové proměnné, kterých existuje obecně nekonečný počet. Kombinací těchto prvků vznikají výroky, označované jako **formule**, jazyka výrokové logiky.

Buď P konečná neprázdná množina výrokových proměnných. Pak p je **atom** pokud $p \in P$ nebo p je jeden z pravdivostních symbolů $true$ a $false$. **Literál** l je atom p nebo jeho negace $\neg p$. **Formule** F je buď samostatný literál l nebo vznikne aplikací některé z následujících spojek na formule F_1 a F_2:

- Negace $\neg F_1$ nebo $\neg F_2$
- Konjunkce $F_1 \wedge F_2$
- Disjunkce $F_1 \vee F_2$
- Implikace $F_1 \rightarrow F_2$
- Ekvivalence $F_1 \leftrightarrow F_2$

Interpretace I přiřazuje každé výrokové proměnné právě jednu pravdivostní hodnotu $true$ nebo $false$ a vyjadřuje tak význam formule složené z těchto proměnných. Pokud je dána formule F a jí odpovídající interpretace I, lze vypočítat výslednou pravdivostní hodnotu této formule. Nejjednodušší metodou pro určení pravdivostní hodnoty formule F je využití pravdivostní tabulky. Pravdivostní tabulka 3.1 shrnuje pravdivostní hodnoty formulí vzniklých aplikací jednotlivých spojek výrokové logiky. Hodnota 1 odpovídá pravdivostní hodnotě $true$, hodnota 0 pravdivostní hodnotě $false$.

F_1	F_2	$\neg F_1$	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Tabulka 3.1: Pravdivostní tabulka pro jednoduché formule výrokové logiky

Formule F je **splnitelná**, právě tehdy když existuje interpretace I, pro kterou formule F nabývá hodnoty $true$, tedy formule F je při daném ohodnocení proměnných pravdivá. Poté se říká, že interpretace I je modelem formule F neboli $I \models F$. Formule F je **validní**, právě tehdy když pro všechny interpretace I platí $I \models F$. Je důležité si uvědomit, že splnitelnost a validita jsou tzv. duální pojmy, tedy ověřování jedné vlastnosti lze jednoduše převést na ověřování druhé. F je validní, právě tehdy když $\neg F$ je nesplnitelná. Díky této dualitě je možné vždy ověřovat pouze jednu z daných vlastností, tedy validitu nebo splnitelnost, podle toho, které ověření je z hlediska použitého postupu výhodnější.

Pro určení, zda formule popisující požadovanou vlastnost systému je logickým důsledkem formulí specifikujících daný systém lze využít principu rezoluce. Rezoluce vyžaduje převedení formule do konjunktní normální formy (CNF). CNF formule je tvořena konjunkcí klauzulí, kde každá klauzule je disjunkcí literálů. Rezoluční princip poté říká, že pokud existují ve formuli dvě klauzule C_1 a C_2, kde $l_1 \in C_1$, $l_2 \in C_2$ a $l_1 \wedge l_2$ nabývá hodnoty $false$, tedy platí, že literál l_1 je negací literálu l_2, pak logickým důsledkem, tzv. rezolventou,

klauzulí C_1 a C_2 je klauzule vzniklá disjunkcí klauzulí C_1 a C_2 po odebrání literálů l_1 a l_2 z těchto klauzulí. Formálně lze tento pricip zapsat následovně:

$$\frac{a_1 \vee \ldots \vee a_i \vee \ldots \vee a_n,\ b_1 \vee \ldots \vee b_j \vee \ldots \vee b_m}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_m}$$

kde a_i and b_j jsou literály $\forall i \in 1..n$, $\forall j \in 1..m$, a_i je negací b_j a oddělující přímka vyjadřuje **je logickým důsledkem**.

Pro ověření validity testované vlastnosti systému lze poté využít rezolučního teorému, který říká, že množina klauzulí S je nesplnitelná, právě tehdy když je odvozena prázdná klauzule z této množiny S. Pokud tedy jakoukoliv aplikací rezolučního principu nedojde k odvození prázdné klauzule, je výsledná množina klauzulí, a tedy i formule, kterou tyto klauzule tvoří, validní.

Tento postup využívá řada nástrojů pro ověřování splnitelnosti nebo validity formulí výrokové logiky. Nevýhodou je exponeciální složitost řešení tohoto problému, ikdyž existují efektivní reprezentace a heuristiky pro snížení jak prostorové, tak časové složitosti řešení.

3.2.1.2 Predikátová logika

Predikátová logika je logikou 1. řádu. Má větší vyjadřovací sílu a je tedy vhodnější pro usuzování nad určitým výpočtem. Rozšiřuje výrokovou logiku o predikáty, funkce a kvantifikátory.

Základním prvkem jazyka je **term**. Nejjednoduššími termy jsou **proměnné** a **konstanty**, složitější termy jsou vytvářeny pomocí **funkcí**. N-ární funkce f přijímá n termů jako své argumenty, na konstanty lze také nahlížet jako na 0-ární funkce.

Výrokové proměnné jsou zastoupeny ve formě **predikátů**. N-ární predikát p přijímá n termů jako své argumenty, 0-ární predikát poté odpovídá výrokové proměnné známé z výrokové logiky. **Atom** je pravdivostní hodnota $true$ nebo $false$, nebo n-ární predikát aplikovaný na n termů. **Literál** je atom nebo jeho negace. **Formule** predikátové logiky je samostatný literál, nebo vznikne aplikací logické spojky $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ na jednu nebo více formulí nebo aplikací kvantifikátoru na danou formuli. V predikátové logice existují dva typy kvalifikátorů:

- Univerzální kvantifikátor $\forall x.F[x]$ vyjadřující, že pro všechny x platí formule F.
- Existenční kvantifikátor $\exists x.F[x]$ vyjadřující, že existuje nějaké x, pro které platí formule F.

Proměnná x se poté označuje jako kvantifikovaná proměnná a $F[x]$ jako rozsah platnosti kvantifikátoru. Proměnná x se označuje jako **vázaná** ve formuli $F[x]$, pokud se x vyskytuje v rozsahu platnosti kvantifikátoru $\forall$ nebo $\exists$. Proměnná x se označuje jako **volná** ve formuli $F[x]$, pokud se ve formuli vyskytuje instance x, která není vázaná žádným kvantifikátorem. Pokud formule neobsahuje žádnou volnou proměnnou, pak se označuje jako **uzavřená**.

Stejně jako u výrokové logiky i zde mohou formule nabývat ohodnocení $true$ nebo $false$. U predikátové logiky je ovšem situace trochu složitější, protože ohodnocení termů může být kromě pravdivostních hodnot téměř jakékoliv, od celých čísel až například po jména lidí.

Interpretace I přiřazuje termům hodnoty z domény D_I. **Doména** D_I interpretace I je množina hodnot nebo objektů, jako celá čísla, reálná čísla, auta, lidi, nebo pouze abstraktní objekty, kterých mohou nabývat jednotlivé termy. $|D_I|$ vyjadřuje kardinalitu množity, tedy celkový počet těchto hodnot. Doména může být konečná, spočetně nekonečná

nebo nespočetně nekonečná, vždy ale musí být neprázdná. Přiřazení α_I interpretace I poté mapuje symboly proměnných, konstant, funkcí a predikátů na prvky, funkce a predikáty domény D_I:

- Každému symbolu proměnné x je přiřazena hodnota x_I z domény D_I
- Každému symbolu n-ární funkce f je je přiřazena n-ární funkce

$$f_I : D_I^n \to D_I$$

  která mapuje n prvků domény D_I na jediný prvek této domény

- Každé konstantě (0-ární funkční symbol) je přirazena hodnota z domény D_I
- Každému symbolu n-árního predikátu p je přižazen n-ární predikát

$$p_I : D_I^n \to \{true, false\}$$

  který mapuje n prvků domény D_I na pravdivostní hodnotu $true$ nebo $false$

- Každé výrokové proměnné (0-ární predikátový symbol) je přiřazena pravdivostní hodnota $true$ nebo $false$

Interpretace $I : (D_I, \alpha_I)$ je tedy dvojicí skládající se z domény hodnot, kterých mohou symboly jazyka nabývat, a přiřazení, které mapuje jednotlivé symboly na tyto hodnoty.

Při přítomnosti kvantifikátorů se situace trochu komplikuje. Nejprve je potřeba definovat x-variantu interpretace $I : (D_I, \alpha_I)$ jako interpretaci $J : (D_J, \alpha_J)$, takovou, že platí následující:

- $D_I = D_J$
- $\alpha_I[y] = \alpha_J[y]$ pro všechny konstanty, volné proměnné, funkce a predikátové symboly y kromě x

tedy interpretace I a J jsou stejné až na ohodnocení proměnné x. Zápis $J : I \triangleleft \{x \mapsto v\}$ poté říká, že J je x-variantou I, kde $\alpha_J[x] = v$ pro nějakou hodnotu $v \in D_I$. Pak platí, že:

- $I \models \forall x.F$ právě tehdy když $\forall v \in D_I$, $I \triangleleft \{x \mapsto v\} \models F$
- $I \models \exists x.F$ právě tehdy když $\exists v \in D_I$ takové, že $I \triangleleft \{x \mapsto v\} \models F$

neboli I je interpretací $\forall x.F$, právě tehdy když všechny její x-varianty jsou interpretací F. A I je interpretací $\exists x.F$, právě tehdy když nějaká x-varianta I je interpretací F.

Pro snažší strojovou manipulaci a analýzu formulí se často provádí úprava těchto formulí do specifického tvaru. Jedním z často využívaných tvarů je tzv. prenexní normální forma, ve které jsou všechny kvantifikátory přesunuty na levou stranu, tedy začátek, formule. Formálně řečeno, formule F je v prenexní normální formě, právě tehdy když je ve tvaru

$$(Q_1v_1)(Q_2v_2)\dots(Q_{n-1}v_{n-1})(Q_nv_n)(M)$$

kde každý výraz (Q_iv_i), pro $i = 1..n$, je buď $(\forall v_i)$ nebo $(\exists v_i)$ a M je formule bez jakýchkoliv kvantifikátorů. $(Q_1v_1)\dots(Q_nv_n)$ se označuje jako prefix a M jako matice formule F. Matice M může být poté upravena do konjunktní normální formy (CNF), která obsahuje pouze konjunkce disjunkcí literálů.

Prenexní CNF může být dále upravena do Skolemovy standardní formy procesem označovaným **skolemizace**. Při skolemizaci dojde k nahrazení existenčních kvantifikátorů Skolemovými funkcemi. Absence existenčních kvantifikátorů často usnadňuje proces ověřování platnosti a validity formulí a také následně umožňuje velice jednoduše odstranit i univerzální kvantifikátory.

Nechť Q_i je existenční kvantifikátor v prefixu $(Q_1 v_1) \dots (Q_n v_n)$, pro $i = 1..n$. Pokud se nalevo od Q_i nevyskytuje žádný unverzální kvantifikátor, pak lze nahradit všechny výrazy v_i v matici M novou konstantou c, která se ještě nevyskytuje v matici M, a odstranit výraz $(Q_i v_i)$ z prefixu. Jestliže $(Q_{u_1} \dots Q_{u_m})$, pro $1 \leq u_1 < \dots < u_m < i$, jsou univerzální kvantifikátory nalevo od Q_i, pak se všechny výrazy v_i nahradí novou m-ární funkcí $f(v_{u_1}, \dots, v_{u_m})$, která se ještě nevyskytuje v matici M, a $(Q_i v_i)$ se odstraní z prefixu. Výsledná formule bez existenčních kvantifikátorů je poté ve Skolemově normální formě. Konstanty a funkce použité při skolemizaci se označují jako Skolemovy konstanty a Skolemovy funkce.

Další, často používanou, operací při zpracování a ověřování formulí je **substituce**. Substituce je syntaktická operace nad formulí, která může výrazně ovlivnit její sémantiku, tedy význam. Umožňuje ověřovat validitu celé množiny formulí pomocí šablon formulí. Je také jeden ze základních nástrojů pro manipulaci s formulemi při jejich ověřování.

Obecně je substituce σ mapování z formule na formuli

$$\sigma : \{F_1 \mapsto G_1, F_2 \mapsto G_2, \dots, F_{n-1} \mapsto G_{n-1}, F_n \mapsto G_n\}$$

kde doména substituce σ, $domain(\sigma)$ je

$$domain(\sigma) : \{F_1, F_2, \dots, F_{n-1}, F_n\}$$

a rozsah substituce σ, $range(\sigma)$ je

$$range(\sigma) : \{G_1, G_2, \dots, G_{n-1}, G_n\}$$

Aplikace substituce σ na formuli F, $F\sigma$, nahradí každý výskyt formule F_i z domény σ odpovídající formulí G_i z rozsahu σ. Při manipulacích s formulemi se většinou nenahrazují celé formule, ale pouze jednotlivé termy, převážně proměnné, za jiné proměnné, konstanty nebo funkce. Dříve zmíněná skolemizace je také určitou formou substituce.

Narozdíl od výrokové logiky se u predikátové logiky vetšinou nepoužívají obecné metody pro ověřování validity a splnitelnosti. Ty jsou často velice neefektivní nebo dokonce nepoužitelné. Sémantika formulí predikátové logiky je charakterizována pomocí **teorií**. Ty určují, z jakých symbolů se dané formule mohou skládat a jaké axiomy pro dané formule musí platit. Na základě použité teorie se poté volí vhodná metoda pro ověření platnosti daných formulí.

3.2.2 Automaty a jazyky

Automat je schopen rozhodnout, zda sekvence slov patří do specifického jazyka. Tento jazyk se skládá z množiny slov nad nějakou konečnou abecedou. Pokud tato sekvence slov odpovídá sekvencí událostí a akcí, je možné sestrojit automat, který bude přijímat pouze správné sekvence těchto událostí a akcí z hlediska určitého systému a tímto bude řešit verifikaci zadaného problému v tomto systému.

Automaty mohou reprezentovat procesy nebo celkový systém. Přesněji specifikační automat bude reprezentovat požadovanou specifikaci systému a implementační automat bude

modelovat implementaci, která se snaží splnit požadovanou specifikaci. Cílem je poté zjistit, zda implementace splňuje zadanou specifikaci. Na tento problém lze nahlížet jako na problém inkluze jazyků, tedy jde o zjištění, zda jazyk, který je přijímán implementačním automatem je podmnožinou jazyka přijímaného specifikačním automatem.

Kapitola 4

Verifikace systémů reálného času [2]

Existují dvě formy specifikace systémů reálného času, první zachycuje strukturu a funkcionalitu systému, druhá pak pouze jeho chování.

Strukturní a funkcionální popis zahrnuje specifikaci mechanických, elektrických a elektrotechnických komponent systému. Ukazuje jak jednotlivé komponenty pracují a jaké funkce a operace poskytují. Popisuje také propojení jednotlivých komponent a jak akce jedné komponenty ovlivňují ostatní komponenty systému.

Popis chování specifikuje odezvu systému na různé akce a události. Ukazuje tedy pouze jak jednotlivé komponenty systému reagují na interní a externí události, ne jak takovýto systém sestrojit. Vzhledem k tomu, že u systémů reálného času jsou zajímavé hlavně časové vlastnosti, popis chování bez zbytečně složité strukturní specifikace často postačuje pro verifikaci splnitelnosti většiny časových omezení kladených na daný systém. Kromě toho redukcí komplexnosti specifikace a analýzy lze omezit použité specifikační jazyky tak, aby se zabývaly pouze časovými závislostmi.

4.1 Specifikace a bezpečnostní tvrzení

Aby bylo možné dokázat, že systém splňuje určitá bezpečnostní kritéria, je potřeba identifikovat vztah specifikace systému k bezpečnostnímu tvrzení, které reprezentuje požadovanou vlastnost systému. Samozřejmě se předpokládá, že implementace systému odpovídá předložené specifikaci. Ikdyž specifikace pomocí popisu chování neukazuje, jak daný systém zkonstruovat, není velký problém dokázat, že systém implementovaný na základě nějaké strukturní a funkční specifikace splňuje požadované chování.

Jeden z následujících tří případů může být výsledkem analýzy vztahu mezi specifikací a bezpečnostním tvrzením:

1. Bezpečnostní tvrzení je teorém odvoditelný ze specifikace, takže systém je bezpečný vzhledem k chování popsanému bezpečnostním tvrzením.
2. Bezpečnostní tvrzení je nesplnitelné vzhledem ke specifikaci, takže systém je určitě nebezpečný, protože specifikace způsobí porušení bezpečnostního tvrzení.
3. Bezpečnostní tvrzení je splnitelné za určitých podmínek, je tedy nutné přidat další omezení systému pro zaručení bezpečnosti.

Specifikace a bezpečnostní tvrzení mohou být zapsané pomocí jednoho z více specifikačních jazyků reálného času. Volba jazyka určuje, jaké algoritmy mohou být použity pro analýzu a verifikaci.

4.2 Model typu událost-akce

Model typu událost-akce zachycuje datové závislosti a časové uspořádání výpočetních akcí, které musí být provedeny jako odezvy na události v systémech reálného času. Skládá se ze čtyř základních prvků:

- **Akce** - plánovatelná jednotka práce, může být jednoduchá nebo složená. Jednoduché akce jsou atomické, nemohou nebo nemusí být rozděleny na několik jednodušších akcí pro potřeby analýzy. Jejich vykonání spotřebuje omezené množství času. Složené akce jsou částečně uspořádané jednoduché nebo složené akce. Stejná akce se může vyskytovat v jedné složené akci i vícekrát. Rekurzivní akce nebo cyklicky řetězené akce, kde jedna akce je podakcí svého předchůdce v daném řetězu úloh, nejsou povoleny. Sekvenční provedení dvou akcí se označuje zápisem $A; B$ a vyjadřuje, že akce A je následována akcí B. Paralelní provedení dvou akcí je označuje zápisem $A \parallel B$ a vyjadřuje, že akce A je provedena souběžně s akcí B.
- **Stavový predikát** - tvrzení o stavu specifikovaného systému.
- **Událost** - časová značka označující bod v čase, který je významný pro popis chování systému, existují čtyři typy událostí:
  - **Externí** událost - způsobena událostí mimo specifikovaný systém.
  - **Spouštěcí** událost - označuje počátek nějaké akce.
  - **Koncová** událost - označuje konec nějaké akce.
  - **Přechodová** událost - označuje změnu nějaké vlastnosti systému.
- **Časové omezení** - tvrzení o úplném časování událostí v systému.

4.3 Logika reálného času (RTL)

Zápis specifikace podle modelu typu událost-akce není ovšem vhodný, protože tento popis je obtížně zpracovatelný počítačem. Jako řešení totoho problému byla vytvořena logika reálného času (*real-time logic*) neboli RTL. RTL je logika 1. řádu speciálně rozšířená o možnosti zachycení časových požadavků specifikovaného systému a přitom umožňující jednoduché mechanické zpracování této specifikace.

RTL je založena na modelu typu událost-akce, ale je rozšířena o několik nových vlastností jako funkce výskytu @, která přiřazuje časové hodnoty výskytům událostí. Zápis $@(E, i) = x$ znamená, že k i-tému výskytu události E dojde v čase x. RTL rozlišuje celkem tři typy konstant - akce, události a celá čísla. Akce jsou definovány stejně jako v modelu typu událost-akce, aby se odlišily od proměnných používají se pro jejich zápis velká písmena. Subakce B_i složené akce A se poté označuje $A.B_i$. Události slouží jako časové značky a stejně jako u modelu typu událost-akce existují čtyři typy těchto událostí:

- Spouštěcí událost označuje začátek dané akce a je uvozena znakem $\uparrow$
- Koncová událost označuje konec dané akce a je uvozena znakem $\downarrow$
- Přechodová událost označuje změnu určité vlastnosti systému
- Externí událost je uvozena znakem Ω

Pro snažší pochopení předkládaných metod a postupů budou jednotlivé kroky verifikace ilustrovány na jednoduchém příkladu železničního přejezdu. Přejezd obsahuje pouze jedinou kolej, v určitý čas tedy může projíždět pouze jediný vlak. Celý systém se bude skládat z komponent vlaku, vlakového senzoru, ovladače závor a samotných závor. Úkolem ovladače závor je zajistit, aby v době průjezdu vlaku železničním přejezdem se na křížení cesty a kolejí nevyskytovalo žádné auto. Pro zjednodušení situace lze předpokládat, že tento úkol je vždy splněn, pokud jsou závory v době průjezdu vlaku sklopeny.

Specifikace systému (**SP**) v přirozeném jazyce vypadá následovně:

- Když se vlak přiblíží k vlakovému senzoru a je zachycen tímto senzorem, je zaslán signál ovladači závor, který začne pomalu sklápět závory před železničním přejezdem.
  - Závora bude sklopena do 30 sekund od doby, kdy se vlak přiblížil a byl zachycen senzorem.
  - Sklopení závory trvá alespoň 15 sekund.

Bezpečnostní tvrzení (**SA**) v přirozeném jazyce lze pak vyjádřit následovně:

- Pokud vlak potřebuje alespoň 45 sekund pro uražení cesty od senzoru k železničnímu přejezdu a průjezd vlaku je dokončen do 60 sekund od doby, kdy byl vlak zachycen senzorem, pak je zajištěno, že v době doražení vlaku k železničnímu přejezdu jsou závory sklopeny a vlak opustí železniční přejezd během 45 sekund od doby, kdy bylo dokončeno sklápění závor.

Pro potřeby počítačového zpracování a verifikace je nyní potřeba převést výše uvedenou specifikaci a bezpečnostní tvrzení na zápis v RTL.

Specifikace systému v RTL:

$$\forall x \ @(TrainApproach, x) \leq @(\uparrow Downgate, x) \wedge$$

$$@(\downarrow Downgate, x) \leq @(TrainApproach, x) + 30$$

$$\forall y \ @(\uparrow Downgate, y) + 15 \leq @(\downarrow Downgate, y)$$

Bezpečnostní tvrzení v RTL:

$$\forall t \forall u \ @(TrainApproach, t) + 45 \leq @(Crossing, u) \wedge$$

$$@(Crossing, u) < @(TrainApproach, t) + 60 \rightarrow$$

$$@(\downarrow Downgate, t) \leq @(Crossing, u) \wedge$$

$$@(Crossing, u) \leq @(\downarrow Downgate, t) + 45$$

Pro dokázání, že bezpečnostní tvrzení je teorém odvoditelný ze specifikace systému, lze použít existující metody pro dokazování teorémů. Ovšem zápis v jazyce RTL není pro tyto metody příliš vhodný. Proto je potřeba nejprve převést RTL formule výše do diferenční logiky, která je mnohem vhodnější a převod mezi těmito logikami je jednoduchý a dobře automatizovatelný. Cílovou logikou zde bude celočíselná diferenční logika (*IDL*) s neinterpretovanými funkcemi.

Specifikace systému v diferenční logice:

$$\forall x \ f(x) \leq g_1(x) \wedge g_2(x) \leq f(x) + 30$$

$$\forall y \; g_1(y) + 15 \leq g_2(y)$$

Bezpečnostní tvrzení v diferenční logice:

$$\forall t \forall u \; f(t) + 45 \leq h(u) \land h(u) < f(t) + 60 \rightarrow g_2(t) \leq h(u) \land h(u) \leq g_2(t) + 45$$

V těchto formulích, t, u, x a y jsou proměnné a f, g_1, g_2 a h jsou neinterpretované celočíselné funkce. f odpovídá funkci výskytu události *TrainApproach*, g_1 odpovídá funkci výskytu označující počátek akce *Downgate*, g_2 odpovídá funkci výskytu označující konec akce *Downgate* a h odpovídá funci výskytu události *Crossing*.

Problém zjištění, zda bezpečnostní tvrzení vyplývá ze specifikace systému je obecně nerozhodnutelný pro celkovou množinu RTL formulí, takže né vždy lze najít řešení. Pro specifickou podmnožinu RTL formulí je ale tento problém rozhodnutelný, ovšem řešení má exponenciální složitost.

Existuje několik způsobů, jak lze zvýšit efektivitu řešení. Prvním způsobem je použití aproximace pro získání jednodušší množiny specifikace systému a bezpečnostích tvrzení. Druhým způsobem je zaměření analýzy pouze na část specifikace systému, která se týká nebo může ovlivnit platnost bezpečnostního tvrzení. Třetím způsobem je omezení specifikačního jazyka tak, že bude méně obecný, ale může být použita efektivnější analýza.

4.4 Omezené RTL formule

Jedna třída omezených RTL formulí vychází ze skutečnosti, že ve specifikaci mnoha systémů reálného času:

1. se RTL formule skládají z aritmetických nerovností zahrnujících dva termy a celočíselnou konstantu, kde term může být buď proměnná nebo funkce

2. RTL formule neobsahují aritmetické výrazy obsahující funkce, které berou jako argument instanci sebe sama

Takováto třída omezených RTL formulí umožňuje potenciální využití přístupů známých z teorie grafů pro jejich analýzu. Například lze využít algoritmus hledání nejkratší cesty z určitého bodu do všech ostatních a řešit jednoduchý problém celočíselného programování, kde každá nerovnost je ve tvaru $x_i - x_j \leq \pm a_{ij}$ kde x_i a x_j jsou proměnné a a_{ij} je celočíselná konstanta. Nebo lze využít grafu omezení pro reprezentaci množiny nerovností. Každá proměnná bude poté reprezentovat uzel v grafu a nerovnost $x_i \pm a_{ij} \leq x_j$ bude reprezentovat orientovanou hranu s váhou a_{ij} jdoucí z uzlu x_i do uzlu x_j. Poté množina nerovností reprezentována takovýmto grafem je nesplnitelná, právě tehdy když existuje v grafu cyklus kladnou celkovou váhou.

Na základě předchozích pozorování je tedy potřeba omezit RTL formule tak, aby obsahovaly aritmetické nerovnosti v následující formě:

$$funkce\ výskytu \pm celočíselná\ konstanta \leq funkce\ výskytu$$

kdy formule $@(E_1, i) \pm I < @(E_2, j)$ lze zapsat jako $@(E_1, i) \pm I + 1 \leq @(E_2, j)$ a formule $\neg(@(E_1, i) \pm I \leq @(E_2, j))$ lze zapsat jako $@(E_2, j) \pm I + 1 \leq @(E_1, i)$.

Všechny formule, které se vyskytují v dříve uvedeném příkladě, splňují výše popsaná omezení a patří tedy do definované třídy omezených RTL formulí. Ovšem například formule

$$\forall t \exists u \; @(TrainApproach, t) + u \leq @(Crossing, t)$$

která se nevyskytuje v předchozím příkladě, nepatří do této třídy omezených RTL formulí, protože první argument nerovnosti $\leq$ je sumou funkce a proměnné.

Aby bylo možné sestrojit graf omezení, který umožní analýzu řešeného problému, je nejprve potřeba transformovat RTL formuli F na odpovídající formuli F' v diferenční logice[1]. Každá funkce výskytu $@(E, i)$ je nahrazena funkcí $f_E(i)$ kde E je událost a i je číslo nebo proměnná. Dále je nutné převést formuli F' na formuli F'' v klauzální formě. Výsledná formule F'' je pak ve formě

$$C_1 \wedge C_2 \wedge \ldots \wedge C_{n-1} \wedge C_n$$

kde C_i pro $i = 1..n$ je disjunktní klauzule ve tvaru

$$L_1 \vee L_2 \vee \ldots \vee L_{m-1} \vee L_m$$

a L_j pro $j = 1..m$ je literál ve tvaru

$$v_1 \pm I \leq v_2$$

kde v_1 a v_2 jsou neinterpretované celočíselné funkce odpovídající funkcím výskytu a I je celočíselná konstanta.

Pokud jsou k dispozici specifikace systému SP a bezpečnostní tvrzení SA vyjádřené pomocí omezených RTL formulí, zbývá dokázat, že SA je teorém odvoditelný z SP, tedy ověřit validitu výroku $SP \rightarrow SA$. Ověřování validity formulí u logik 1. řádů je často problematické, lze ovšem využít dualitu validity a splnitelnosti uvedenou v kapitole 3.2.1.1. Ověření validity formule $SP \rightarrow SA$ lze pak jednoduše převést na odpovídající problém ověření nesplnitelnosti formule $\neg(SP \rightarrow SA)$. Protože formule $SP \rightarrow SA$ lze zapsat jako $\neg SP \vee SA$, stačí tedy dokázat, že její negace, formule $SP \wedge \neg SA$, je nesplnitelná. Proces verifikace se tedy místo potvrzení platnosti bezpečnostního tvrzení bude snažit vyvrátit jeho opak.

Pro větší názornost předložených postupů a úprav budou jednotlivé kroky demonstrovány na příkladě dále. Ten bude navazovat na příklad z minulé kapitoly, jehož výsledkem byl zápis specifikace systému a bezpečnostního pravidla ve formě RTL formulí. Nyní je potřeba převést tyto formule do klauzálního tvaru, který navíc splňuje podmínky a omezení probírané v této kapitole.

Specifikace systému je vetšinou tvořena celou množinou formulí, které popisují jeho chování. Všechny tyto formule samozřejmě musí platit zároveň, specifikaci celého systému lze tedy popsat jedinou formulí, která vznikne konjunkcí všech formulí z této množiny. Specifikace systému v příkladě z předchozí kapitoly obsahuje pouze dvě formule, výsledná formule popisující celý systém bude vypadat následovně:

$$f(x) \leq g_1(x) \wedge g_2(x) \leq f(x) + 30 \wedge g_1(y) + 15 \leq g_2(y)$$

Je patrné, že tato formule je již v klauzální formě, tedy v tomto ohledu již nejsou potřeba žádné úpravy. Ovšem druhá klauzule stále nesplňuje omezení kladené na tvar aritmetické nerovnosti klauzulí. Je tedy potřeba provést její úpravu. První klauzule tvar aritmetické nerovnosti splňuje, hodnota celočíselné konstanty je zde jednoduše 0, ekvivalentně by mohla být zapsána ve tvaru $f(x) + 0 \leq g_1(x)$, což odpovídá požadavkům. Výsledná upravená formule se bude skládat celkem ze tří klauzulí:

$$f(x) \leq g_1(x)$$

[1] Přesněji na odpovídající formuli v celočíselné diferenční logice s neinterpretovanými funkcemi

$$g_2(x) - 30 \leq f(x)$$

$$g_1(y) + 15 \leq g_2(y)$$

Bezpečnostní tvzení je vyjádřeno vždy jedinou, často složitější, formulí. Ikdyž nic nebrání existenci více samostatných bezpečnostních tvrzení, z pohledu analýzy se neprovádí ověřování všech těchto tvrzení zároveň. Je sice možné sloučit tato tvrzení do jediné formule, jak se to dělá u specifikace systému, z hlediska efektivity a paměťové náročnosti je ovšem výhodnější ověřovat každé tvrzení samostatně. Protože popsaná verifikace se snaží vyvrátit opak bezpečnostního tvrzení, pracuje se dále s negací tohoto tvrzení, což je formule

$$\neg(\forall t \forall u\ f(t) + 45 \leq h(u) \wedge h(u) < f(t) + 60 \rightarrow g_2(t) \leq h(u) \wedge h(u) \leq g_2(t) + 45)$$

Tato formule obsahuje implikaci, takže určite není v klauzální formě. Nejprve je tedy potřeba převést formuli do této formy. Po odstranění implikace vznikne formule

$$\neg(\forall t \forall u\ \neg(f(t) + 45 \leq h(u) \wedge h(u) < f(t) + 60) \vee (g_2(t) \leq h(u) \wedge h(u) \leq g_2(t) + 45))$$

Po provedení negací zůstane formule

$$\exists t \exists u\ (f(t) + 45 \leq h(u) \wedge h(u) < f(t) + 60) \wedge (h(u) < g_2(t) \vee g_2(t) + 45 < h(u))$$

Pro konečné převedení formule do klauzální formy je třeba eliminovat existenční kvantifikátory, což lze jednoduše provést pomocí skolemizace uvedené v kapitole 3.2.1.2, kde proměnné t a u budou nahrazeny skolemovými konstantami T a U. Výsledná formule

$$f(T) + 45 \leq h(U) \wedge h(U) < f(T) + 60 \wedge (h(U) < g_2(T) \vee g_2(T) + 45 < h(U))$$

je již formulí v klauzální formě, ovšem některé klauzule opět nesplňují omezení tvaru aritmetické nerovnosti. Po patřičných úpravách obsahuje výsledná formule tři následující klauzule:

$$f(T) + 45 \leq h(U)$$

$$h(U) - 59 \leq f(T)$$

$$h(U) + 1 \leq g_2(T) \vee g_2(T) + 46 \leq h(U)$$

Tyto klauzuje již splňují veškerá omezení. Specifikace systému i bezpečnostní tvrzení jsou tedy nyní ve tvaru, který je potřebný pro konstrukci grafu omezení.

4.5 Graf omezení

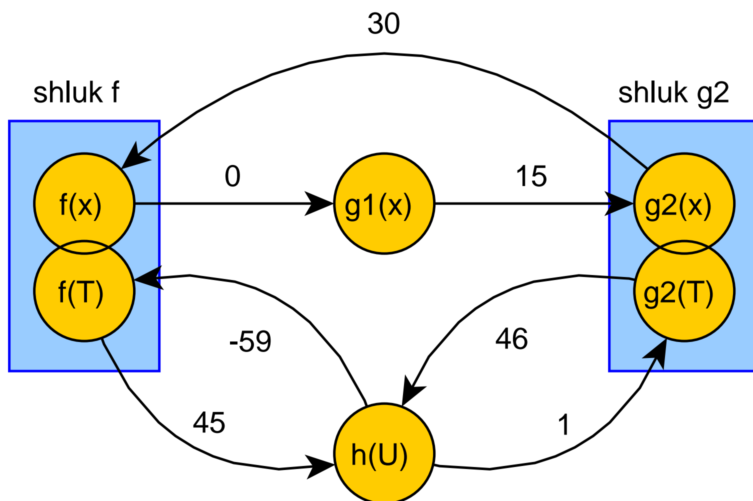
Algoritmus konstrukce grafu omezení vyžaduje, aby formule zahrnující specifikaci systému a bezpečnostní tvrzení splňovaly požadavky uvedené v kapitole 4.4. Pro každý literál ve tvaru $v_1 \pm I \leq v_2$ se poté zkonstruuje uzel označený v_1, uzel označený v_2 a hrana $\langle v_1, v_2 \rangle$ s váhou $\pm I$ směřující z uzlu v_1 do uzlu v_2. Algoritmus pro konstrukci grafu omezení je následující:

Vstup: Množina klauzulí S
Výstup: Graf omezení G
Metoda:

- Na počátku je graf G prázdný.

- Pro každou klauzuli $C_i \in S$, pro každý literál $L_j \in C_i$, kde L_j je ve tvaru $v_1 \pm I \leq v_2$:
    1. Najdi shluk s funkčním symbolem termu v_1. Pokud takovýto shluk neexistuje, vytvoř nový.
    2. Najdi uzel označený v_1 ve shluku nalezeném nebo vytvořeném v bodě 1. Pokud takovýto uzel neexistuje, přidej uzel označený v_1 do tohoto shluku.
    3. Opakuj body 1 a 2 pro term v_2.
    4. Vytvoř orientovanou hranu $\langle v_1, v_2 \rangle$ s váhou $\pm I$ jdoucí z uzlu v_1 do uzlu v_2.

Množina klauzulí, kterou využívá algoritmus, se odvíjí od způsobu ověřování platnosti bezpečnostního tvrzení. Zde se využívá přístup vyvrácení platnosti opaku tohoto tvrzení, tedy ověření platnosti formule $F'' = SP \wedge \neg SA$. Množina klauzulí v tomto případě bude tedy obsahovat klauzule obsažené ve formuli SP a znegované formuli SA. Na obrázku 4.1 je zobrazen graf omezení pro probíraný příklad.



Obrázek 4.1: Graf omezení

4.6 Ověření nesplnitelnosti

Pro ověření nesplnitelnosti formule F'' lze nyní využít zkonstruovaný graf omezení G, který tuto formuli reprezentuje. Důkaz nesplnitelnosti formule F'' se provádí pomocí identifikace kladných cyklů v grafu G. Pro nalezení těchto cyklů je potřeba nejprve definovat proces unifikace a pojmy cesta a cyklus v tomto typu grafu.

Unifikace: Unifikací v_i a v'_i se označuje případ, kdy existuje substituce σ taková, že $v_i\sigma = v'_i\sigma$, kde $v_i\sigma$ a $v'_i\sigma$ jsou termy vzniklé aplikací substituce σ na v_i a v'_i.

Cesta: Cestou z uzlu v_0 do uzlu v_n v grafu G se označuje sekvence hran

$$\langle v_0, v_1 \rangle, \langle v'_1, v_2 \rangle, \langle v'_2, v_3 \rangle, \ldots, \langle v'_{n-2}, v_{n-1} \rangle, \langle v'_{n-1}, v_n \rangle$$

pro kterou platí, že existuje substituce σ, pomocí níž lze provést párovou unifikaci v_i a v'_i pro všechna $1 \leq i < n$. Tedy pro každý pár v_i a v'_i, kde $1 \leq i < n$, musí platit, že uzly v_i a v'_i jsou totožné nebo se nacházejí ve stejném shluku.

Cyklus: Cyklem v grafu G se označuje sekvence hran

$$\langle v_0, v_1 \rangle, \langle v'_1, v_2 \rangle, \langle v'_2, v_3 \rangle, \ldots, \langle v'_{n-2}, v_{n-1} \rangle, \langle v'_{n-1}, v_n \rangle$$

která je cestou vytvořenou na základě substituce σ a zároveň lze pomocí této substituce σ provést unifikaci v_0 a v_n. Opět musí platit, že uzly v_0 a v_n jsou totožné nebo se nacházejí ve stejném shluku. Cyklus je tedy jednoduše cesta, pro kterou lze unifikovat její počáteční a koncový uzel. Váha cesty nebo cyklu je poté definována jako suma všech vah hran, které tato cesta nebo cyklus obsahují.

Nyní lze již dokázat, že pokud existuje v grafu G, který odpovídá formuli F'', pozitivní cyklus, pak formule, která je složena s konjunkcí literálů odpovídajících jednotlivým hranách v tomto cyklu, je nesplnitelná. Aplikováním substituce σ na každý literál (nerovnost) L_i v cyklu, tedy na formuli tvaru:

$$\begin{gathered} v_0\sigma + I_0 \leq v_1\sigma \wedge \\ v'_1\sigma + I_1 \leq v_2\sigma \wedge \\ \vdots \\ v'_{n-2}\sigma + I_{n-2} \leq v_{n-1}\sigma \wedge \\ v'_{n-1}\sigma + I_{n-1} \leq v_n\sigma \end{gathered}$$

a sloučením těchto nerovností vznikne formule

$$I_0 + I_1 + \ldots + I_{n-1} + I_n \leq 0$$

která je očividně nesplnitelná, protože součet $I_0+I_1+\ldots+I_{n-1}+I_n$ je suma vah jednotlivých hran cyklu, a tato suma u kladného cyklu nemůže být menší ani rovna nule. Nesplnitelnost této nerovnosti znamená, že také původní RTL nerovnosti, odpovídající těmto hranám, jsou nesplnitelné.

4.7 Prohledávací strom

Pokud každá hrana kladného cyklu odpovídá literálu, který náleží jednotkové klauzuli, tedy klauzule je tvořena pouze tímto literálem, pak musí být formule F'' nesplnitelná a bezpečnostní tvrzení SA odvoditelné ze specifikace systému SP. Ovšem pokud hrana v nějakém cyklu odpovídá literálu, který je součástí nejednotkové klauzule, je potřeba ukázat, že zbylé literály této klauzule odpovídají hranám z jiných kladných cyklů. Potřeba tohoto důkazu jednoduše plyne z faktu, že klauzule jsou disjunktní, takže pro dokázání nesplnitelnosti celé klauzule je potřeba ukázát, že všechny její literály (disjunkty) jsou nesplnitelné. Například negace bezpečnostního tvrzení obsahuje klauzuli

$$h(U) + 1 \leq g_2(T) \vee g_2(T) + 46 \leq h(U)$$

Jestliže existuje pozitivní cyklus obsahující hranu odpovídající nerovnosti $h(U)+1 \leq g_2(T)$, pak je navíc potřeba ukázát, že existuje jiný kladný cyklus, který obsahuje hranu odpovídající nerovnosti $g_2(T) + 46 \leq h(U)$.

Samozřejmě, že s počtem hran pozitivních cyklů, jejichž odpovídající literály náleží nejednotkovým klauzulím, roste kombinatoricky také potřeba identifikace dalších kladných cyklů. Problém ověření nesplnitelnosti F'' je NP-úplný. Z důvodu složitosti řešení tohoto problému je důležité proces verifikace maximálně zefektivnit.

Jeden z algoritmů, který efektivněji řeší popsaný problém, využívá tzv. prohledávacího stromu. Tento algoritmus vychází z následujících pozorování. Mějme formuli F'' v konjunktní formě, tedy ve tvaru

$$C_1 \wedge C_2 \wedge \ldots \wedge C_{n-1} \wedge C_n$$

kde každá klauzule C_i, $i = 1..n$, je v disjunktní formě, tedy ve tvaru

$$L_{k,1} \vee L_{k,2} \vee \ldots \vee L_{k,m_k-1} \vee L_{k,m_k}$$

kde literály v ruzných klauzulích mohou být stejné. Zápis

$$X_{i,1}, X_{i,2}, \ldots, X_{i,n_i-1}, X_{i,n_i}$$

bude označovat nerovnosti odpovídající hranám v i-tém kladném cyklu grafu G, kde $X_{i,j}$ je literál odpovídající j-té hraně v i-tém kladném cyklu a každá nerovnost $X_{i,j}$ se vyskytuje alespoň v jedné klauzuli C_k. Nechť

$$P_i = X_{i,1} \wedge X_{i,2} \wedge \ldots \wedge X_{i,n_i-1} \wedge X_{i,n_i}$$

Z kapitoly 4.6 je již známo, že formule P_i je nesplnitelná. Pak F'' je ale splnitelná právě tehdy když $F'' \wedge \neg P_i$ je splnitelná. Ve vysledku je tedy existence kladného cyklu ekvivalentní přidání klauzule

$$\neg P_i = \neg X_{i,1} \vee \neg X_{i,2} \vee \ldots \vee \neg X_{i,n_i-1} \vee \neg X_{i,n_i}$$

do formule F'', což procesu verifikace umožňuje využít navíc klauzuli $\neg P_i$ při ověřování nesplnitelnosti F''.

Protože popis algoritmu ve formě formálního zápisu či pseudokódu není příliš srozumitelný, bude jeho činnost popsána názorně na praktickém příkladě. Pro větší přehlednost konstruovaného prohledávacího stromu se nejprve přiřadí každému literálu z množiny klauzulí S formule F'' písmeno abecedy, které bude tento literál reprezentovat. Odpovídající přiřazení jsou následující:

$$\begin{aligned}
A &= f(x) \leq g_1(x) \\
B &= g_2(x) - 30 \leq f(x) \\
C &= g_1(y) + 15 \leq g_2(y) \\
D &= f(T) + 45 \leq h(U) \\
E &= h(U) - 59 \leq f(T) \\
F \vee G &= h(U) + 1 \leq g_2(T) \vee g_2(T) + 46 \leq h(U)
\end{aligned}$$

Poté je potřeba vytvořit klauzule $\neg P_i$ reprezentující kladné cykly v grafu G, které budou přidány ke klauzulím formule F'', množina S_C těchto klauzulí obsahuje:

$$\begin{aligned}
&\neg F \vee \neg G \\
&\neg B \vee \neg D \vee \neg F \\
&\neg A \vee \neg C \vee \neg G \vee \neg E
\end{aligned}$$

Algoritmus následně vytváří prohledávací stromu s využitím klauzulí z množiny S_C a při jeho konstrukci ověřuje nesplnitelnost klauzulí z množin S a S_C. Každá nová úroveň tvořeného stromu je výsledkem analýzy další klauzule z množiny S_C, tedy analýzy dalšího kladného cyklu v grafu G. každý uzel stromu je buď samostatný literál nějaké klauzule z množiny S_C nebo konjunkce literálů z různých klauzulí množiny S_C. Na obrázku 4.2 je nejhorší případ vytvořeného prohlédavacího stromu ilustrovaného příkladu, kde jsou prozkoumány všechny konjunkce jednotlivých literálů. První úroveň stromu je tvořena odpovídajícími literály první klauzule z množity S_C, tedy literály $\neg F$ a $\neg G$. Pro vytvoření další úrovně se přidají ke všem uzlům aktuální úrovně literály další, ještě neanalyzované, klauzule. V tomto případě se přidají literály druhé klauzule, tedy literály $\neg B$, $\neg D$ a $\neg F$. Tento postup se opakuje, dokud se neanalyzují všechny klauzule z množiny S_C, tedy veškeré kladné cykly. Z obrázku 4.2 je patrné, že strom narůstá exponenciálně s počtem klauzulí množiny S_C. Naštěstí v praxi není potřeba velké množství uzlů vůbec analyzovat, tedy ani vytvářet. Metody pro redukci stavového prostoru budou obsahem další kapitoly.

Pro důkaz, že konjunkce klauzulí z množin S a S_C, tedy formule F'', je nesplnitelná, je potřeba ukázát, že každý listový uzel stromu splňuje jednu z následujících podmínek:

1. Konjunkce literálů listového uzlu a alespoň jedné klauzule z množiny S je nesplnitelná, tedy nabývá ohodnocení $false$.

2. Konjunkce literálů listového uzlu je sama nesplnitelná.

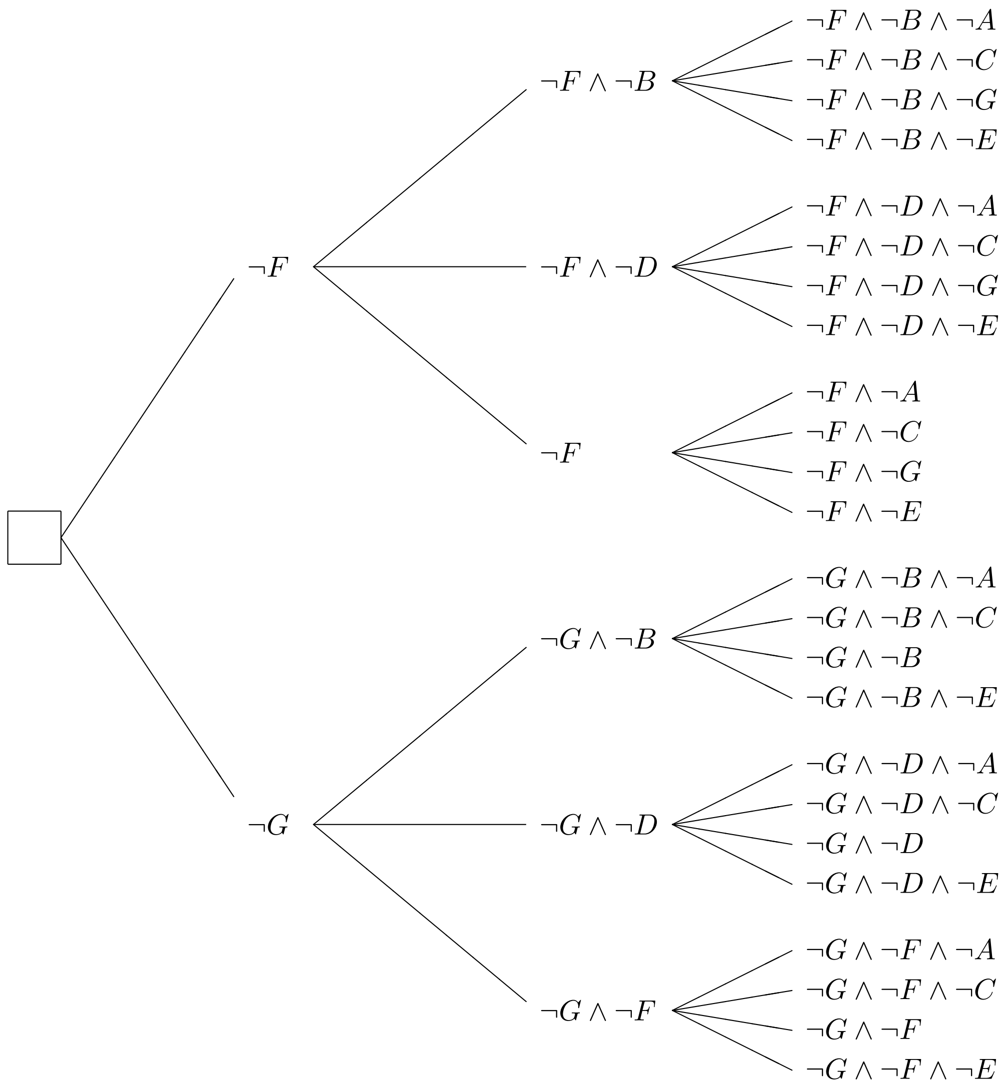
První podmínka vychází z jednoduché logické úvahy. Pokud existuje formule $C_k \wedge \neg C_k$, kde C_k je klauzule z množiny S a $\neg C_k$ je konjunkcí literálů v listovém uzlu stromu, tedy formule je ve tvaru

$$(L_{k,1} \vee L_{k,2} \vee \ldots \vee L_{k,m_k-1} \vee L_{k,m_k}) \wedge (\neg L_{k,1} \wedge \neg L_{k,2} \wedge \ldots \wedge \neg L_{k,m_k-1} \wedge \neg L_{k,m_k})$$

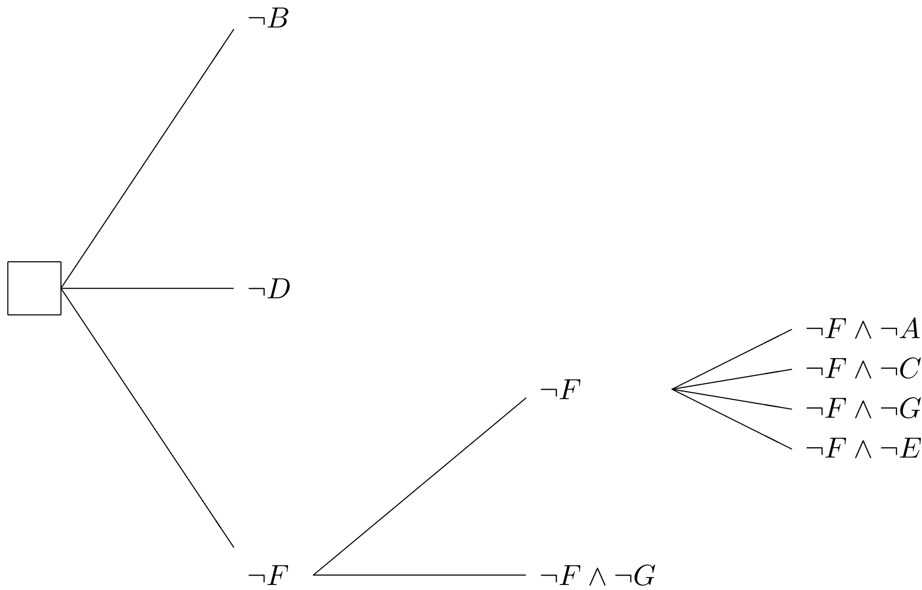
pak je tato formule vždy nesplnitelná. Stačí tedy jednoduše ukázat, že některý z literálů L_i v klauzuli $\neg C_k$ je nesplnitelný, tedy že existuje komplementární literál $\neg L_i$, který je vždy splnitelný. Například nesplnitelnost prvního listového uzlu v probíraném příkladě obsahující konjunkci $\neg F \wedge \neg B \wedge \neg A$ může být dokázána pomocí jedné z klauzulí A nebo B z množiny S. Obě tyto klauzule obsahují jediný literál, který tedy musí být vždy splněn, zároveň ale má být splněn i výraz v listovém uzlu, který vyžaduje platnost opaku obou těchto literálů. Konjunkce listového uzlu s jednou z těchto klauzulí, tedy např. $A \wedge (\neg F \wedge \neg B \wedge \neg A)$, je tedy určitě nesplnitelná. Protože pro každý listový uzel lze najít alespoň jednu klauzuli z množiny S, která dokáže jeho nesplnitelnost, lze říci, že klauzule z množin S a S_C jsou nesplnitelné, a tedy i formule F'' je nesplnitelná.

4.8 Optimalizace

Jak již bylo řečeno v předchozí kapitole, v nejhorším případě je prohledávací strom vytvořen analýzou všech kladných cyklů. Časová složitost tohoto algoritmu je bohužel exponenciální s ohledem na počet kladných cyklů. Pokud graf omezení G obsahuje celkem n kladných cyklů a každý cyklus obsahuje m hran, pak množina S_C obsahuje celkem n klauzulí, kde každá klauzule je disjunkcí m literálů. Pak prohledávací strom vytvořený z této množiny klauzulí bude obsahovat celkem m^n listových uzlů, které, v nejhorším případě, bude potřeba všechny ověřit. Ikdyž je výpočetní složitost algoritmu exponenciální, existují různé možnosti optimalizace, které často výrazně redukují velikost stavového prostoru, tedy počet uzlů stromu, a tím i časovou náročnost ověřování.



Obrázek 4.2: Prohledávací strom



Obrázek 4.3: Prohledávací strom při použití optimalizací

První možností redukce prohledávacího stromu je zastavení generování nových uzlů z uzlu, jež sám činí množinu klauzulí S nesplnitelnou. Protože uzly obsahují konjunkce negací literálů, musí být pro celkovou platnost tohoto uzlu platné všechny tyto literály. Pokud se dokáže neplatnost některé negace literálu v uzlu, pak přidáním dalších literálů k tomuto uzlu, tedy vygenerováním synovských uzlů, budou tyto uzly opět nesplnitelné, nemá tedy smysl je vůbec generovat. Například synovské uzly uzlu $\neg F \wedge \neg B$ není potřeba již generovat, protože konjunkce tohoto uzlu s klauzulí B z množiny S je nesplnitelná, tedy tento uzel již zneplatňuje množinu S, všechny synovské uzly vygenerované z tohoto uzlu by opět vyžadovaly platnost $\neg F \wedge \neg B$, která určitě nikdy nebude splněna.

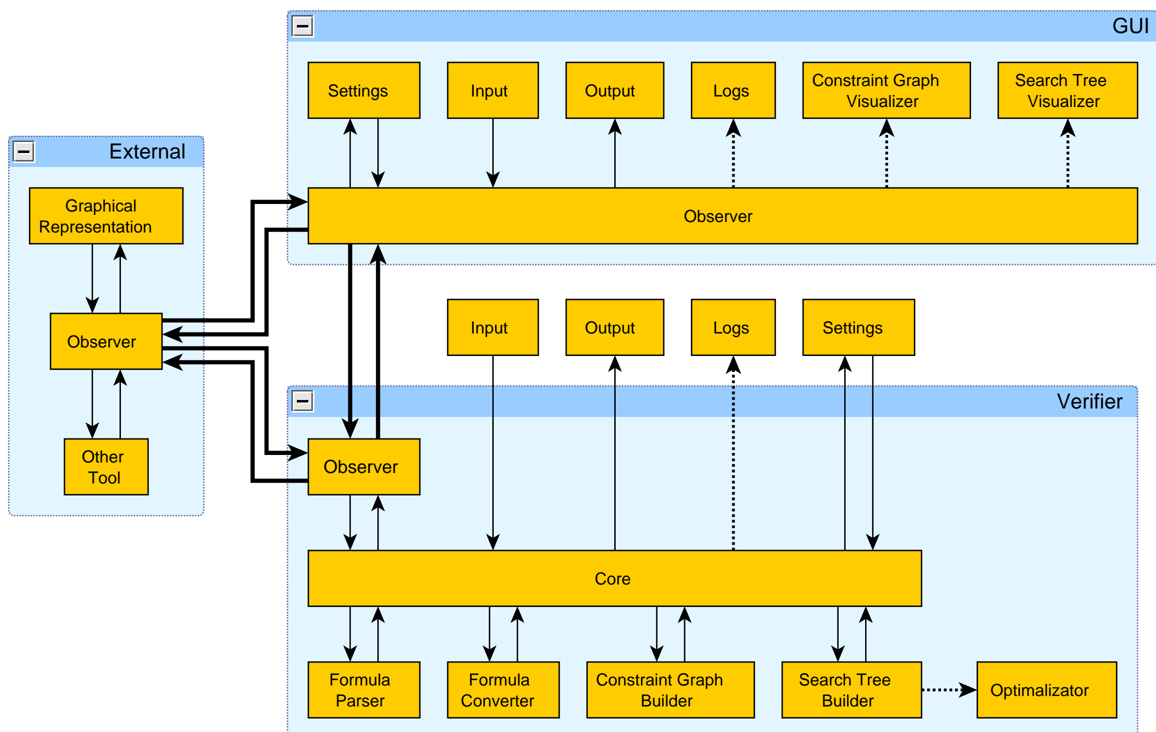
Druhou možností optimalizace je přeskupení pořadí analýzy klauzulí z množiny S_C tak, aby bylo možné aplikovat první optimalizaci čím jak nejdříve. Čím blíže kořeni stromu se generování uzlů zastaví, tím větší bude výsledná redukce tohoto stromu. Nevýhodou této optimalizace je nutnost využití různých heuristik, které určí pořadí analýzy klauzulí z množiny S_C, nebo backtrackingu v případě zjištění, že dané pořadí nevede k přínosné redukci. V předchozím příkladě již zmíněný uzel $\neg F \wedge \neg B$ způsoboval zneplatnění množiny S, protože jeho konjunkce s klauzulí B je nesplnitelná. Je zřejmé, že literál $\neg F$ jinak nepřispívá k zneplatnění množiny S. Je tedy možné odložit analýzu první klauzule z množiny S_C a přednostně začít analýzou druhé klauzule, která více napomáhá ověřování neplatnosti. Přehození pořadí analýzy prvních dvou klauzulí způsobí, že již na první úrovni stromu lze zastavit generování synovských uzlů pro uzly $\neg B$ a $\neg D$, protože tyto uzly již zajišťují nesplnitelnost množiny S. Použití obou zmíněných optimalizací má za následek výraznou redukci prohledávacího stromu do podoby na obrázku 4.3.

Třetí optimalizace využívá dříve vygenerovaných uzlů, které již byly označeny jako nesplnitelné. Tedy nově vygenerovaný uzel v je nesplnitelný, jestliže již někdy dříve byl vygenerován uzel označený v. Například nejnižší uzel $\neg F \wedge \neg G$ ve stromu na obrázku 4.3 se již nemusí ověřovat, protože tento stejný uzel byl již vygenerován a ověřen dříve jako listový uzel.

Kapitola 5

Návrh nástroje pro verifikaci

Protože metod verifikace je obecně velké množství a řada metod může využívat různé optimalizace a heuristiky, je vhodné, aby byl nástroj dobře rozšiřitelný. Samozřejmě je důležité, aby samotný nástroj byl rychlý a efektivní, ale pokud možno jednoduchý pro použití. Na obrázku 5.1 je zobrazen návrh blokového schématu nástroje s přihlédnutím k výše uvedeným základním požadavkům.



Obrázek 5.1: Blokové schéma nástroje

Nástroj se skládá ze dvou základních částí, verifikátor (*Verifier*) a grafické uživatelské rozhraní (*GUI*). Verifikátor řeší samotné ověřování vlastností systémů popsaných ve formě RTL formulí. Jako implementační jazyk této části lze použít např. jazyk C++, který

umožňuje celkem snadnou tvorbu i složitějších programů, zatímco si zachovává slušnou rychlost a efektivitu. Navíc existuje pro tento jazyk řada knihoven, které mohou být užitečné, např. knihovny pro vizualizaci grafů a stromů, nebo knihovny pro paralelizaci výpočtů. Jednotlivé části procesu ověřování u metody popsané v kapitole 4 jsou zde zastoupeny ve formě modulů. První modul, *Formula Parser*, zajišťuje zpracování vstupních formulí a jejich převedení do interní reprezentace. Druhý modul, *Formula Converter*, převádí formule do diferenční logiky, která je vyžadována pro konstrukci grafu omezení. Třetí modul, *Constraint Graph Builder*, vytváří samotný graf omezení. Čtvrtý modul, *Search Tree Builder*, pak vytváří prohledávací strom a provádí ověřování jeho uzlů. Tento modul může navíc využívat různé optimalizační moduly. Nastavení programu se může načítat např. z externích XML souborů. Verifikátor lze spouštět jako samostatný program z příkazové řádky, čímž poskytuje možnosti skriptování a nezávislost na grafických rozhraních.

Grafické uživatelské rozhraní slouží k zjednodušení použití verifikátoru. Jako implementační jazyk lze použít např. jazyk Java, který umožňuje jednoduše vytvořit uživatelské rozhraní, které je snadno přenositelné mezi různými operačními systémy. Kromě ovládání verifikátoru, tedy nastavení parametrů programu a zpracování vstupů, výstupů a záznamů o činnosti, zde jsou i další moduly. Modul *Constraint Graph Visualizer* slouží k zobrazení vytvořeného grafu omezení a modul *Search Tree Visualizer* k zobrazení prohledávacího stromu. Tyto moduly mohou být užitečné např. při kontrole správnosti procesu ověřování nebo pro ilustraci činnosti ověřovací metody pro studijní účely.

Poslední důležitou částí návrhu je způsob komunikace. Komunikace mezi jednotlivými moduly je celkem bezproblémová, jelikož je to komunikace mezi částmi implementovanými ve stejném jazyce. Lze například využít rozhraní v jazyce Java a virtuální metody v jazyce C++. Komunikace mezi samotným verifikátorem a grafickým uživatelským rozhraním je již složitější. Jednou z možností, která se zde nabízí, je využití systému CORBA. CORBA poskytuje univerzální rozhraní pro volání metod objektů, její implementace je k dispozici pro velkou řadu jazyků, včetně jazyků Java a C++. Umožňuje tedy standardizovanou komunikaci nezávislou na použitém jazyce, což je přesně to, co je zde potřeba. CORBA se využívá hlavně u distribuovaných aplikací, kde části aplikace mohou být situovány na různých počítačích. Tato vlastnost je zde také přínosná, protože může být velice výhodné, a často i nezbytné, aby verifikátor pracoval na jiném stroji než samotné grafické uživatelské rozhraní. Jednotlivé části obsahují modul *Observer*, který zajišťuje realizaci komunikace a v případě grafického uživatelského rozhraní také koordinuje komunikaci mezi jednotlivými moduly, navíc tento modul umožňuje zprostředkování komunikace s externími programy a tím jednoduché využití verifikátoru dalšími nástroji.

Kapitola 6

Realizace nástroje pro verifikaci

Tato kapitola se zabývá celkovou realizací nástroje pro verifikaci. Nástroj se skládá ze dvou celků, z verifikátoru, programu ověřujícího zadané vlastnosti vloženého systému, a z grafického rozhraní, aplikace, jenž slouží pro snadnější a uživatelsky přívětivější použití verifikátoru. Následující podkapitoly popisují konečný návrh obou těchto celků a zabývají se řešením jednotlivých kroků verifikačního procesu a dalších důležitých implementačních problémů.

6.1 Implementační prostředí

Vhodný výběr implementačního prostředí výrazně ovlivňuje rychlost a efektivnost jak samotného vývoje, tak i vytvořeného programu. Je také důležité uvědomit si rozdílné požadavky obou celků nástroje.

Z pohledu verifikátoru je důležitá hlavně rychlost výpočtu, implementační prostředí musí tedy poskytovat vysokou rychlost a efektivitu, ale zároveň také konstrukce na vyšší úrovni abstrakce, aby byl řešený problém zvládnutelný. Všechny tyto požadavky nejlépe splňuje jazyk C++, který byl také vybrán jako implementační jazyk verifikátoru. Jazyk C++ je objektově orientovaný programovací jazyk umožňující například polymorfismus, přetěžování metod a operátorů nebo použití vyjímek. Protože jde o kompilovaný jazyk, programy v tomto jazyce jsou velice výkonné, narozdíl od vyšších programovacích jazyků. Možnou nevýhodou tohoto jazyka je však horší přenositelnost programů na jiné architektury a operační systémy. Tento problém lze ovšem vyřešit podmíněným překladem a získaná rychlost a efektivita stále výrazně převyšují tuto, trochu nepříjemnou, skutečnost.

Grafické rozhraní si klade úplně jiné priority z hlediska potřeb implementace. Jelikož celý verifikační proces vykonává verifikátor, grafické rozhraní vůbec nevyžaduje rychlý a efektivní chod. Nejdůležitějším požadavkem je zde dobrá přenostitelnost aplikace. Tato aplikace slouží jako grafická nádstavba nad verifikátorem, jenž má za úkol pouze shromažďovat vstupní data, zjednodušit konfiguraci a vhodně interpretovat a zobrazovat data výstupní. K tomuto účelu lze s výhodou využít platforem .NET nebo Java. Obě platformy kompilují vytvořené programy pouze to speciálního mezikódu, který následně interpretují. Důsledkem je možnost spuštění již zkompilovaného programu kdekoliv, kde je k dispozici daná platforma, tedy nezávisle na architektuře nebo operačním systému, na kterých tato platforma běží. Kromě této nezávislosti navíc poskytují obě tyto platformy možnosti pro tvorbu grafického uživatelského rozhraní a obsahují velké množství vestavěných knihoven. Cenou za veškeré tyto výhody je výrazné zpomalení chodu programů, kód totiž musí být

interpretován. Nepomáhají zde ani různé optimalizace za chodu, jenž tyto platformy často provádí a zvyšují tak rychlost interpretace. Jelikož platforma .NET stále není příliš rozšířená v operačních systémech jako je Linux, byla pro implementaci grafického rozhraní zvolena platforma Java a její stejnojmenný jazyk.

6.2 Modularita

Protože je implementovaný program značně rozsáhlý a složitý je vhodné jej vystavět modulárně. Modularitu lze pozorovat na dvou úrovních:

- **Interní modularita programu** - Cílem je rozdělit vnitřní strukturu programu na jednotlivé části, které se vyznačují vysokou nezávislostí. Zde jsou tyto části reprezentované subsystémy, které plní vždy určitou úlohu bez větší návaznosti na zbytek programu. Výhodou je zjednodušení a zpřehlednění kódu a určitá centralizace specifických služeb. Při nutnosti modifikace jedné služby pak nehrozí nebezpečí zavlečení chyb do jiných částí programu. V případě potřeby nové služby se jednoduše přidá nový subsystém a definuje se interakce tohoto subsystému s ostatními částmi programu, pokud je to potřeba.

- **Externí modularita programu** - Cílem je zajistit jednoduchou rozšiřitelnost programu bez nutnosti rekompilace nebo dokonce modifikace zdrojového kódu programu. Tato modularita je zajištěna zásuvnými moduly. Tyto moduly jsou reprezentovány pomocí dynamických knihoven, sdílených objektů nebo speciálních archívů v závislosti na použitém implementačním prostření a jsou dynamicky načítány za chodu programu.

6.3 Verifikátor

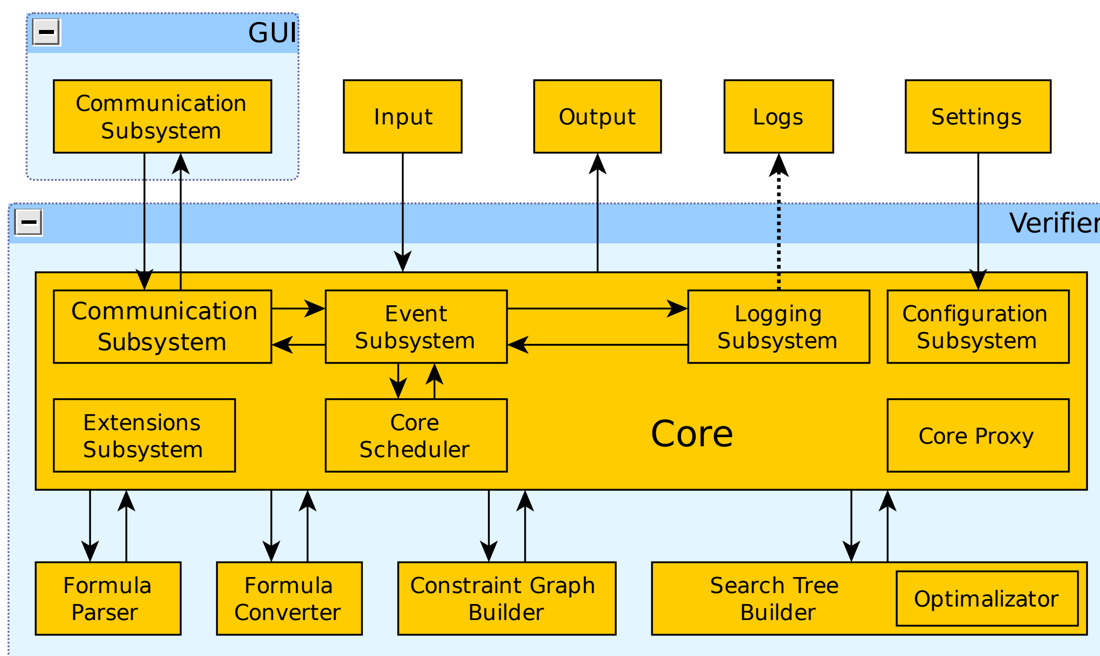
Jak již bylo zmíněno v kapitole 6.1, verifikátor je napsán v jazyce C++. Návrh je vysoce modulární s cílem zajistit jednoduchou modifikovatelnost kódu, snadnou rozšiřitelnost programu a částečnou nezávislost na konkrétních implementacích jednotlivých částí. K dosažení těchto cílů výrazně přispívá využití velkého počtu návrhových vzorů[3]. Blokové schéma na obrázku 6.1 zachycuje detailní strukturu verifikátoru.

Verifikátor se skládá celkem z šesti částí - jádra a pěti subsystémů. Jádro a téměř všechny subsystémy jsou inicializovány při startu verifikátoru. Jedinou vyjímkou je komunikační subsystém, který je nastartován pouze, pokud je verifikátor spuštěn v tzv. *módu serveru*, jinak totiž není potřeba.

6.3.1 Jádro

Jádro je reprezentováno třídou `Core`, jenž obsahuje všechny subsystémy. Tato třída je implementována jako návrhový vzor *singleton*[3], vždy tedy existuje pouze jediná instance této třídy a k této instanci je umožněn přístup v celém programu.

Hlavním úkolem jádra je inicializace a konfigurace všech potřebných subsystémů. Jádro zajišťuje registraci interních a načtení externích zásuvných modulů, zpracování konfigurace a nastavení jednotlivých subsystémů podle této konfigurace. Umožňuje také spustit samotný proces verifikace.



Obrázek 6.1: Detailní blokové schéma verifikátoru

6.3.2 Proxy jádra

Protože přímý přístup k jednotlivým subsystémům by byl značně nebezpečný a umožňoval by, ať již cíleně nebo pouze omylem, modifikovat důležitá nastavení těchto subsystémů, které by mohly zapříčinit nesprávný chod jádra nebo celého programu, jsou veškeré subsystémy chráněny jádrem.

Přístup k důležitým službám jádra je možný pouze skrz obecné rozhraní `ICoreProxy`. Toto rozhraní je implementováno jako návrhový vzor *facade*[3] a slouží pro jednotný přístup ke službám jádra. Rozhraní přesně definuje služby, které musí jádro poskytovat svému okolí, tedy konkrétním subsystémům a zásuvným modulům.

Proxy jádra, instance třídy `CoreProxy`, je konkrétní implementací rozhraní `ICoreProxy`. Požadované služby jsou zde jednoduše delegovány na specifické subsystémy, které mohou příchozí požadavek vyřídit.

6.3.3 Plánovač jádra

Plánovač jádra, instance třídy `CoreScheduler`, je potřebný pouze v tzv. *módu serveru*. Zajišťuje totiž plánování akcí, které jsou potřeba pro vyřízení požadavků vzdáleného klienta. Po vykonání všech těchto akcí sestavuje také výslednou zprávu, jenž je zaslána jako odpověď na příchozí požadavek.

6.3.4 Událostní subsystém

Událostní subsystém realizuje komunikaci založenou na zasílání událostí. Událost je reprezentována objektem třídy `Event` nebo třídy z ní zděděné. Událostní subsystém rozlišuje

celkem dva typy událostí:

- **Základní (*basic*) události** - Tyto události jsou identifikovány celočíselnou konstantou, kterou lze získat metodou `getId()`, a využívá je hlavně sám verifikátor pro svou činnost. Patří zde například třída `LogEvent` reprezentující zápis do protokolu událostí.
- **Uživatelem definované (*custom*) události** - Tyto události jsou identifikovány řetězcem, který lze získat metodou `getStringId()`. Všechny tyto události musejí mít nastaven celočíselný identifikátor na hodnotu konstanty `CUSTOM`, tak událostní sybsystém pozná, že se jedná o uživatelem definovanou událost. Tento typ událostí umožňuje uživateli vytvářet vlastní události, které lze například použít pro komunikaci mezi různými externími zásuvnými moduly.

Při vytváření nového typu události je nutné korektně implementovat metodu `clone()`, jenž slouží k vytváření kopií dané události. Kopie událostí jsou pořizovány v případě, že má být stejná událost doručena více naslouchajícím objektům. Cílové objekty, které zpracovávají přijatou událost, musí také korektně uvolnit objekt události, jakmile ho již dále nepotřebují.

Událostní subsystém je reprezentován třídou `EventMediator`, která je implementována jako návrhový vzor *mediator*[3] a působí jako prostředník mezi dvěmi komunikujícími stranami. Objekty nikdy nezasílají událost přímo cílovému objektu, ale vždy oněmu prostředníkovi, vytvořené instanci třídy `EventMediator`, který je filtruje a propaguje cílovým objektům. Aby cílový objekt přijímal události specifického typu, musí se nejprve zaregistrovat u prostředníka jako odběratel tohoto typu události. Registrace se provádí skrz obecné rozhraní `ICoreProxy` pomocí metody `subscribeForEvents(...)`, která registraci deleguje na patřičné metody událostního subsystému. Při registraci se vyžaduje specifikace typu události, kterou chce daný objekt odebírat a cíl, objekt implementující rozhraní `IEventListener`, kde se budou tyto události zasílat. Událostní subsystém si udržuje tabulky registrací pro jednotlivé typy základních i uživatelem definovaných událostí. Jakmile prostředník přijme nějakou událost, zjistí pomocí těchto tabulek cílové objekty, kterým tuto událost doručit, a patřičně je propaguje dále. Pokud žádný objekt nevyžaduje doručení daného typu události, je objekt reprezentující událost automaticky uvolněn.

6.3.5 Subsystém rozšíření

Subsystém rozšíření zajišťuje externí, a částečně interní, modularitu programu. Modularita je založena na principu zásuvných modulů a definicích obecných rozhraní pro komunikaci s těmito moduly. Zásuvné moduly mohou být jak interní, tak externí.

Interní moduly jsou součástí samotného kódu verifikátoru a tedy vždy k dispozici. Tyto moduly se registrují, a často i využívají, při samotné inicializaci a prvotní konfiguraci verifikátoru a musí být tedy zajištěna jejich přítomnost.

Externí moduly jsou automaticky načteny a zaregistrovány až za chodu programu po inicializaci subsystému rozšíření. Jsou reprezentovány formou dynamických knihoven (soubory **.dll*) nebo sdílených objektů (soubory **.so*), v závislosti na operačním systému.

Subsystém rozšíření je reprezentován třídou `ExtensionsManager`, která zajišťuje kompletní správu interních i externích zásuvných modulů. Zároveň je tato třída implementována podle návrhového vzoru *factory*[3] a plní úlohu správce vytvořených instancí tříd z těchto modulů.

Než je možné jednotlivé zásuvné moduly používat, je potřeba je načíst. Subsystém rozšíření prohledává adresáře dle načtené konfigurace, ve výchozím nastavení jsou prohledány adresáře *modules* a *plugins* v aktuálním pracovním adresáři verifikátoru. Vyhledá-

vají se vždy pouze validní soubory zásuvných modulů, tedy soubory s příponou **.dll* nebo **.so*, ostatní jsou ignorovány. Jelikož práce se souborovým systémem vyžaduje volání specifických funkcí použitého operačního systému, jsou tyto operace abstrahovány pomocí třídy `Directory`, jenž zastřešuje rozdílnost implementace na jednotlivých operačních systémech. Tato třída poskytuje metody pro prohledání určitého adresáře a vrací seznam všech potenciálních zásuvných modulů v tomto adresáři.

Protože struktury reprezentující dynamickou knihovnu a sdílený objekt se obecně liší a funkce pro jejich obsluhu jsou opět závislé na operačním systému, je i zde vytvořena jejich abstraktní reprezentace ve formě třídy `Library`. Tato třída zapouzdřuje nativní struktury a poskytuje potřebné metody pro obsluhu jako je načtení a uvolnění knihovny nebo vyhledávání symbolů. Načtení zásuvného modulu se skládá ze dvou kroků. Nejprve se načte nalezený modul do paměti a následně se vyhledá vstupní bod tohoto modulu. Vstupní bod tvoří funkce `zetavPluginEntry()`. Tato funkce je definována jako funkce jazyka C a symbol, který ji v zásuvném modulu identifikuje je tedy shodný s jejím názvem. Pokud se subsystému rozšíření nepodaří v zásuvném modulu tento symbol nalézt, není modul validním rozšířením a je ignorován. Pokud je symbol nalezen, je odkaz na něj převeden na ukazatel na funkci, čímž ve skutečnosti je.

Funkce `zetavPluginEntry()` vrací ukazatel na objekt implementující rozhraní `IPlugin`. Toto rozhraní definuje metody, jenž poskytují důležité informace o zásuvném modulu, kterými jsou:

- **SID** - Řetězec identifikující konkrétní zásuvný modul a objekt, který tento modul vytváří. Tento řetězec lze získat pomocí metody `sid()`.
- **Implementované rozhraní** - Řetězec identifikující obecné rozhraní, které implementuje objekt vytvářený zásuvným modulem. Tento řetězec lze získat pomocí metody `implements()`.
- **Vyžadovaná data** - Řetězec popisující vstupní data, která potřebuje objekt vytvářený zásuvným modulem ke své činnosti. Tato informace má význam hlavně u verifikačních modulů. Tento řetězec lze získat pomocí metody `requires()`.
- **Produkovaná data** - Řetězec popisující výstupní data, která produkuje objekt vytvářený zásuvným modulem jako výsledek své činnosti. Tato informace má význam hlavně u verifikačních modulů. Tento řetězec lze získat pomocí metody `provides()`.

Kromě výše popsaných metod poskytuje rozhraní `IPlugin` ještě poslední metodu `create()`, jenž slouží k vytvoření výše zmíněného objektu vytvářeného zásuvným modulem. Tato metoda vrací ukazatel na obecné bázové rozhraní `Interface`, ze kterého dědí všechna rozhraní verifikátoru, včetně rozhraní `IPlugin`. Konkrétní implementované rozhraní lze zjistit pomocí metody `implements()`. Tento přístup umožňuje tzv. pozdní zavádění objektů zásuvných modulů. Při načítání modulu dojde pouze k vytvoření instance třídy implementující rozhraní `IPlugin`, která je obecně velmi malá. Samotný objekt zásuvného modulu je vytvořen teprve tehdy, až verifikátor, nebo některá jeho část, tento objekt vyžaduje pro svou činnost.

Verifikátor rozlišuje dva typy zásuvných modulů:

- **Zásuvný modul (*plugin*)** - Standardní zásuvný modul jehož vytvářené objekty implementují jakékoliv rozhraní zděděné z obecného bázového rozhraní `Interface`. Hlavním přínosem těchto modulů je zajištění nezávislosti na konkrétní implementaci.

Verifikátor, nebo jeho součásti, využívájí pouze abstraktní metody určitého rozhraní a nezajímá je konkrétní realizace. Uživatel si pak jednoduše může upravit konkrétní chování výměnou modulu za jiný. Příkladem tohoto typu rozhraní je například rozhraní `IConfigReader` pro načítání konfigurace nebo `ILogger` pro zápis do protokolu událostí.

- **Verifikační modul (*module*)** - Speciální zásuvný modul realizující určitou část verifikačního procesu. Implementuje pseudorozhraní `Module`. Tato třída je rozhraním, ale zároveň také plnohodnotnou komponentou verifikátoru. Může tedy využívat událostní subsystém i veškeré služby jádra. Každý verifikační modul musí implementovat metodu `run(...)`, jenž spouští chod modulu. Tato metoda je volána verifikátorem v průběhu verifikačního procesu a jsou ji předány vyžadovaná vstupní data. Vstupní data jsou reprezentována strukturou `TInputData`, která obsahuje základní konfigurační data modulu a výstupní data vytvořená předchozím volaným modulem. Po ukončení činnosti vrací tato metoda vyprodukovaná výstupní data reprezentována strukturou `TOutputData`.

Jak již bylo zmíněno dříve, subsystém rozšíření neplní pouze funkci správy zásuvných modulů, ale také spravuje a monitoruje vytvořené objekty těchto modulů. Vytváření objektů je možné skrz rozhraní `ICoreProxy` pomocí metody `createInstance(...)`, která deleguje tento požadavek patřičné metodě subsystému rozšíření. Vytvářený objekt je identifikován pomocí *SID* zásuvného modulu, který jej vytváří. Subsystém umožňuje vytvářet dva typy objektů zásuvných modulů:

- **Standardní objekt** - Standardní instance třídy, po vytvoření je odkaz na tento objekt zařazen do seznamu instancí a při uvolnění je tento objekt automaticky smazán.
- **Sdílený objekt** - Speciální typ objektu, kdy více komponent systému může využívat stejnou instanci třídy. Pokud si nějaká komponenta vyžádá vytvoření sdíleného objektu a ten již existuje, nedojde k vytvoření nové instance, nýbrž k použití již dříve vytvořeného objektu a komponentě bude vrácen odkaz na tento objekt. Sdílené objekty jsou výhodné v situacích, kdy konkrétní implementace nějakého rozhraní má určitým způsobem neměnný vnitřní stav, nebo je úplně bezestavová. Místo několika instancí lze pak jednoduše využít již vytvořenou a ušetřit tak prostředky verifikátoru. Při uvolnění se pouze sníží počítadlo referencí, teprve jakmile toto počítadlo klesne na nulu je sdílený objekt smazán.

Veškeré objekty vytvořené subsystémem rozšíření jsou plně v jeho režii. Jakmile dojde k rušení objektu reprezentujícího tento subsystém, jsou zároveň s ním uvolněny i veškeré vytvořené objekty, jak standardní, tak sdílené. Komponenty, které využívají tyto objekty, se nemusí starat o jejich korektní uvolnění. Je ovšem důležité, aby všechny třídy, jejichž objekty se vytvářejí, implementovaly virtuální destruktor, jinak nemusí dojít k uvolnění všech prostředků rušeného objektu.

6.3.6 Protokolovací subsystém

Protokolovací subsystém je reprezentován třídou `LogManager`, která centralizuje zápisy do protokolů událostí. Tato třída se během své inicializace zaregistruje u událostního subsystému jako odběratel protokolových událostí, základních událostí typu `LOG` reprezentovaných třídou `LogEvent`. Objekt této události obsahuje zprávu, jenž se má zaznamenat do

protokolu událostí, a také informaci o důležitosti této zprávy, tzv. úroveň protokolování. Úroveň protokolování je vyjádřena konstantami od 1 do 7 s následujícím významem:

- 1. úroveň, `ERROR`, je vyhrazena pro chyby, které vylučují další korektní běh programu.
- 2. úroveň, `WARNING`, je vyhrazena pro varování, které mohou ovlivnit správný běh nebo konfiguraci programu, případně zásuvného modulu, ale často mohou být ignorovány.
- 3. - 7. úroveň, `INFORMATION`, `FINE`, `FINER`, `FINEST` a `DEBUG`, jsou vyhrazeny pro zprávy informačního charakteru. Tyto zprávy mají za úkol informovat uživatele o průběhu práce verifikátoru a jednotlivých verifikačních modulů. Poslední 7. úroveň slouží k výpisu ladících informací a neměla by se používat.

Protokolovací subsystém dělí události do tří kategorií - chyby, varování a informační zprávy. Každá kategorie je zaznamenávána pomocí vlastního protokolovacího objektu. Protokolovací objekt je instance třídy implementující rozhraní `ILogger`, jenž definuje metody pro zápis do protokolu událostí. Verifikátor obsahuje celkem tři interní zásuvné moduly pro protokolování, které zaznamenávají zprávy na standardní výstup, standardní chybový výstup nebo do specifikovaného souboru. Další mohou být dodány ve formě externích zásuvných modulů. Při přijetí protokolové události se k obsažené zprávě nejprve připojí řetězec identifikující objekt, jenž zaslal zpracovávanou událost. Následně se použije příslušný protokolovací objekt pro zápis výtvořené zprávy do cílového protokolu událostí. Protokolovací objekt ještě připojí ke zprávě časovou značku zápisu a provede záznamenání zprávy.

Jelikož ve velkém množství případů je vytváření zprávy protokolovací události značně komplikované, často je potřeba provádět konkatenace jednotlivých částí zprávy, formátování, konverze různých datových typů do textové podoby apod., je zde zvolen přístup filtrování na úrovni odesilatele. Všechny komponenty si mohou zjistit nastavení úrovně protokolování pomocí metody `getLogLevel()` rozhraní `ICoreProxy` a na základě této získané hodnoty rozhodnout, zda protokolovací událost vytvořit či nikoliv. Otestování jedné celočíselné hodnoty je podstatně rychlejší než složitá konstrukce zprávy, která by nakonec stejně byla zahozena. Verifikátor navíc obsahuje sadu maker pro usnadnění protokolování, které automatizují a zjednodušují celý proces. Patří zde makra `LOG_ERROR(...)`, `LOG_WARNING(...)`, `LOG_INFO(...)` a `LOG_DEBUG(...)`. Tyto makra přijímají jako parametr text zprávy s podporou konverze a konkatenace pomocí přetěžování operátoru `<<`.

6.3.7 Konfigurační subsystém

Konfigurační subsystém je reprezentován třídou `ConfigManager`, která spravuje konkrétní zdroje konfigurace. K jednotlivým zdrojům konfigurace se přistupuje skrz obecné rozhraní `IConfigReader`, které definuje metody pro načtení konfigurace a získání hodnoty určité položky této konfigurace. Verifikátor obsahuje jednu konkrétní implementaci tohoto rozhraní, jenž využívá XML souborů pro uložení konfigurace. Tento objekt je vytvářen interním zásuvným modulem.

Identifikace položky konfigurace je hierarchická. Název položky se zapisuje ve formátu `cesta/k/položce/konfigurace`, způsob uložení konfigurace je plně na konkrétní třídě implementující rozhraní `IConfigReader`, stejně tak možný převod do interní reprezentace během načítání nebo možnosti kešování. Z hlediska získávání hodnot položek konfigurace musí konkrétní třída implementovat pouze metodu `getRawValue(...)`, jenž vrací hledanou

hodnotu ve formě řetězce nebo řetězec s výchozí hodnotou, pokud není hledaná položka konfigurace nalezena. Navrácení hodnot jiných datových typů, nežli řetězec, je realizováno formou šablonovaných metod `getValue(...)`, které automaticky řeší konverze na požadované datové typy pomocí přetíženého operátoru `>>`. Standardní datové typy toto přetížení obsahují, uživatelem definované datové typy musí pro správnou funkčnost konverze toto přetížení implementovat.

Konfigurační subsystém poskytuje identickou šablonovanou metodu `getValue(...)` pro přístup k hodnotám položek konfigurace. Subsystém ovšem prohledává veškeré spravované zdroje konfigurací, které byly zaregistrovány pomocí metody `addConfigReader(...)`. Spravované zdroje jsou uloženy v seznamu a při hledání se postupně prohledávají v pořadí jejich výskytu v tomto seznamu. Vrácena je hodnota první nalezené položky konfigurace se zadaným názvem. Pokud tedy hledanou položku obsahuje více zdrojů konfigurace, je vrácena hodnota dříve zaregistrovaného zdroje. Někdy je ovšem potřeba, aby právě přidávaný zdroj konfigurace přepsal konfigurace předchozí, tedy aby byl tento zdroj prohledán dříve. K tomu slouží parametr `forceMaxPriority`. Tento parametr vynutí přidání zdroje konfigurace na počátek seznamu zdrojů a tedy vynutí prioritní prohledání tohoto zdroje před již přítomnými zdroji.

6.3.8 Komunikační subsystém

Komunikační subsystém je aktivován pouze v případě, že je verifikátor spuštěn v tzv. *módu serveru.* V tomto módu verifikátor naslouchá na příchozí požadavky připojeného klienta a vyřizuje je. Konceptuálně je tento režim navržen pro připojení pouze jediného klienta, nejčastěji zastávajícího roli grafického rozhraní pro zjednodušení použití verifikátoru. I přesto, že se tento stav může jevit jako značně limitující, není nikterak v rozporu s užitím verifikátoru. Je potřeba si uvědomit, že proces verifikace je obecně náročný a zdlouhavý výpočet, jenž využívá téměř všechny dostupné zdroje hostitelského počítače a je tedy krajně nevhodné spouštět více takovýchto výpočtů zároveň. V praxi jsou často požadavky zcela opačné, tedy, místo spouštění více verifikačních procesů na jediném počítači, paralelizovat jeden výpočet na velké množství počítačů, případně specializovaných procesorů, s cílem ještě více tento výpočet urychlit.

Komunikační subsystém je jediným subsystémem, který není reprezentován konkrétní třídou. Jádro obsahuje pouze referenci na objekt implementující rozhraní `ICommunicator`, jenž definuje metody pro inicializaci a spuštění serveru. Jelikož *mód serveru* je jakýmsi rozšířením, které nemusí být vůbec využito pro potřeby verifikace, neobsahuje verifikátor žádný interní zásuvný modul implementující toto rozhraní. Konkrétní objekt, který realizuje komunikaci, musí být tedy dodán ve formě externího zásuvného modulu.

Pokud je verifikátor spuštěn v *módu serveru*, pak se po inicializaci jádra a základních subsystémů navíc aktivuje komunikační subsystém a plánovač. Je důležité si ale uvědomit, který objekt má nyní na starosti řízení verifikátoru. Zatímco ve standardním módu je činnost verifikátoru řízena jádrem na základě načtené konfigurace, zde je situace zcela odlišná. Po inicializaci komunikačního subsystému je spuštěn server a jádro mu předá kompletní kontrolu nad řízením další činnosti verifikátoru. Verifikace je nyní ovládána připojeným vzdáleným klientem na základě zaslaných požadavků. Vzniká zde tedy potřeba zpětné komunikace serveru s jádrem a přístup k potřebným službám jádra.

Během inicializace objektu implementujícího rozhraní `ICommunicator` mu jádro předá, skrz toto rozhraní, odkazy na proxy jádra a událostní subsystém. Objekt tedy získá možnost využití komunikace založené na zasílání událostí a zároveň také přístup ke službám jádra.

Zpětná komunikace s jádrem tak nyní může jednoduše probíhat zasíláním událostí. Server zpracovává požadavky přijaté od vzdáleného klienta a transformuje tyto požadavky na volání služeb jádra nebo na příkazy, speciální události reprezentované třídou `CommandEvent`, jenž poté zasílá jádru, přesněji plánovači jádra. Plánovač jádra pak po obdržení tyto příkazy vykonává a výsledek jejich provedení zasílá zpět ve formě objektu třídy `ResultEvent`.

6.3.9 Modul pro zpracování omezených RTL formulí

Modul pro zpracování omezených RTL formulí (*Restricted RTL formula parser*) je jeden z verifikačních modulů. Realizuje převod omezených RTL formulí v textové podobě do interní reprezentace, se kterou poté pracují další moduly. Tento modul vyžaduje jako vstupní data textový soubor obsahující omezené RTL formule popisující zadaný systém nebo jeho testované vlastnosti. Jako výstupní data poté produkuje seznam omezených RTL formulí v interní reprezentaci spolu s přiloženou překladovou tabulkou.

Modul nezpracovává vstupní soubor jako celek, načte vždy jednu formuli, jenž je následně zpracována a převedena do své interní reprezentace. Pokud obsahuje vstupní soubor více formulí, musí být tyto formule navzájem odděleny středníkem. Pro zpracování načtených formulí využívá modul syntaktický analyzátor vytvořený generátorem syntaktických analyzátorů ANTLR[1].

ANTLR generuje kód syntaktického analyzátoru na základě předložené bezkontextové gramatiky. Tato gramatika musí být zapsána ve formě rozšířené Backus–Naurovy formy (EBNF, *Extended Backus–Naur Form*), jenž obsahuje, jak lexikální, tak syntaktická, pravidla dané gramatiky. Použitá bezkontextová gramatika zde popisuje jazyk tvořící omezené RTL formule a zápis této gramatiky ve formě EBNF lze nalézt v sekci 7.2 v přílohách. Vytvořený syntaktický analyzátor je typu LL(*), což je rozšíření typu LL(k).

ANTLR umožňuje vytvářet kód syntaktického analyzátoru ve velké řadě programovacích jazyků jako je Java, C nebo C#. Neumí sice generovat kód přímo v jazyce C++, ovšem vytvořený kód v jazyce C je plně kompatibilní s C++. Vygenerovaný kód je tvořen dvěma částmi:

- **Lexikální analyzátor (*scanner*)** - Zpracovává vstupní text. Přetváří úseky textu, podle jejich významu, na tzv. *tokeny*, jenž jsou jakousi abstraktní reprezentací určitého úseku textu. Význam každého úseku textu je určen na základě lexikálních pravidel použité gramatiky. Lexikální analyzátor tedy provádí určitý druh předzpracování vstupních formulí, při kterém je transformuje na posloupnost tokenů, tzv. *token stream*.

- **Syntaktický analyzátor (*parser*)** - Ověřuje správnost formulí. Analyzuje vstupní posloupnost tokenů a na základě syntaktických pravidel použité gramatiky se snaží vytvořit tzv. derivační strom. Existence derivačního stromu pro analyzovanou formuli znamená, že tato formule může být generována gramatikou omezených RTL formulí a je tedy syntakticky platnou omezenou RTL formulí. V opačném případě obsahuje analyzovaný text syntaktickou chybu.

Jelikož matematický zápis omezených RTL formulí není vhodný pro strojové zpracování, přijímá modul, přesněji jeho syntaktický analyzátor, pouze formule ve formátu ASCII. Je tedy potřeba zavést mapování matematických symbolů, jenž se mohou vyskytnout v těchto formulích, na znaky přípustné ve formátu ASCII. Protože z EBNF lze jen obtížně vyvodit

[1] Více informací na http://www.antlr.org/

Symbol	Matematický zápis	Programový zápis	Interní reprezentace
Konstanta	$\pm$ Konstanta	`[-]?[0-9]+`	třída `Constant`
Proměnná	Identifikátor	`[a-z][a-zA-Z0-9]*`	třída `Variable`
Akce	Identifikátor	`[A-Z][a-zA-Z0-9]*`	třída `Action`
Externí akce	Ω	`#`	třída `Action`, typ `EXTERNAL`
Spouštěcí akce	$\uparrow$	`^`	třída `Action`, typ `START`
Koncová akce	$\downarrow$	`$`	třída `Action`, typ `STOP`
Přechodová akce	*žádný*	`%`	třída `Action`, typ `TRANSITION`
Predikát $<$	$<$	`<`	třída `BinaryPredicate`, typ `LESS`
Predikát $\leq$	$\leq$	`=<`	třída `BinaryPredicate`, typ `LEQ`
Predikát $=$	$=$	`=`	třída `BinaryPredicate`, typ `EQ`
Predikát $\geq$	$\geq$	`>=`	třída `BinaryPredicate`, typ `GEQ`
Predikát $>$	$>$	`>`	třída `BinaryPredicate`, typ `GREATER`
Funkce $+$	$+$	`+`	třída `BinaryFunction`, typ `PLUS`
Funkce $-$	$-$	`-`	třída `BinaryFunction`, typ `MINUS`
Negace	$\neg$	`not` *nebo* `!`	třída `Connective`, typ `NEG`
Konjunkce	$\wedge$	`and` *nebo* `&` *nebo* `/\`	třída `Connective`, typ `AND`
Disjunkce	$\vee$	`or` *nebo* `|` *nebo* `\/`	třída `Connective`, typ `OR`
Implikace	$\rightarrow$	`imply` *nebo* `->`	třída `Connective`, typ `IMPLY`
Pro všechny	$\forall$	`forall` *nebo* `V`	třída `Quantifier`, typ `FORALL`
Existuje	$\exists$	`exists` *nebo* `E`	třída `Quantifier`, typ `EXISTS`
Funkce výskytu	@	`@`	třída `OccurenceFunction`

Tabulka 6.1: Mapování symbolů omezených RTL formulí na programový zápis

použité mapování, je zde toto mapování přehledně shrnuto ve formě tabulky 6.1. Tato tabulka zachycuje několik důležitých aspektů zápisu vstupních formulí:

- Pro některé symboly je přípustných hned několik zápisů, uživatel si tedy může zvolit formu zápisu, která mu nejvíce vyhovuje.
- Názvy proměnných i akcí jsou zastoupeny identifikátory tvořenými posloupností alfanumerických znaků. Syntaktický analyzátor rozlišuje proměnné od akcí na základě prvního písmena identifikátoru. Názvy akcí musí vždy začínat velkým písmenem, zatímco proměnné malým.
- Je potřeba dbát na správné oddělení jednotlivých symbolů bílými znaky. Nepřítomnost bílého znaku mezi dvěma symboly může změnit význam určitého úseku formule. Například zapsání $\exists x$ jako `Ex` místo `E x` způsobí, že tento úsek bude interpretován jako název akce místo existenčního kvantifikátoru s navázanou proměnnou x.

Chyby způsobené nesprávným zápisem omezené RTL formule v drtivé většině případů odhalí sám syntaktický analyzátor. Problematičtější jsou chyby sémantické, vzniklé nesprávným přepisem matematického vyjádření formule do programového zápisu. I když veškeré sémantické detaily jsou vyjádřeny pravidly syntaktického analyzátoru ve formě EBNF, velké řadě uživatelů nic neříkají. Uživateli pro zápis vstupních formulí úplně postačuje mít na paměti následující:

- Symboly $\wedge$, $\vee$ a $\rightarrow$ mají stejnou prioritu. Je tedy důležité dbát na správné uzávorkování formulí. Pokud zapisovaná formule například vyjadřuje, že $A \wedge B$ implikuje

$C \vee D$, pak ekvivalentní programový zápis bude `(A /\ B) -> (C \/ D)`. Při absenci závorek by byl tento zápis ekvivalentní formuli `A /\ (B -> (C \/ D))`.

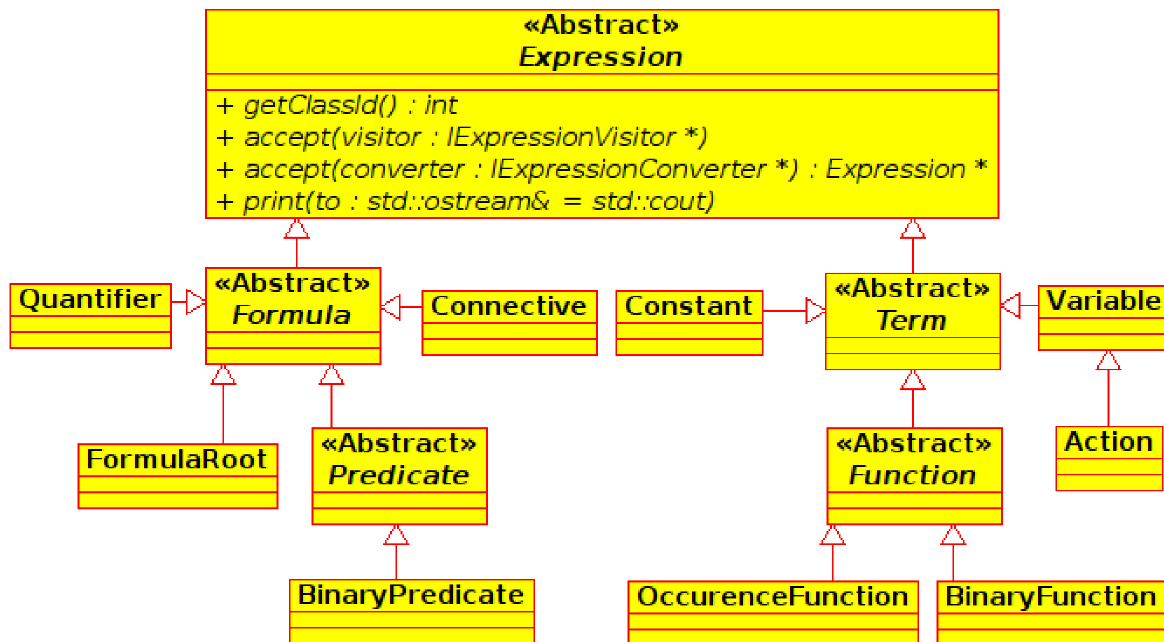
- Rozsah platnosti kvantifikátorů je omezen na nejbližší formuli. Pokud za kvantifikátorem následuje další kvantifikátor, sdílejí tyto kvantifikátory stejný rozsah platnosti. Pokud za kvantifikátorem následuje závorka, je rozsah platnosti rozšířen na veškeré formule obsažené v této závorce. Zápis `V x E y (A /\ B)` tedy vyjadřuje, že všechny výskyty proměnných x a y jsou vázané v obou formulích A i B. Bez patřičného uzávorkování by se rozsah platnosti obou kvantifikátorů vztahoval pouze na formuli A a proměnné x a y ve formuli B by měly volný výskyt.

Jakmile syntaktický analyzátor ověří správnost vstupní formule, je tato formule převedena do své interní reprezentace. ANLTR poskytuje dva způsoby, jak přetvořit zpracovanou formuli do požadované formy.

První možností je vytvořit tzv. AST (*Abstract Syntax Tree*). AST je určitá forma derivačního stromu, který vytváří sám syntaktický analyzátor. Informace potřebné pro jeho konstrukci jsou součástí EBNF. Každé obsažené pravidlo může mít definováno mapování na specifický objekt, jenž se vytvoří, a přidá do AST, při použití odpovídajícího pravidla. Mapování nemusí být úplné, mohou tedy existovat pravidla negenerující žádný uzel stromu, tyto pravidla často slouží pouze k ověřování správné syntaxe formule. Zatímco derivační strom tedy detailně popisuje postup, který vedl k ověření formule, AST často obsahuje, a poskytuje, pouze informace o již ověřených datech formule. AST je tedy vhodný pro následné zpracování a vytvoření odpovídající interní reprezentace těchto dat. Výhodou AST je jednoduchost následného zpracování, které někdy ani není potřeba, pokud se jako interní reprezentace použije přímo AST. Nevýhodou může být nižší efektivita, jak po stránce časové, tak prostorové. Následná potřeba přetvoření AST do jiné formy znamená druhé zpracování stejných dat. Využití AST jako interní reprezentace zase vyžaduje, aby veškeré použité objekty byly zděděny ze speciálních bázových tříd, jenž tvoří základ AST. Tyto třídy obsahují data potřebná pro konstrukci AST, ale zbytečná pro potřeby verifikačních modulů. Jelikož těchto objektů je vytvářeno obrovské množství, může tato skutečnost značně zvýšit paměťové nároky verifikace.

Druhou možností je generovat interní reprezentaci přímo. ANTLR dovoluje přidávat k definovaným pravidlům obslužný kód, který je vykonán při použití tohoto pravidla. V tomto kódu lze tedy rovnou vytvářet požadovanou interní reprezentaci, což činí tento přístup velice efektivní. Nevýhodou je podstatně složitější implementace. Obslužný kód je totiž volán až na konci metod, jenž reprezentují jednotlivá pravidla. Na jedné straně to znamená, že generování interní reprezentace začne teprve tehdy, jakmile je vytvořen celý derivační strom a formule je tedy validní. Na druhé straně to ovšem znamená, že interní reprezentace je generována obráceně vzhledem k vytváření derivačního stromu. Zatímco proces konstrukce derivačního stromu postupně dekomponuje celou formuli na úseky, které následně ověřuje, kód musí vytvářet odpovídající objekty jako samostatné celky, jenž musí být následně spojovány do vyšších celků reprezentujících predikáty, funkce nebo atomické či složené formule.

Syntaktický analyzátor využívá druhého přístupu, hlavně pro jeho vysokou efektivitu. Formule je interně reprezentována jako n-ární strom složený z výrazů, objektů tříd zděděných ze třídy `Expression`. Koncepce tříd jednotlivých typů výrazů je založena na návrhovém vzoru *composite*[3]. Kompletní hierarchie těchto tříd je zobrazena na obrázku 6.2, mapování symbolů, a jiných částí, formule na odpovídající typ objektu lze poté nalézt v tabulce 6.1.



Obrázek 6.2: Hierarchie tříd interní reprezentace omezených RTL formulí

Jelikož vložený obslužný kód pravidel zpracovává úseky v opačném pořadí, než je volání pravidel, je potřeba uchovávat si dříve vytvořené objekty pro následné zpracování. K tomuto účelu slouží syntaktickému analyzátoru třída `Storage`. Tato třída obsahuje sadu zásobníků pro uložení již vytvořených objektů a zajišťuje převod názvů proměnných a akcí na číselné identifikátory. Pokud bude syntaktický analyzátor zpracovávat například formuli $A \wedge B$, pak nejdříve vytvoří odpovídající derivační strom. Následně dojde k vykonání obslužného kódu v opačném pořadí, než je volání pravidel při konstrukci derivačního stromu (dochází vlastně k vynořování z rekurze způsobené sadou metod reprezentujících jednotlivá pravidla). Nejprve se tedy vytvoří objekt reprezentující formuli A a uloží se na adekvátní zásobník, stejný postup se zopakuje i v případě formule B. Poté dojde k návratu do metody zpracovávající spojení dvou formulí pomocí logické spojky, kde se vytvoří nový objekt reprezentující spojku $\wedge$. Jelikož v tomto čase je již provedeno ověření správnosti formule, ví analyzátor, že se vynořil z metod, které zpracovaly obě spojované formule, a že tyto formule musí nalézt na odpovídajícím zásobníku. Analyzátor tedy vyjme obě vytvořené formule ze zásobníku a vytvoří z nich požadovanou složenou formuli, neboli přiřadí obě formule objektu reprezentujícímu spojku $\wedge$. Jakmile je zpracována celá formule, najde ji modul pro zpracování omezených RTL formulí uloženou na jednom ze zásobníků.

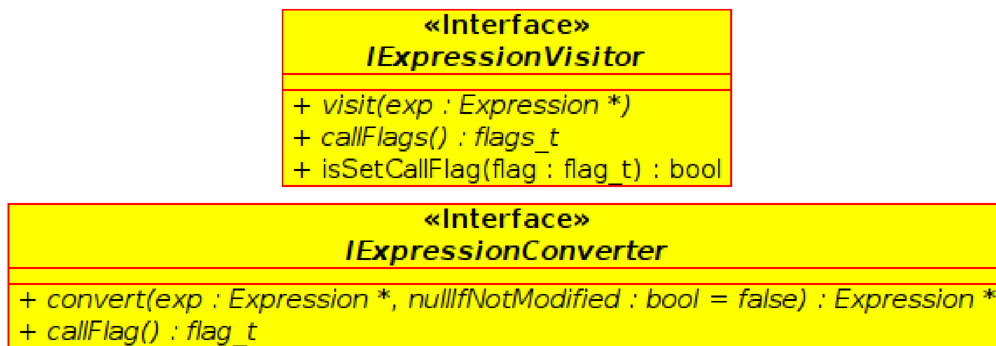
Posledním problémem, jenž je potřeba vyřešit, z hlediska interní reprezentace formulí je identifikace názvů proměnných a akcí. Z výpočetního hlediska je krajně nevhodné použít pro identifikaci přímo názvy proměnných a akcí. Pro potřeby následné verifikace stačí pouze zajistit, aby zvolená identifikace umožnovala porovnat dva názvy, zda jsou stejné, a získat typ entity daného názvu. Oba tyto požadavky lze jednoduše zajistit i s použitím celočíselných identifikátorů. Veškeré názvy jsou tedy převáděny na unikátní celočíselný identifikátor typu `id_t` pomocí objektu implementujícího rozhraní `ITranslationTable`. Objekt implementující toto rozhraní musí zajistit, že stejnému názvu bude vždy přidělen tentýž identifikátor.

Toto rozhraní poskytuje dvě možnosti převodu názvu na identifikátor. Metoda `getId(...)` jednoduše převede zadaný název na identifikátor, zatímco metoda `generateId(...)` použije zadaný název pouze jako prefix pro nově generovaný unikátní název a následně tomuto názvu přiřadí unikátní identifikátor.

6.3.10 Modul pro konverzi omezených RTL formulí

Modul pro konverzi omezených RTL formulí (*Restricted RTL formula converter*) je dalším z verifikačních modulů. Realizuje transformaci omezených RTL formulí na formuli v diferenční logice, přesněji v logice označované jako QF_UFIDL[2]. Navíc všechny atomické formule, nerovnosti z pohledu diferenční logiky, obsažené v této formuli normalizuje do formy $funkce\ výskytu \pm celočíselná\ konstanta \leq funkce\ výskytu$, která je vyžadována algoritmem pro konstrukci grafu omezení. Modul vyžaduje jako vstupní data omezené RTL formule ve formě interní reprezentace a produkuje jedinou QF_UFIDL formuli s doplněnou překladovou tabulkou.

Celý proces konverze je založen na transformačních objektech, jenž provádějí potřebné úpravy jednotlivých omezených RTL formulí. Implementace vychází z návrhového vzoru *visitor*[3], který definuje rozhraní pro interakci mezi transformačními a zpracovávanými objekty. Všechny výrazy, objekty tříd zděděných ze třídy `Expression` (viz. 6.2), implementují metodu `accept(...)`, jenž umožňuje těmto výrazům přijmout libovolný transformační objekt. Jelikož je omezená RTL formule reprezentována formou stromu, je vhodné zajistit propagaci transformačních objektů celým tímto stromem. K tomu lze ovšem s výhodou využít navržené hierarchie výrazů, implementující návrhový vzor *composite*[3], a integrovat propagaci přímo do metody `accept(...)`. Transformační objekty musí vždy implementovat jedno z rozhraní `IExpressionVisitor` či `IExpressionConverter`, které poskytuje metody pro zpracování výrazů, definice těchto rozhraní je zobrazena na obrázku 6.3.



Obrázek 6.3: Rozhraní pro konverzní operace nad formulemi

Rozhraní `IExpressionVisitor` odpovídá standardní implementaci návrhového vzoru *visitor*[3]. Definuje abstraktní metodu `visit(...)`, která přijímá obecný výraz ke zpracování. Typ výrazu, jenž byl předán této metodě, lze zjistit pomocí metody `getClassId()`. Metoda `visit(...)` je volána samotným výrazem na sebe sama v rámci obsluhy transformačního objektu metodou `accept(...)` a může být volána i opakovaně. Každý transformační objekt si může zvolit, kdy má být volán, specifikací příslušných příznaků. Výraz

[2] QF_UFIDL (*Quantifier-Free Integer Difference Logic with Uninterpreted Functions*) označuje celočíselnou diferenční logiku s neinterpretovanými funkcemi neobsahující kvantifikátory.

určuje dobu volání na základě vektoru těchto příznaků, který vrací metoda `callFlags()`. Podporované příznaky jsou:

- `BEFORE_CHILDREN` - Nastavení tohoto příznaku zajistí volání transformačního objektu před jeho propagací případným synovským výrazům zpracovávaného výrazu.
- `AFTER_CHILDREN` - Nastavení tohoto příznaku zajistí volání transformačního objektu po jeho propagaci případným synovským výrazům zpracovávaného výrazu.

Oba tyto příznaky mohou být nastaveny současně, transformační objekt tedy může být volán před i po jeho propagaci synovským výrazům. Pokud zpracovávaný výraz neobsahuje žádné synovské výrazy, je nastavení těchto příznaků ignorováno a transformační objekt vždy ihned volán. Pro testování, zda je určitý z příznaků nastaven, lze využít metody `isSetCallFlag()`. Tento typ transformačního objektu je využíván hlavně pro sběr informací a detailnější analýzu částí formule.

Předchozí typ transformačního objektu trpí jedním vážným nedostatkem. Jakmile vyžaduje úprava zrušení výrazu nebo jeho náhradu jiným typem, nelze tyto změny promítnout do stromové struktury formule. Transformační objekt se totiž nedostane k nadřazeným uzlům tohoto stromu. Řešením tohoto problému je rozhraní `IExpressionConverter`. Toto rozhraní definuje abstraktní metodu `convert(...)`, která, narozdíl od metody `visit(...)`, vrací výsledek provedených úprav. Výsledkem může být předaný, jiný nebo nově vytvořený výraz. Tento výsledný výraz je poté připojen do stromové struktury formule na místo předchozího výrazu. Toto rozšíření ale způsobuje vážné komlikace z hlediska propagace transformačních objektů tohoto typu. Důvodem je volání metody `convert(...)` z metod upravovaného výrazu. Pokud je během úprav tento výraz smazán a navrácen výsledek, dojde k návratu do metody `accept(...)` tohoto smazaného výrazu a implementace musí s tímto stavem počítat. Výrazy řeší tento problém omezením možností propagace a opětovného volání transformačního objektu. Způsob a čas volání transformačního objektu je dán zvoleným příznakem, který výraz zjišťuje pomocí metody `callFlag()`. Podporovány jsou celkem tři typy příznaků:

- `DO_NOT_PROPAGATE` - Tento příznak způsobí volání transformačního objektu ihned po jeho přijetí výrazem a okamžité vrácení výsledku. Transformační objekt nebude propagován dále možným synovským výrazům.
- `PROPAGATE_IF_NOT_MODIFIED` - Tento příznak způsobí volání transformačního objektu ihned po jeho přijetí výrazem. Pokud transformační objekt nahradil výchozí výraz jiným, dojde k okamžitému vrácení výsledku. Jesliže však byl výraz pouze upraven nebo vůbec nebyl změněn, je transformační objekt propagován dále synovským výrazům. K rozpoznání, zda vrácený výsledek odpovídá výchozímu výrazu, slouží příznak `nullIfNotModified` metody `convert(...)`. Pokud je nastaven, pak transformační objekt vrací výsledný výraz pouze v případě, že došlo k nahrazení původního výrazu jiným, jinak vrací `NULL`. To umožňuje metodě `accept(...)` otestovat, zda výraz pořád existuje a pokud ano, provést následnou propagaci transformačního objektu synovským výrazům.
- `PROPAGATE_FIRST` - Tento příznak vynutí okamžitou propagaci transformačního objektu synovským výrazům. Po dokončení propagace je teprve volán transformační objekt.

Specifikace více příznaků zároveň zde není podporována, transformační objekt musí tedy zvolit pouze jediný způsob volání.

Jelikož prakticky veškeré úpravy omezených RTL formulí vyžadují zásahy do stromové reprezentace těchto formulí, využívá modul výhradně druhý typ transformačních objektů. Tyto objekty ovšem často využívají transformační objekty prvního typu pro kolekci důležitých informací potřebných pro prováděné úpravy. Mezi pomocné transformační objekty prvního typu patří:

- `FreeVariableFinder` - Zjišťuje počet volných výskytů zadané proměnné ve formuli.
- `FreeVariableCollector` - Vyhledá veškeré volné výskyty zadané proměnné a vytvoří seznam odkazů na výrazy reprezentující tyto proměnné.
- `QuantifiersCollector` - Vyhledá všechny kvantifikátory ve formuli a zkonstruuje seznam odkazů na výrazy reprezentující tyto kvantifikátory.
- `FunctionParametersCollector` - Sesbírá a zpracuje jednotlivé parametry zřetězené binární funkce[3]. Nalezené parametry, přesněji odkazy na výrazy, jenž tyto parametry reprezentují, jsou uloženy podle typu do vytvořených seznamů a převedeny do formy, v níž reprezentují parametry n-nární funkce sčítání. Pokud nalezený parametr náležel funkci odčítání, je modifikován. V případě konstanty je změněno znaménko u její hodnoty, v případě funkce výskytu je uložen příznak indikující tuto skutečnost. Navíc jsou všechny funkce výskytu vyjmuty z formule. Aplikace tohoto transformačního objektu zanechá formuli v nekonzistentním stavu. Návrh předpokládá následné zpracování konstant a okamžité uvolnění podstromu, který reprezentoval zřetězenou binární funkci.
- `ClausesCreator` - Vytvoří seznam klauzulí QF_UFIDL formule z predikátů obsažených v omezené RTL formuli.

Kromě výše popsaných transformačních objektů prvního typu obsahuje modul i několik pomocných transformačních objektů druhého typu. Tyto objekty realizují časté operace nad formulemi používané při velké řadě úprav. Patří zde:

- `FormulaNegator` - Provede negaci celé formule.
- `QuantifiersRemover` - Odstraní veškeré kvantifikátory z formule a uloží je do vytvořeného seznamu.

Modul realizuje převod omezených RTL formulí na QF_UFIDL formuli postupnou aplikací devíti transformačních objektů v následujícím pořadí:

- `ConstantSumator` - Sečte veškeré konstanty v řetězené binární funkci na jedné straně predikátu. Využívá `FunctionParametersCollector` pro kolekci jednotlivých parametrů. Nalezené konstanty jsou zpracovány a nahrazeny konstantou reprezentující jejich celkovou sumu, staré konstanty jsou uvolněny.
- `RedundantQuantifierEliminator` - Eliminuje všechny zbytečné kvantifikátory, tedy kvantifikátory, jenž ve svém rozsahu platnosti neobsahují žádnou vázanou proměnnou a jejich vypuštění tedy nijak nezmění význam formule. Hledání těchto kvantifikátorů je

[3] Zřetězenou binární funkcí je myšlena sekvence binárních funkcí, kde výsledek předchozí binární funkce je jedním z parametrů následující binární funkce.

založeno na jednoduchém principu. Pokud se odstraní kvantifikátor, který ve svém rozsahu platnosti obsahoval vázanou proměnnou, stane se tato proměnná volnou. Počet těchto proměnných lze zjistit pomocí transformačního objektu `FreeVariableFinder`. Pokud je počet nulový, kvantifikátor neobsahoval ve svém rozsahu platnosti žádné vázané proměnné a může být odstraněn.

- `QuantifiedVariableRenamer` - Přejmenovává vázané proměnné. Tato úprava je důležitou součástí předzpracování formule pro její převedení do prenexní normální formy. Převod vyžaduje, aby se žádná vázaná proměnná nevyskytovala v rozsahu platnosti jiného kvantifikátoru, jenž váže stejnou proměnnou. Navíc nesmí být žádná proměnná ve formuli zároveň volná i vázaná. Transformační objekt nejprve vytvoří seznam všech kvantifikátorů ve formuli pomocí `QuantifiersCollector`. Pak tento seznam postupně prochází a zpracovává jednotlivé vázané proměnné. V případě, že se jedná o první zpracování vázané proměnné daného názvu, ověří pomocí `FreeVariableFinder`, zda se ve formuli nevyskytují volné proměnné stejného názvu na nejvyšší úrovni. Pokud ano, musí být zpracovávaná vázaná proměnná přejmenována. Jakmile je nalezena vázaná proměnná s názvem, který již existuje a byl zpracován, musí být tato vázaná proměnná taktéž přejmenována. Přejmenování se provádí nahrazením původního názvu kvantifikované proměnné a všech vázaných proměnných kvantifikátoru novým unikátním názvem. Tento název, spolu s unikátním identifikátorem, je vytvořen metodou `generateId(...)` ve formě `_rq#`*`původní_název`*`->`*`nové_ID`*. Vázané proměnné jsou vyhledány pomocí `FreeVariableCollector`.

- `NegationEliminator` - Eliminuje všechny symboly negace ve formuli. Transformační objekt postupně prohledává formuli a při nalezení symbolu negace provede negaci následující formule aplikací transformačního objektu `FormulaNegator` na tuto formuli. Výraz reprezentující nalezený symbol negace je pak odstraněn z formule.

- `ImplicationEliminator` - Eliminuje všechny symboly implikace ve formuli. Transformační objekt postupně prohledává formuli a při nalezení symbolu implikace převede tuto implikaci na disjunkci podle vztahu $A \to B \leftrightarrow \neg A \vee B$. Negace formule A je provedena aplikací transformačního objektu `FormulaNegator`.

- `PrenexNormalFormCreator` - Převede formuli do prenexní normální formy. V této fázi je formule ve tvaru, kdy pro převod formule do prenexní normální formy postačuje přesunout veškeré kvantifikátory na začatek celé formule. Transformační objekt aplikuje pravidla $((Qx)(A)\ op\ B) \leftrightarrow (Qx)(A\ op\ B)$ a $(A\ op\ (Qx)(B)) \leftrightarrow (Qx)(A\ op\ B)$, kde Q je univerzální nebo existenční kvantifikátor a *op* je symbol konjunkce nebo disjunkce. Kvantifikátory jsou vyjmuty z formule pomocí transformačního objektu `QuantifiersRemover`, následně jsou pak přidány na počátek celé formule.

- `PredicateNormalizator` - Provede normalizaci predikátů, nerovností diferenční logiky, do tvaru $funkce\ výskytu \pm celočíselná\ konstanta \leq funkce\ výskytu$.

- `SkolemNormalFormCreator` - Provede skolemizaci, čili převod formule do Skolemovy normální formy. Tato transformace způsobí odstranění existenčních kvantifikátorů a nahrazení vázaných proměnných těchto kvantifikátorů odpovídajícími Skolemovými konstantami či funkcemi. Generování unikátních názvů a identifikátorů pro vytvořené Skolemovy konstanty a funkce je prováděno metodou `generateId(...)`. Názvy konstant jsou ve formě `_sc#`*`název_vázané_proměnné`*`->`*`ID_konstanty`*, názvy funkcí ve

formě _sc#*název_vázané_proměnné*->f(*seznam_ID*)->*ID_funkce*. Vázané proměnné jsou vyhledány pomocí `FreeVariableCollector`.

- `QFUFIDLFormulaCreator` - Převede omezené RTL formule na jedinou QF_UFIDL formuli. Pro vytvoření jednotlivých klauzulí této formule se využívá `ClausesCreator`.

Po aplikaci výše zmíněných transformačních objektů jsou tedy vstupní omezené RTL formule převedeny na jedinou QF_UFIDL formuli s přiloženou překladovou tabulkou, jenž nyní obsahuje informace o přejmenovaných vázaných proměnných a vytvořených Skolemových konstantách a funkcích. QF_UFIDL formule je vždy tvořena konjunkcemi disjunkcí nerovností ve tvaru *funkce výskytu* ± *celočíselná konstanta* ≤ *funkce výskytu*. Interní reprezentace využívá tohoto neměnného schématu a ukládá pouze jeho jakési parametry. QF_UFIDL formule je reprezentována seznamem klauzulí, objektů třídy `Clause`, které jsou tvořeny seznamem nerovností, struktur `TInequality`. Každá nerovnost je charakterizována sadou šesti atributů:

- `id` - Jednoznačný unikátní identifikátor nerovnosti a zároveň hrany grafu omezení.
- `leftFunctionId` - Identifikátor názvu akce na levé straně nerovnosti.
- `leftFunctionParameterId` - Identifikátor konstanty nebo proměnné určující pořadové číslo výskytu akce na levé straně nerovnosti.
- `constantValue` - Hodnota konstanty v nerovnosti.
- `rightFunctionId` - Identifikátor názvu akce na pravé straně nerovnosti.
- `rightFunctionParameterId` - Identifikátor konstanty nebo proměnné určující pořadové číslo výskytu akce na pravé straně nerovnosti.

6.3.11 Modul pro tvorbu grafu omezení

Modul pro tvorbu grafu omezení (*Constraint Graph Builder*) je třetím z verifikačních modulů. Realizuje tvorbu grafu omezení z QF_UFIDL formule a vyhledává kladné cykly, které se v tomto grafu nacházejí. Modul vyžaduje jako vstupní data QF_UFIDL formuli ve formě interní reprezentace a generuje k této formuli tzv. formule cyklů, jenž popisují nalezené pozitivní cykly.

Konstrukce grafu omezení probíhá na základě algoritmu popsaného v kapitole 4.5. Uzly tohoto grafu tvoří tzv. shluky (*clusters*) a orientované hrany zase jednotlivé nerovnosti. Graf je reprezentován formou tabulky sousednosti, která mapuje shluky na seznam hran z nich vycházejících. Shluky odpovídají obecným akcím v nerovnostech, tedy akcím stejného názvu. Každý shluk navíc obsahuje neomezeně konkrétních akcí, neboli akcí parametrizovaných proměnnou nebo konstantou, jenž vyjadřuje pořadové číslo výskytu dané akce. Hrany jsou ohodnoceny váhou na základě konstanty obsažené v nerovnosti, kterou reprezentují. Graf omezení je vytvořen z nerovností obsažených v QF_UFIDL formuli.

Hlavním úkolem modulu je nalezení pozitivních cyklů v grafu omezení. Bohužel neexistují žádné algoritmy, jenž by vyhledávaly přímo cíleně takovéto cyklu. Modul tedy pracuje ve dvou krocích. Nejprve identifikuje veškeré cykly nacházející se v grafu a následně ověří, zda nalezený cyklus splňuje potřebné podmínky.

Algoritmus pro vyhledání všech cyklů v grafu omezení je založen na Tarjanově algoritmu pro výpočet elementárních okruhů v orientovaném grafu[7], který je upraven, aby

byl aplikovatelný na graf omezení. Algoritmus předpokládá vzestupné číslování uzlů grafu. Shluky, které tvoří uzly, jsou identifikovány celočíselným identifikátorem přiřazeným akci, jenž shluk reprezentuje. Tyto identifikátory splňují požadavky kladené na číslování a lze je tedy využít jako náhradu. Pseudokód popisující činnost navrženého algoritmu je zobrazen na obrázku 6.4.

Princip algoritmu je založen na prohledávání DFS (*Depth First Search*). Algoritmus postupně prochází možné cesty z každého uzlu grafu a testuje zda netvoří tyto cesty cyklus. Jádro výpočtu tvoří metoda `backtrack(...)`, která je rekurzivně volána na každý další uzel prohledávané cesty na základě existence hrany mezi tímto uzlem a aktuálně zpracovávaným. Tato metoda realizuje řízené prohledávání grafu omezení algoritmem DFS. Cílem algoritmu je nalézt cesty, jenž vedou z výchozího uzlu a zase se do něj zpět vracejí, takovéto cesty tvoří hledané cykly. Hledání těchto cest je ovšem trochu problematické, jelikož každému cyklu grafu odpovídá obecně více cest, přesněji cyklu tvořenému k uzly odpovídá k ekvivalentních cest, které ho popisují. Například v grafu na obrázku 4.1 lze nalézt cyklus tvořený třemi uzly f, $g1$ a $g2$, tomuto grafu ale odpovídají hned tři cesty $fg1g2f$, $g1g2fg1$ a $g2fg1g2$, podle toho, ze kterého uzlu začíná prohledávání. Samozřejmě by bylo možné vyhledat veškeré tyto cesty a následně provést filtrování na základě ekvivalence, tento přístup ovšem zbytečně zpracovává obrovské množství cest navíc. Použitý algoritmus řeší tento problém efektivněji aplikací následující úvahy. Prohledání možných cest z každého uzlu určitého cyklu vždy vyústí v nalezení jedné konkrétní cesty, která tento cyklus popisuje. Tyto nalezené cesty budou ale ekvivalentní a stačilo by tedy nalézt jedinou z nich. Pokud bude vybrán vždy jediný uzel cyklu jako výchozí uzel prohledávání, pak algoritmus nalezne jedinou cestu pro každý cyklus grafu. Zbývá tedy určit postup, jak vybrat tento uzel. Navržený algoritmus ho určuje jednoduše podle očíslování. Vždy je prohledán uzel označený nejmenším číslem.

Celková činnost algoritmu by se dala shrnout a popsat asi následovně. Algoritmus prohledává uzly grafu od uzlů s nejmenším číslem do uzlů s nejvetším číslem. Na každý uzel je zavolána metoda `backtrack(...)`, která postupně prohledává jednotlivé možné cesty z výchozího uzlu. Aktuální cesta je uložena ve formě posloupnosti uzlů na zásobníku `pointStack`. Jedno volání metody `backtrack(...)` vždy zpracovává hrany jdoucí ze specifikovaného uzlu. Pokud metoda zjistí, že některá z hran končí ve výchozím uzlu, je nalezen cyklus, který je zpracován podle potřeb modulu. Jinak je cílový uzel, kde hrana končí, rekurzivně prohledán metodou `backtrack(...)`, ovšem pouze za předpokladu, že má cílový uzel vyšší číslo než aktuálně zpracovávaný. Toto omezení zajišťuje právě nalezení jediné cesty pro každý cyklus, jelikož algoritmus se nemůže vrátit do nižší uzlů. Tedy pouze ten uzel cyklu, který má nejnižší číslo ze všech může projít všemi uzly tohoto cyklu a vytvořit tak cestu, jenž ho reprezentuje. Zároveň si algoritmus značkuje uzly, kterými již prošel, pomocí příznaku `mark` a ukládá si informace o provedeném značkování aktuální rekurze na zásobník `markedStack`. Tento příznak umožňuje algoritmu ověřovat, zda cílový uzel hrany není již součástí aktuální prohledávané cesty. Pokud ano, nelze tímto směrem v cestě pokračovat, jelikož cyklus nesmí obsahovat jeden uzel dvakrát (samozřejmě s vyjímkou prvního a posledního uzlu cesty). Je zřejmé, že tato situace vlastně znamená nalezení cyklu, algoritmus se vrátil do uzlu, který již navštívil během konstrukce aktuální cesty. Ignorování tohoto cyklu ovšem jinak nevadí, jelikož tento cyklus bude nalezen opětovně později, při prohledávání cest z tohoto uzlu (což musí nastat jelikož algoritmus může prohledávat pouze uzly s vyšším číslem a tedy tento uzel ještě nemohl být použit jako výchozí uzel prohledávání).

Časová složitost popsaného algoritmu je $O((V+E)(C+1))$, kde V je počet uzlů grafu, E počet hran grafu a C počet cyklů grafu. I když na první pohled se zdá tato složitost jako přijatelná, je potřeba si uvědomit, že počet cyklů grafu C může narůstat exponenciálně

```
procedure findPositiveCycles;
  function backtrack(cluster : integer) : boolean;
    var f, g : boolean;
    begin
      f := false;
      place cluster on pointStack;
      mark(cluster) := true;
      place cluster on markedStack;
      foreach c in adjacencyTable(cluster) do
        if c == cluster then // Hrana se vrací do výchozího uzlu
          begin // Cyklus tvoří posloupnost shluků na zásobníku
            process cycle given by pointStack;
            f := true; // Nalezen cyklus
          end;
        else if c > cluster and not mark(c) then
          begin // Rozšiř prohledávanou cestu o další uzel
            g := backtrack(c);
            f := f or g; // Byl nalezen cyklus ?
          end;
      if (f == true)
        begin
          while top of markedStack <> cluster do
            begin
              t := top of markedStack;
              mark(t) := false;
              delete t from markedStack;
            end;
          delete cluster from markedStack;
          mark(cluster) := false;
        end;
      delete cluster from pointStack;
      backtrack := f;
    end;
  begin
    foreach cluster in clusterTable do mark(cluster) := false;
    foreach cluster in clusterTable do
      begin
        backtrack(cluster);
        while markedStack not empty do
          begin
            t := top of markedStack;
            mark(t) := false;
            delete t from markedStack;
          end;
      end;
  end;
```

Obrázek 6.4: Pseudokód algoritmu pro detekci cyklů v grafu omezení

s počtem uzlů V a hran E. Složitost tedy musí být brána jako exponenciální.

Po nalezení všech cyklů grafu ovšem práce modulu nekončí, následně je potřeba odfiltrovat cykly, které nesplňují podmínky kladných cyklů grafu omezení. Filtrování je integrováno do metody `processCycle()`, jenž je volána vždy po nalezení cyklu v grafu. Informace o nalezeném cyklu jsou uloženy v seznamu `m_pointList`, který obsahuje posloupnost hran tvořících tento cyklus. Ověření kladnosti cyklu je jednoduché, postačuje projít hrany cyklu a sečíst jejich váhy, pokud je výsledná suma větší než nula, je cyklus kladný.

Složitější je ověření celkové platnosti cyklu z hlediska grafu omezení. Podle definice cyklu grafu omezení uvedené v kapitole 4.6 musí existovat substituce σ, kterou lze unifikovat jednotlivé konkrétní uzly v_i a v_i' navzájem navazujících hran. Algoritmus pro vyhledání cyklů totiž pracuje pouze se shluky a ignoruje jejich vnitřní strukturu, tedy vlastně předpokládá automatickou existenci substituce σ. Modul tedy musí takovou substituci nalézt a dokázat tak platnost cyklu. Ověřování exitence substituce σ provádí metoda `isValidSubstitution(...)`. Tato metoda pracuje inkrementálně postupným rozšiřováním cesty cyklu o další uzly. To umožňuje zastavit zpracování cyklu, jakmile je zjištěno, že platnou substituci již není možné vytvořit. Přidání dalších uzlů na tomto stavu nemůže nic změnit.

Specifikace vytvářené substituce je uložena ve struktuře `SubstitutionTable`, která obsahuje informace o provedených substitucích pro každý shluk obsažený v daném cyklu. Při přidání dalšího uzlu nějaké shluku se ověřuje, zda je možné provést substituci, jenž není v rozporu s ostatními. Povolené substituce jsou následující:

- **Substituce proměnné za jinou proměnnou** - Platná substituce, pokud nejsou zároveň obě tyto proměnné již substituovány za odlišné konstanty.
- **Substituce proměnné za konstantu** - Platná substituce, pokud proměnná není zatím substituována za žádnou konstantu a zároveň žádná jiná proměnná, která ma definovanou substituci s touto proměnnou, není substituována za odlišnou proměnnou.

Nalezené a ověřené kladné cykly jsou uloženy jako posloupnost hran reprezentujících jednotlivé nerovnosti s přiloženými informacemi o provedené substituci.

6.3.12 Modul pro tvorbu prohledávacího stromu

Modul pro tvorbu prohledávacího stromu (*Search Tree Builder*) je posledním z verifikačních modulů. Realizuje tvorbu prohledávacího stromu z nalezených kladných cyklů grafu omezení a dokazuje neplatnost těchto cyklů na základě nesplnitelnosti některých nerovností, které hrany těchto cyklů reprezentují. Modul vyžaduje jako vstupní data QF_UFIDL formuli ve formě interní reprezentace s přiloženým seznamem nalezených kladných cyklů a produkuje zprávu, jenž je výsledkem celého procesu verifikace, tedy zda je ověřovaná vlastnost platná nebo její platnost nelze dokázat.

Celý proces ověřování je založen na principech popsaných v kapitole 4.7. Modul postupně vytváří prohledávací strom a dokazuje neplatnost všech cest tímto stromem, neboli nesplnitelnost formule tvořené konjunkcí negací nerovností, kterou tato cesta reprezentuje. Jelikož je formule tvořena konjunkcí několika atomických formulí, stačí pro dokázání její nesplnitelnosti vyvrátit platnost kterékoliv obsažené atomické formule.

Vyvrácení platnosti formule využívá zákon vyloučení sporu, který říká, že nemůže platit výrok a zároveň jeho negace. Ověřované formule obsahují jako atomické formule negace nerovností, modul má zároveň k dispozici seznam klauzulí, disjunkcí nerovností, které jsou v daném systému platné. Algoritmus se snaží nalézt k některé negaci nerovnosti adekvátní

nerovnost platnou v daném systému. Pokud ji najde, musí být negace této nerovnosti a zároveň i celé formule, cesty v prohledávacím stromu, nesplnitelná. Při hledání potřebné nerovnosti mohou nastat dva případy:

- **Nalezená nerovnost náleží atomické klauzuli** - V tomto případě musí být nerovnost platná, jelikož klauzule je vždy platná.
- **Nalezená nerovnost je jedna z nerovností obsažených v klauzuli** - V tomto případě nemusí být nerovnost nutně platná. Klauzule je tvořena disjunkcí jednotlivých nerovností a pro její platnost tedy postačuje, aby byla platná alespoň jedna z obsažených nerovností. Aby mohla být tato klauzule, přesněji její nerovnost, použita pro vyvrácení platnosti negace této nerovnosti, je potřeba navíc dokázat, že ostatní nerovnosti nemohou být platné.

Aby se urychlilo hledání potřebných nerovností, vytváří si modul tabulku mapování identifikátorů nerovností na klauzule, které tyto nerovnosti obsahují. Cesta v prohledávacím stromu je tvořena posloupností identifikátorů nerovností, které jsou interpretovány jako negace těchto nerovností. Algoritmus vezme jednu negaci nerovnosti z této posloupnosti a vyhledá skrz tabulku klauzuli, jenž obsahuje původní nerovnost. Pokud je klauzule atomická, pak je platnost cesty vyvrácena. Pokud obsahuje klauzule více nerovností, poznačí si algoritmus, že pro vyvrácení cesty pomocí této klauzule musí zároveň vyvrátit platnost ostaních nerovností obsažených v klauzuli a pokračuje analýzou další negace nerovnosti ověřované cesty. Pokud v dalších krocích ověřování narazí algoritmus na negaci nerovnosti, která je v rozporu s nerovností v již dříve použité neatomické klauzuli, pak ověří, zda nejsou již všechny nerovnosti této klauzule v rozporu s nějakými negacemi nerovností v aktuálně ověřované cestě. Pokud ano, je formule reprezentována touto cestou nesplnitelná, jelikož všechny negace nerovností v cestě musí být splněny, ale zároveň alespoň jedna nerovnost klauzule musí být také splněna, což si navzájem odporuje.

V kapitole 4.8 bylo zmíněno několik možností optimalizace prohledávacího stromu. Nejpřínosnější optimalizací je zastavení prohledávání, pokud je nějaký úsek cesty označen jako nesplnitelný. V takovém případě již nemá smysl dále generovat uzly prohledávacího stromu, jelikož rozšíření cesty o jakékoliv nové negace nerovností nijak nezmění fakt, že tato cesta bude nesplnitelná. Tato optimalizace je také implementována v modulu.

Druhou možnou optimalizací je přeskládání formulí cyklů, jenž vlastně způsobí přesunutí určitých negací nerovností na začátek cesty prohledávacího stromu. Pokud jsou tyto negace jednoduše vyvrátitelné, zastaví se dříve generování dalších uzlů a zmenší se tak velikost prohledávacího stromu. Tato optimalizace je závislá na použití heuristik pro určení vhodného pořadí formulí cyklů a přínosnost je dosti sporná. Proto implementovaný modul tento druh optimalizace zatím neprovádí.

Poslední optimalizací je využití dříve ověřených cest. Pokud je aktuálně ověřovaná cesta ekvivalentní s jinou, již ověřenou, cestou, nemusí se ověřování vůbec provádět. I když se tato optimalizace zdá jako přínosná, je potřeba dobře zvážit celkový efekt této optimalizace. V případě implementovaného modulu znamená nalezení ekvivalentní, již ověřené, cesty vykonání několika kroků. Modul si musí uchovávat informace o již ověřených cestách a umět tyto cesty porovnat. Cesty reprezentují formule tvořené konjunkcí negací nerovností. Dvě takovéto formule jsou ekvivalentní, pokud obsahují identické nerovnosti, neboli jedna formule je tvořena permutací nerovností formule druhé. Pro provedení porovnání je tedy nejprve potřeba seřadit tyto permutace nerovností do pořadí, které bude pro všechny ekvivalentní permutace totožné. Toho lze dosáhnout například vzestupným seřazením podle

celočíselného identifikátoru nerovností. I s použitím nejlepšího řadícího algoritmu bude tato složitost $O(n\ log\ n)$. I pokud následné porovnání bude provedeno vyhledáním záznamu v hashovací tabulce v konstantním čase $O(1)$, pak výsledný čas zjištění, zda již daná cesta nebyla ověřena dříve, bude mít časovou náročnost $O(n\ log\ n)$. Tato složitost navíc nezahrnuje případ neexistence ekvivalentní, dříve ověřené, cesty, kdy je stejně potřeba provést následné ověření aktuální cesty. Oproti tomu je zde možnost ověření cesty, kterou již možná algoritmus dříve ověřil. V nejhorším případě musí být ověřeny všechny negace nerovností v aktuální cestě, což znamená složitost $O(n)$. Ověření negace nerovnosti je provedeno vyhledáním potřebné klauzule s původní nerovností pomocí hashovací tabulky v konstantním čase $O(1)$. Výsledná časová složitost ověřování cesty je tedy $O(n)$. Tedy zavedení optimalizace neověřování již ověřených cest by nakonec znamenalo celkové zpomalení algoritmu místo předpokládaného zrychlení. Proto tuto optimalizaci modul vůbec neimplementuje.

6.4 Grafické uživatelské rozhraní

Jak již bylo zmíněno v kapitole 6.1, grafické uživatelské rozhraní je napsáno v jazyce Java. Návrh je, stejně jako v případě verifikátoru, zaměřen na modularitu a dobrou rozšiřitelnost programu. Účelem aplikace je zprostředkovat služby a možnosti verifikátoru jednotlivým svým modulům. Blokové schéma na obrázku 6.5 zachycuje detailní strukturu grafického uživatelského rozhraní.

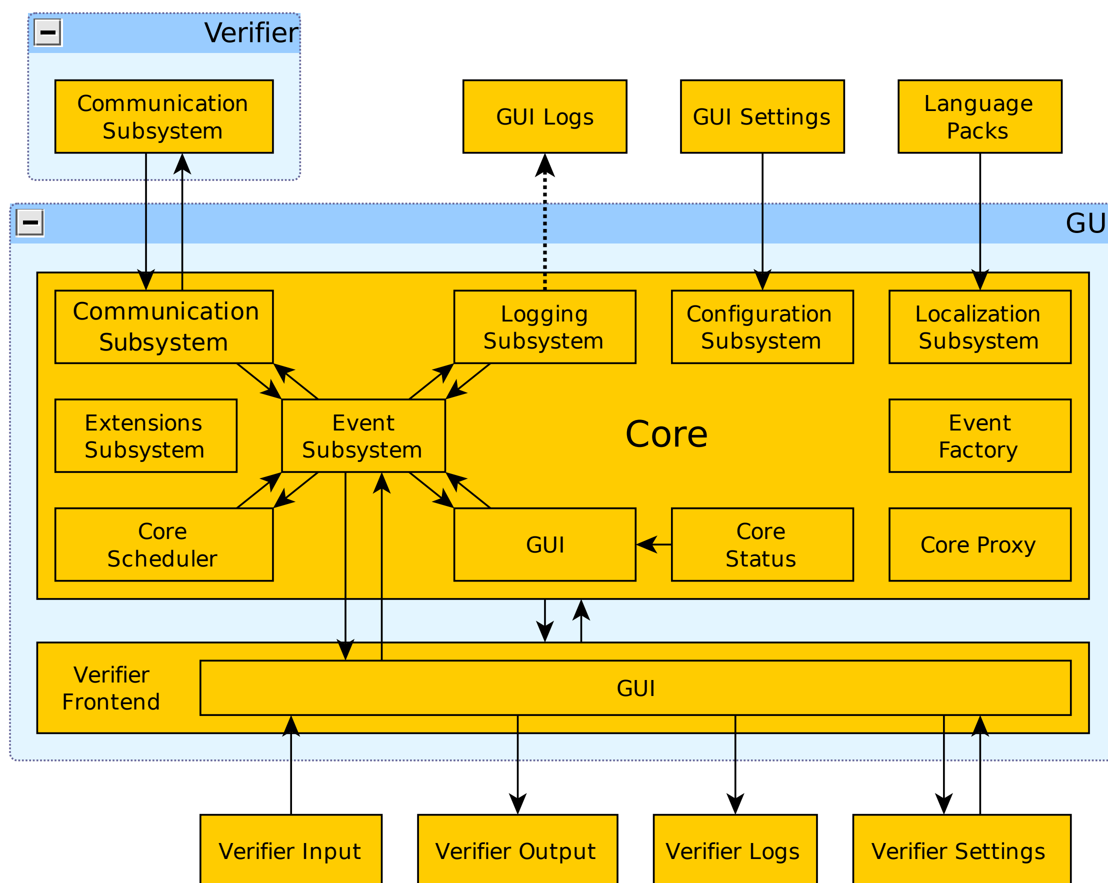
Grafické uživatelské rozhraní se skládá z jádra a šesti subsystémů. Většina přítomných subsystémů je modifikovanou implementací stejnojmenných subsystémů verifikátoru. Odlišnosti plynou z rozdílných vlastností jazyka Java a C++. Hlavním rozdílem je, že grafické aplikace v jazyce Java jsou ze své podstaty vícevláknové a je zde tedy potřeba zajistit správnou synchronizaci těchto vláken. Dále jsou zde také odlišné přístupy z hlediska správy paměti, využití externích knihoven nebo třeba nemožnost použití vícenásobné dědičnosti. Všechny tyto aspekty mohou ovlivnit návrh a následnou implementaci jednotlivých subsystémů nebo jiných částí.

6.4.1 Komponenty

Jelikož je celá aplikace vícevláknová a často je výhodné, nebo dokonce nutné, provádět určitý výpočet či nějakou činnost v separátním vlákně, poskytuje aplikace několik tříd pro tyto účely. Tyto třídy rozšiřují standardní komponenty grafického uživatelského rozhraní a obohacují je o možnosti běhu v separátním vlákně. Cílem těchto tříd je usnadnit vývoj komponent, které provádějí práci v separátním vlákně a neblokují tak činnost grafického uživatelského rozhraní. Za zmínku stojí hlavně dvě třídy, hojně využívané i v aplikaci samotné:

- Třída `ZDaemon` - Poskytuje zděděným třídám možnosti vytvoření nového vlákna a spuštění kódu obsaženého v metodě `run()` v tomto vlákně. Spuštění kódu metody `run()` ve vytvořeném vlákně se provádí pomocí metody `start()` a běh vlákna končí jakmile dojde k návratu z metody `run()` nebo je aplikace ukončena.

- Třída `ZEventProcessingDaemon` - Rozšíření třídy `ZDaemon` poskytující zděděným třídám možnosti zpracování přijatých událostí v separátním vlákně. Třídám stačí pouze implementovat metodu `processEnqueuedZEvent(...)`, jenž je volána při přijetí události, a spustit zpracovávání událostí pomocí metody `start()`. Třída uchovává přijaté události v interní frontě a předkládá je metodě `processEnqueuedZEvent(...)`.



Obrázek 6.5: Detailní blokové schéma grafického uživatelského rozhraní

Pokud je fronta prázdná, vlákno se automaticky uspí a čeká na nové události. Jakmile nějaká dorazí, vlákno je opětovně spuštěno a provádí zpracování. Přístup k frontě událostí je chráněná zámkem, je tedy zajištěna její konzistence a synchronizace při přístupu z více vláken současně. Třída navíc umožňuje úplné zastavení zpracovávání a tedy i ukončení vlákna pomocí metody `stop()`, nejdříve je ale dokončeno zpracování událostí přítomných ve frontě událostí.

6.4.2 Jádro

Jádro grafického uživatelského rozhraní je reprezentováno třídou `Core` a plní stejnou úlohu jako jádro u verifikátoru (viz. 6.3.1). Na rozdíl od verifikátoru zde ovšem není implementováno jako *singleton*[3] objekt. Důvodem je odlišnost v řízení běhu aplikace. Aplikace běží tak dlouho, dokud se neuzavře její hlavní okno, neboli dokud se nezruší vytvořená instance třídy, která toto okno reprezentuje. Instance jádra je zde součástí vytvořené instance třídy hlavního okna a je tedy přítomná po celou dobu chodu aplikace.

Jelikož grafické rozhraní musí adekvátně reagovat na aktuální činnost jádra, je potřeba umožnit jeho monitorování. Monitorování je založeno na návrhovém vzoru *observer*[3]. Jádro udržuje svůj vnitřní stav, reprezentovaný instancí třídy `CoreStatus`, který popisuje

činnost aktuálně vykonávanou jádrem. Pokud dojde ke změně stavu jádra, jsou o této změně informovány všechny objekty monitorující tento stav. Spuštění monitorování se provádí pomocí metody jádra `addStatusObserver(...)`.

6.4.3 Proxy jádra

Proxy jádra slouží ke stejnému účelu jako v případě verifikátoru (viz. 6.3.2). Je reprezentováno rozhraním `ZCoreProxy` a rozšířeno o několik nových služeb, které se týkají hlavně nových součástí jádra.

6.4.4 Plánovač jádra

Plánovač jádra je první částí jádra, jenž doznala vetších změn. Je tvořen instancí třídy `CoreScheduler` a zajišťuje plánování veškerých činností jádra. Třída přijímá a zpracovává příkazy, události reprezentované třídou `ZCommandEvent`, které zasílá grafické uživatelské rozhraní na základě interakce s uživatelem. Plánovač tyto příkazy mapuje na konkrétní činnosti jádra. Jelikož činnost jádra nesmí blokovat grafické uživatelské rozhraní, jsou všechny příkazy zpracovány v separátním vlákně. Využívá se zde třídy `ZEventProcessingDaemon`, jenž tvoří základ třídy plánovače.

6.4.5 Událostní subsystém

Událostní subsystém je založen na stejném principu jako u verifikátoru (viz. 6.3.4). Událost je zde reprezentována třídou `ZEvent`. Na rozdíl od verifikátoru definuje grafické uživatelské rozhraní třetí typ události, tzv. vzdálenou (*remote*) událost. Tento typ události, identifikovaný konstantou `REMOTE`, se využívá pro události zaslané verifikátorem. Jelikož události vytvářené v jazyce Java jsou, jako všechny jiné objekty, automaticky rušeny, jakmile již nejsou nikde potřeba, odpadá zde nutnost vytváření kopií událostí při zasílání několika odběratelům zároveň. Stejný objekt události je tedy zaslán všem odběratelům.

Událostní subsystém je reprezentován třídou `EventMediator` a všechny objekty, které chcou naslouchat na příchozí událostí, musí implementovat rozhraní `ZEventListener`.

6.4.6 Subsystém rozšíření

Subsystém rozšíření, zastoupený třídou `ExtensionsManager`, je konceptuálně stejný jako v případě verifikátoru (viz. 6.3.5). Hlavní rozdíl spočívá v odlišné reprezentaci a přístupu k externím zásuvným modulům a také v rozdílných požadavcích kladených na tyto moduly.

Externí zásuvné moduly tvoří JAR archívy (soubory **.jar*), které jsou načteny z adresáře specifikovaného aktuální konfigurací. Ve výchozím nastavení se prohledávají, stejně jako u verifikátoru, adresáře *modules* a *plugins*. Vstupní bod zásuvného modulu tvoří třída implementující rozhraní `ZPluginEntry`. Toto rozhraní definuje metodu `getPlugin()`, jenž vrací konkrétní objekt implementucící rozhraní `ZPlugin`. Rozhraní `ZPlugin` pak poskytuje metody, které vracejí informace o zásuvném modulu, jmenovitě:

- **SID** - Řetězec identifikující konkrétní zásuvný modul a objekt, jenž tento modul vytváří. Tento řetězec lze získat pomocí metody `sid()`.
- **Implementované rozhraní** - Řetězec identifikující rozhraní, resp. objekt, které implementuje, resp. rozšiřuje, objekt vytvářený zásuvným modulem. Tento řetězec lze získat pomocí metody `implement()`.

Objekty se, stejně jako u verifikátoru, vytvářejí pomocí metody `create()`. Pro nalezení vstupního bodu zásuvného modulu se využívá jeho název. Subsystém rozšíření předpokládá, že vstupní třída je nazvána `Entry` a balíček, jenž tuto třídu obsahuje, je zároveň názvem zásuvného modulu. Pro oddělení jednotlivých názvů úrovní balíčku lze použít buď standardního znaku tečka nebo pomlčku. Například v případě zásuvného modulu reprezentovaného souborem *zetav-gui-plugin.jar* bude subsystém rozšíření hledat vstupní třídu s názvem `zetav.gui.plugin.Entry`.

Grafické uživatelské rozhraní rozlišuje dva typy zásuvných modulů:

- **Zásuvný modul (*plugin*)** - Standardní zásuvný modul jehož vytvářené objekty implementují kterékoliv rozhraní nebo rozšiřují jakýkoliv, abstraktní či konkrétní, objekt.
- **Zásuvný panel (*module*)** - Speciální zásuvný modul s grafickým rozhraním. Objekty vytvářené tímto modulem musí rozšiřovat abstraktní třídu `ZGuiModule`. Tato třída definuje metodu `getModuleGui()`, jenž vrací instanci grafické komponenty, která je přidána ve formě nové záložky do grafického uživatelského rozhraní.

Jelikož správa všech vytvářených objektů je v režii jazyka Java, je zbytečné, aby tuto funkci zastával i subsystém rozšíření, jak tomu je v případě verifikátoru. Díky zrušení této role ovšem není možné vytvářet sdílené objekty.

6.4.7 Protokolovací subsystém

Protokolovací subsystém je i zde reprezentován třídou `LogManager` a plní stejnou funkci jako v případě verifikátoru (viz. 6.3.6). Zpracovává protokolovací události, objekty třídy `ZLogEvent`, a zapisuje v nich obsažené zprávy do protokolu událostí. Pro zápis využívá objekty implementující rozhraní `ZLogger`. Grafické uživatelské rozhraní umožňuje zaznamenávat protokolovací zprávy pouze to určeného souboru.

Prakticky jediným konceptuálním rozdílem od implementace ve verifikátoru je odlišná forma filtrování zápisů v závislosti na nastavení úrovně protokolování. Zatímco v případě verifikátoru došlo k vytvoření a následnému odeslání protokolovací události až po otestování, zda nastavená úroveň vyžaduje zaprotokolování tvořené zprávy, zde je protokolovací událost vytvořena a zaslána vždy a protokolovací sybsystém rozhodne, zda tuto zprávu zaprotokolovat či nikoliv.

6.4.8 Konfigurační subsystém

Konfigurační subsystém prošel asi největšími změnami. Hlavním důvodem jsou značně odlišné požadavky, které na tento subsystém klade grafické uživatelské rozhraní. Jediná část, jenž zůstala nezměněna, je způsob identifikace položek konfigurace (viz. 6.3.7).

Na rozdíl od verifikátoru lze využívat pouze jediný zdroj konfigurace. K této konfiguraci se přistupuje skrz rozhraní `ZConfig`. Toto rozhraní definuje metody nejen pro načítání a získávání hodnot konfigurace, ale také metody pro ukládání a nastavování hodnot této konfigurace. Hodnoty musí být navíc vždy reprezentovány řetězcem, v případě získávání nebo nastavování hodnot jiných datových typů musí sám uživatel provádět potřebné konverze.

Grafické uživatelské rozhraní obsahuje jednu konkrétní implementaci tohoto rozhraní ve formě interního zásuvného modulu. Tato implementace využívá pro načítání a ukládání konfigurace XML soubory.

6.4.9 Komunikační subsystém

Stejně jako v případě verifikátoru i zde je komunikační subsystém implementován formou externího zásuvného modulu. Jádro obsahuje pouze odkaz na třídu implementující rozhraní `ZCommunicator`, jenž definuje metody pro celkovou inicializaci subsystému a pro spuštění a ukončení serveru a klienta. I když grafické uživatelské rozhraní vyžaduje pro svou činnost přítomnost konkrétní implementace tohoto rozhraní, je zbytečné zahrnout nějakou takovouto implementaci jako interní modul. Důvodem je neznalost implementace, jenž bude použita na straně verifikátoru, který sám neobsahuje žádnou konkrétní implementaci vlastního komunikačního subsystému. Je tedy na uživateli, aby poskytnul adekvátní zásuvné moduly na obou stranách.

Po nastartování klienta a serveru probíhá veškerá komunikace skrz událostní subsystém. Klient přijímá požadavky, události reprezentované třídou `ZRequestEvent`, a převádí je na vzdálená volání služeb serveru verifikátoru, výsledky volání pak zasílá zpět ve formě odpovědí, událostí reprezentovaných třídou `ZResponseEvent`. Server slouží k přijímání událostí verifikátoru, které si klient vyžádal. Tyto události jsou následně zaslány cílovým objektům ve formě vzdálených (*remote*) událostí.

6.4.10 Lokalizační subsystém

Lokalizační subsystém, reprezentovaný třídou `LocalizationManager`, zajišťuje načítání a správu jazykových balíčků a poskytuje metody pro získávání adekvátních řetězců v konfigurací definovaném jazyce. Cílem tohoto subsystému je poskytnout jednoduché možnosti lokalizace grafického uživatelského rozhraní, případně i zásuvných modulů, pro ještě snadnější obsluhu aplikace.

Lokalizační balíčky tvoří, stejně jako zásuvné moduly, JAR archívy. Tyto archívy musí být umístěny v adresáři *languages*, odkud jsou automaticky načteny během startu aplikace. Jeden archív může obsahovat lokalizační soubory pro více jazyků zároveň. Koncept lokalizace je založen na principech využívaných samotným jazykem Java. Využívá se tzv. *resource bundles*, objektů obsahujících mapování klíčů na jimi zastoupené konkrétní hodnoty. Toto mapování je uloženo v lokalizačních souborech ve formátu tzv. *property* souborů[4], textových souborů obsahujích položky ve formě dvojice *klíč=hodnota*. Kódování znaků těchto souborů musí být ISO-8859-1, znaky nepřítomné v tomto kódování je potřeba zapsat ve tvaru `\uHHHH`, kde `HHHH` je hexadecimální index Unicode znaku. Lokalizační soubor je identifikován na základě manifestu[5] JAR archívu. Tento manifest obsahuje pro každý lokalizační soubor atributy, tzv. *per-entry attributes*, s informacemi o příslušné lokalizaci. Každý lokalizační soubor musí definovat následující dva atributy:

- `Localization-Name` - Název lokalizace, ke které lokalizační soubor přísluší. Pojmenování lokalizace umožňuje rozlišovat lokalizační soubory různých částí, aby se předešlo konfliktům mezi lokalizacemi zásuvných modulů a samotné aplikace. Grafické uživatelské rozhraní používá lokalizaci s názvem `zetav.gui.lang.ZetavGUI`.

- `Localization-Locale` - Identifikátor jazyka lokalizačního souboru ve standardním formátu *`kód-jazyka_kód-země`*.

V případě lokalizace grafického uživatelské rozhraní do českého jazyka, kdy plná cesta k lokalizačnímu souboru v JAR archívu je `zetav/gui/lang/ZetavGUI_cs_CZ.lang`, by manifest

[4] Pro bližší informace viz. http://en.wikipedia.org/wiki/.properties

[5] Viz. http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#JAR%20Manifest

obsahoval následující definici:

```
Name: zetav/gui/lang/ZetavGUI_cs_CZ.lang // Per-entry atribut
Localization-Name: zetav.gui.lang.ZetavGUI
Localization-Locale: cs_CZ
```

6.5 Komunikace

Komunikace mezi grafickým uživatelským rozhraním a verifikátorem je realizována pomocí páru externích zásuvných modulů, které implementují komunikační subsystémy těchto nástrojů. Kromě implementace požadovaných rozhraní musí tyto moduly navíc adekvátně reagovat na požadavky zasílané ve formě událostí skrz událostní subsystém nebo naopak generovat očekávavé události jako reakci na přijetí určitého požadavku vzdálené strany.

6.5.1 Implementační požadavky

Implementační požadavky na oba komunikační subsystémy se výrazně liší. Zatímco komunikační subsystém grafického uživatelského rozhraní plní především úlohu klienta, komunikační subsystém verifikátoru zastává hlavně funkci serveru.

V následujícím textu bude komunikační subsystém grafického uživatelského rozhraní označován jako klient a komunikační subsystém verifikátoru jako server. Požadavkem typu `typ` je myšlen objekt `ZRequestEvent`, jehož metoda `getRequest()` vrací řetězec *typ*. Odpovědí typu `typ` je myšlen objekt `ZResponseEvent`, jehož metoda `getResponse()` vrací řetězec *typ*. Daty jsou myšleny objekty získané voláním metody `getData()` daného objektu události. Komponenty označují objekty, které mohou komunikovat skrz událostní subsystém.

Klient a server musí zajistit přesnou implementaci následujících služeb:

- **Přeposílání vyžádaných událostí verifikátoru** - Tato služba umožňuje zaregistrovat klienta jako odběratele specifického typu událostí. Komponenty žádají o tuto službu zasláním požadavku typu `subscribeForEvents`, který jako data obsahuje identifikaci typu události. Pokud jsou data datového typu `Integer` resp. `String`, vyjadřuje jejich hodnota celočíselný identifikátor (*ID*) resp. textový identifikátor (*String ID*) typu události. Klient musí reagovat na tyto požadavky a zajistit registraci na straně serveru. Server má k dispozici proxy jádra a přístup k událostnímu subsystému, může tedy jednoduše zaregistrovat sám sebe jako odběratele událostí vyžadovaných klientem a přeposílat mu tyto události.

- **Konfigurace verifikátoru** - Tato služba umožňuje provést vzdálenou konfiguraci verifikátoru. Komponenty žádají o tuto službu zasláním požadavku typu `configure`, který jako data obsahuje seznam, datový typ `List`, tří řetězců datového typu `String`. První řetězec obsahuje *SID* zásuvného modulu, který bude použit pro zpracování konfigurace. Další dva určují zdroj dat. Pokud je hodnota druhého řetězce *data*, resp. *file*, pak hodnota třetího řetězce obsahuje přímo konfigurační data, resp. název konfiguračního souboru na straně serveru. Klientovi postačuje reagovat na tyto požadavky přeposláním informací serveru. Server musí zajistit zpracování konfiguračních dat, resp. načtení konfiguračního souboru, a následnou konfiguraci verifikátoru, pro kterou může opět využít služeb jádra.

- **Přenos textových souborů** - Tato služba umožňuje uložit předložený text do souboru na straně serveru. Komponenty žádají o tuto službu zasláním požadavku typu `transferTextFile`, který jako data obsahuje seznam, datový typ `List`, tří řetězců datového typu `String`. První řetězec obsahuje název cílového souboru na straně serveru, který bude vytvořen nebo přepsán. Další dva určují zdroj dat. Pokud je hodnota druhého řetězce *data*, resp. *file*, pak hodnota třetího řetězce obsahuje přímo text přenášeného souboru, resp. název přenášeného souboru na straně klienta. Klient musí zajistit zaslání přijatých dat, resp. dat načtených ze zdrojového souboru, na server. Server je povinen obstarat uložení těchto dat do cílového souboru.

- **Spuštění verifikace** - Tato služba umožňuje spuštění verifikačního procesu na straně serveru. Komponenty žádají o tuto službu zasláním požadavku typu `verify`. Klient musí, kromě spuštění verifikačního procesu, také zajistit a zpracovat výsledek provedené verifikace. Tento výsledek pak musí poslat ve formě odpovědi typu `verify` obsahující jako data řetězec, datový typ `String`, popisující výsledek verifikace. Řetězec musí nabývat jedné ze tří hodnot:

  - *valid* - Verifikace prokázala platnost bezpečnostního tvrzení.
  - *not-valid* - Verifikace nemůže prokázat platnost bezpečnostního tvrzení.
  - *error* - Došlo k chybě při procesu verifikace.

  Požadavky kladené na server jsou stejné jako v případě klienta. Server spouští verifikační proces zasláním příkazu, události reprezentované třídou `CommandEvent`, typu `verify` (název příkazu, parametr `command`, je řetězec *verify*). Po dokončení verifikace je výsledek vrácen ve formě výsledku, události reprezentované třídou `ResultEvent`. Přiřazení výsledku k zaslanému příkazu se provádí na základě unikátních identifikátorů událostí. Po vytvoření příkazu lze metodou `getEventId()` získat jeho unikátní identifikátor, tento identifikátor je pak součástí zaslané odpovědi a vrací jej metoda `getResultId()`. Výsledek verifikace pak popisuje řetězec, jenž vrací metoda `getResult()`. Hodnota tohoto řetězce může nabývat stejných tří hodnot jako v případě odpovědi u klienta a má ekvivalentní význam.

6.5.2 Přenos událostí

Jednou ze služeb, jenž musí zajistit komunikační subsystémy, je přeposílání vyžádaných událostí verifikátoru. Je ovšem otázkou jak realizovat samotný přenos událostí. Delegovat tento problém na komunikační subsystém není vůbec vhodným řešením. Komunikační subsystém neví, a ani nemůže vědět, jaké události bude muset přeposílat. Způsob přenosu musí být tedy nezávislý na typu události.

Přenos událostí je založen na principech serializace a opětovného vytvoření události. Bázová třída všech událostí, třída `Event`, implementuje rozhraní `ISerializable`. Toto rozhraní definuje metodu `serialize()`, jenž vrací textový řetězec reprezentující danou událost. Všechny třídy událostí, které mohou být přenášeny, pak musí reimplementovat tuto metodu. Komunikační subsystém pak může, nezávisle na konkrétním typu události, provést převod události do textové podoby a v této podobě tuto událost zaslat klientovi. Ten ji musí převést zpět do podoby objektu reprezentujícího tuto událost. Klient tento převod ovšem nemusí realizovat, tato činnost je zajišťována samotným jádrem grafického uživatelského rozhraní. Jádro obsahuje instanci třídy `EventFactory`, jenž zajišťuje tvorbu událostí z jejich textové reprezentace pomocí registrovaných zpracovatelů (*handlers*) pro

jednotlivé typy událostí. Zpracovatel je konkrétní implementací rozhraní `ZEventHandler`, které definuje metody pro převod události z textové formy do formy objektu. Metoda `canCreateZEvent(...)` určuje, zda daný zpracovatel umí zpracovat událost specifikovaného typu, metoda `createZEvent(...)` pak vytváří událost tohoto typu z předložených dat. Objekty si mohou zaregistrovat vlastní zpracovatele pro přeposílané typy událostí pomocí metody `registerEventHandler(...)` a vyžádat vytvoření události z její textové reprezentace metodou `createEvent(...)`. Obě tyto metody jsou poskytovány proxy objektem jádra.

Takto navržený systém umožňuje jednoduše přenášet jakýkoliv typ události. Stačí pouze reimplementovat metodu `serialize()` u třídy události, která se bude přenášet, a implementovat a zaregistrovat zpracovatele tohoto typu události.

6.5.3 Konkrétní implementace

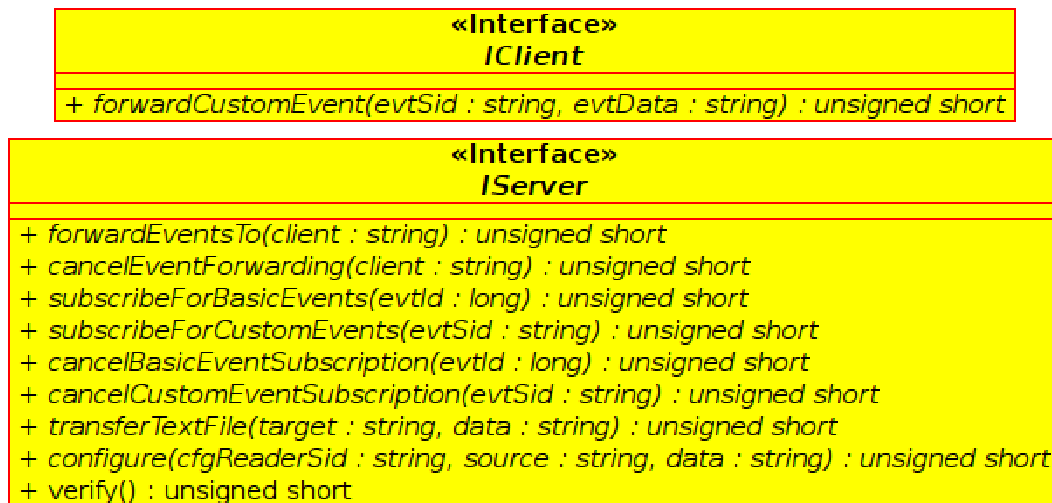
Pro zajištění spojení mezi grafickým uživatelským rozhraním a verifikátorem obsahují tyto nástroje pár zásuvných modulů realizujících komunikaci pomocí systému CORBA[6]. V případě zásuvného modulu pro grafické uživatelské rozhraní lze jednoduše využít možností jazyka Java, který obsahuje implementaci tohoto systému jako svou součást. Jazyk C++ takové možnosti neposkytuje, zásuvný modul pro verifikátor tedy využívá služeb knihovny omniORB verze 4.1[4], jenž poskytuje implementaci systému CORBA odpovídající specifikaci CORBA verze 2.6[6].

Systém CORBA je založen na tzv. distribuovaných objektech, jenž poskytují služby připojeným klientům. Na rozdíl od obvyklých forem komunikace, založených na protokolech a zasílání zpráv, je komunikace v systémech CORBA realizována formou volání metod distribuovaných objektů. Lokalizace distribuovaného objektu je prováděna na základě jeho IOR (*Interoperable Object Reference*). IOR[6] je struktura obsahující, kromě jiných informací, také informace o umístění distribuovaného objektu. I když lze IOR zapsat formou textového řetězce, je tento zápis pro uživatele absolutně nesrozumitelný. Naštěstí poskytuje systém CORBA i další možnosti lokalizace distribuovaného objektu založené na tzv. objektovém URL (*Object URL*). Klient využívá pro lokalizaci distribuovaného objektu serveru `corbaloc` URL[6], jenž umožňuje lokalizovat tento objekt na základě hostitelského jména, případně IP adresy, a čísla portu, kde distribuovaný objekt naslouchá na příchozí spojení klientů.

Jakmile se klient připojí k serveru, může požadovat služby, které nabízí distribuovaný objekt tohoto serveru. Klient k těmto službám přistupuje skrz rozhraní definované v jazyce IDL[6] (*Interface Definition Language*). Konkrétní implementace systému CORBA pak na základě definice tohoto rozhraní vytvoří rozhraní či abstraktní objekt v cílovém programovacím jazyce. Klient pak jednoduše volá metody tohoto rozhraní či abstraktního objektu a systém CORBA zajistí automatické vykonání odpovídající implementace těchto metod na straně serveru a vrátí výsledek. Na obrázku 6.6 jsou zobrazeny definice rozhraní distribuovaných objektů klienta a serveru. Tyto metody zajišťují realizaci čtyř služeb, které jsou vyžadovány od konkrétní implementace komunikačního subsystému.

Realizace většiny služeb je prováděna voláním jedné z metod rozhraní `IServer`. Konfiguraci verifikátoru zajišťuje metoda `configure(...)`, přenos textových souborů metoda `transferTextFile(...)` a spuštění verifikace zase metoda `verify(...)`. Nejkomplikovanější je implementace přeposílání vyžádaných událostí klientovi, ta je výrazně ovlivněna samotným systémem CORBA. Registraci klienta, jako odběratele specifického typu událostí, lze jednoduše provést pomocí metod `subscribeForXXXEvents(...)`. Otázkou je jak zasílat

[6] Pro základní informace o systému CORBA viz. http://en.wikipedia.org/wiki/CORBA



Obrázek 6.6: Rozhraní CORBA klienta a serveru

tyto události zpět klientovi. Komunikace pomocí systému CORBA je ze své podstaty jednosměrná. Klient volá metody serveru a jediná data, která získá zpět, jsou návratové hodnoty volaných metod. Založit způsob přenosu událostí na vyzývání, tzv. *pollingu*, kdy klient bude v periodických intervalech volat určitou metodu serveru vracející vyžadovanou událost, je neefektivní. Nejlepším řešením je vytvořit separátní komunikační kanál, kterým bude server zasílat vyžadované události klientovi. Tento přístup je použit i v případě popisované implementace.

Distribuovaný objekt serveru je tzv. perzistentní objekt. IOR tohoto objektu se nemění ani pokud tento objekt zanikne a je následně opět vytvořen. Tento druh objektu také umožňuje lokalizaci pomocí `corbaloc` URL. Klient vytváří na své straně tzv. dočasný distribuovaný objekt, jenž implementuje rozhraní `IClient`. Tento druh objektu má odlišné IOR při každém vytvoření, tedy jeho IOR je platné pouze po dobu jeho života, tedy po dobu běhu grafického uživatelského rozhraní. Klient při realizaci přeposílání vyžádaných událostí volá metodu `forwardEventsTo(...)`, které jako parametr předá IOR vytvořeného dočasného distribuovaného objektu. Server pomocí získaného IOR lokalizuje tento objekt a spustí na své straně proces, jenž bude zajišťovat přeposílání událostí tomuto objektu. Tento proces se zaregistruje jako odběratel vyžádaných typů událostí a při přijetí nějaké události ji přepošle voláním metody `forwardCustomEvent(...)` distribuovanému objektu klienta.

Kapitola 7

Závěr a zhodnocení

7.1 Směry dalšího vývoje

Oba vytvořené nástroje jsou vysoce modulární s cílem umožnit jejich další rozšíření. Z hlediska verifikátoru se zde nabízejí možnosti optimalizace jednotlivých verifikačních modulů nebo jejich nahrazení moduly, jenž řeší daný problém jiným postupem. V případě grafického uživatelského rozhraní lze přidat další zásuvné panely, jenž například lépe reprezentují výsledky procesu verifikace. Verifikátor a grafické uživatelské rozhraní nemusí být dokonce ani rozšiřovány, ale třeba použity jako základ při návrhu jiných programů, kterým mohou poskytnout služby pro komunikaci, protokolování, konfiguraci, lokalizaci či správu zásuvných modulů.

Lze ovšem zmínit alespoň několik slibných rozšíření z pohledu problému řešeného implementovanými nástroji:

- Vytvořit zásuvné panely pro vizualizaci vytvořeného grafu omezení a prohledávacího stromu.
- Vytvořit verifikační modul řešící ověřování validity zadané vlastnosti jako SMT (*Satisfiability Modulo Theories*) problém. Programy řešící SMT problém, tzv. *SMT solvery*, dnes umí řešit problémy zadané ve formě QF_UFIDL formulí.
- Paralelizovat verifikační proces. Řešení některých části verifikačního procesu lze rozdělit na více procesorů nebo jader, což by mohlo výrazně urychlit celkovou dobu verifikace.

7.2 Závěr

Cílem této práce bylo navrhout a implementovat nástroj pro formální verifikaci systémů specifikovaných jazykem RT logiky.

Vytvořený nástroj řeší verifikaci zadané vlastnosti systému postupy popsanými v kapitole 4. Celý proces verifikace je rozdělen do čtyř modulů, které řeší jednotlivé kroky verifikace. Správnost výpočtu byla ověřena na několika testovacích příkladech, které lze nalézt v přílohách.

Nad rámec zadání pak byla vytvořena aplikace, jenž poskytuje grafické uživatelské rozhraní pro vytvořený nástroj. Tato aplikace usnadňuje použití a práci s nástrojem.

Literatura

[1] Bradley, A. R.; Manna, Z.: *The Calculus of Computation*. Berlin, Heidelberg, New York: Springer, 2007, ISBN 978-3-540-74112-1.

[2] Cheng, A. M. K.: *Real-Time Systems: Scheduling, Analysis, and Verification*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2002, ISBN 978-0-471-18406-5.

[3] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns*. Addison-Wesley Publishing Company, 1994, ISBN 0-201-63361-2.

[4] Grisby, D.; Lo, S.-L.; Riddoch, D.: *The omniORB version 4.1 User's Guide*. Červenec 2007.
Dostupné na URL: <http://omniorb.sourceforge.net/omni41/omniORB.pdf>

[5] Laplante, P. A.: *Real-Time Systems Design and Analysis (Third Edition)*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2004, ISBN 978-0-471-22855-4.

[6] Object Management Group, Framingham, Massachusetts: *The Common Object Request Broker: Architecture and Specification — Version 2.6*. Prosinec 2001.
Dostupné na URL: <http://www.omg.org/cgi-bin/doc?formal/01-12-01>

[7] Tarjan, R. E.: Enumeration of the Elementary Circuits of a Directed Graph. Technická zpráva, Ithaca, NY, USA, 1972.

Přílohy

Příklady specifikací systémů reálného času

Železniční přejezd

Přejezd obsahuje pouze jedinou kolej, v určitý čas tedy může projíždět pouze jediný vlak. Systém je složen z komponent vlaku, vlakového senzoru, ovladače závor a samotných závor. Úkolem ovladače závor je zajistit, aby v době průjezdu vlaku železničním přejezdem se na křížení cesty a kolejí nevyskytovalo žádné auto. Lze předpokládat, že tento úkol je vždy splněn, pokud jsou závory v době průjezdu vlaku sklopeny.

Specifikace systému:

- Když se vlak přiblíží k vlakovému senzoru a je zachycen tímto senzorem, je zaslán signál ovladači závor, který začne pomalu sklápět závory před železničním přejezdem.
  - Závora bude sklopena do 30 sekund od doby, kdy se vlak přiblížil a byl zachycen senzorem.
  - Sklopení závory trvá alespoň 15 sekund.

Bezpečnostní tvrzení:

- Pokud vlak potřebuje alespoň 45 sekund pro uražení cesty od senzoru k železničnímu přejezdu a průjezd vlaku je dokončen do 60 sekund od doby, kdy byl vlak zachycen senzorem, pak je zajištěno, že v době doražení vlaku k železničnímu přejezdu jsou závory sklopeny a vlak opustí železniční přejezd během 45 sekund od doby, kdy bylo dokončeno sklápění závor.

Specifikace systému ve formě omezených RTL formulí:

$$\forall x \; @(TrainApproach, x) \leq @(\uparrow Downgate, x) \wedge$$

$$@(\downarrow Downgate, x) \leq @(TrainApproach, x) + 30$$

$$\forall y \; @(\uparrow Downgate, y) + 15 \leq @(\downarrow Downgate, y)$$

Bezpečnostní tvrzení ve formě omezených RTL formulí:

$$\forall t \forall u \; @(TrainApproach, t) + 45 \leq @(Crossing, u) \wedge$$

$$@(Crossing, u) < @(TrainApproach, t) + 60 \rightarrow$$

$$@(\downarrow Downgate, t) \leq @(Crossing, u) \wedge$$

$$@(Crossing, u) \leq @(\downarrow Downgate, t) + 45$$

Specifikace systému ve formě programového zápisu:

```
V x (@(% TrainApproach, x) =< @(^ Downgate, x) /\
@($ Downgate, x) =< @(% TrainApproach, x) + 30);
V y (@(^ Downgate, y) + 15 =< @($ Downgate, y))
```

Bezpečnostní tvrzení ve formě programového zápisu:

```
V t V u ((@(% TrainApproach, t) + 45 =< @(% Crossing, u) /\
@(% Crossing, u) < @(% TrainApproach, t) + 60)
-> (@($ Downgate, t) =< @(% Crossing, u) /\
@(% Crossing, u) =< @($ Downgate, t) + 45))
```

Raketový systém stíhacího letounu

Stíhačka je vybavena raketami na boj na dálku. Po zaměření rakety je nutno ji vypustit. Je potřeba zajistit, aby došlo ke včasnému otevření zbraňové šachty a aby raketa zažehla motor v bezpečné vzdálenosti od letadla. Také je nesmírně důležité, aby se raketa stihla před zásahem cíle odjistit.

Specifikace systému:

- Při aktivaci systému pro zaměření cíle dojde k zahájení otevírání zbraňové šachty.
- Otevírání zbraňové šachty bude dokončeno do 4 sekund od aktivace systému pro zaměřování.
- Uplné otevření zbraňové šachty trvá alespoň 2 sekundy.
- Po otevření šachty lze kdykoliv vypustit raketu pokud je zaměřen cíl, ta se nejprve uvolní ze zbraňové šachty, což trvá maximálně 1 sekundu, poté začne padat pod letadlo.
- Po uvolnění rakety bude zbraňová šachta jestě 3 sekundy otevřena a poté se začne zavírat.
- Uzavření je stejně časově náročné jako otevírání, tzn. trvá minimálně 2 sekundy a maximálně 4 sekundy.
- Po uplynutí 1 sekundy od uvolňení rakety dojde k aktivaci polohového čidla, které vyhodnocuje pozici rakety vzhledem k letadlu.
- Čidlo vždy maximálně do 2 sekund vyhodnotí, že vzdálenost od letadla je bezpečná pro zažehnutí raketového pohonu (čas se může lišit podle rychlosti pádu rakety a směru pohybu stíhačky, ale při jakémkoliv pohybu a manévrech se stíhačka do 2 sekund dostatečně vzdálí od rakety).
- Podle pohybu rakety a stíhačky poté dojde do 3 sekund od zažehnutí raketového pohonu k odjištění rakety.
- Odjištění rakety je ale vždy provedeno alespoň po 2 sekundách.

Bezpečnostní tvrzení:

- Pokud bude zaměřování cíle dokončeno do 5 sekund od okamžiku započetí zaměřování a pilot do 2 sekund po zaměření vypustí raketu, je zaručeno, že do 15 sekund bude k nepřátelskému letadlu mířit odjištěná raketa.

Specifikace systému ve formě omezených RTL formulí:

$$\forall x\ @(\uparrow MissileAimed, x) =< @(\uparrow MissilePitOpen, x) \land$$
$$@(\downarrow MissilePitOpen, x) =< @(\uparrow MissileAimed, x) + 4$$
$$\forall y\ @(\uparrow MissilePitOpen, y) + 2 =< @(\downarrow MissilePitOpen, y)$$
$$\forall z\ @(\downarrow MissilePitOpen, z) =< @(MissileReleased, z) \land$$
$$@(MissileReleased, z) =< @(\downarrow MissilePitOpen, z) + 1$$
$$\forall m\ @(\uparrow MissilePitClose, m) =< @(MissileReleased, m) + 3 \land$$
$$@(\uparrow MissilePitClose, m) >= @(MissileReleased, m) + 3$$
$$\forall n\ @(\uparrow MissilePitClose, n) + 2 =< @(\downarrow MissilePitClose, n)$$
$$\forall o\ @(\uparrow MissilePitClose, o) + 4 >= @(\downarrow MissilePitClose, o)$$
$$\forall p\ @(MissileSensorActive, p) =< @(MissileReleased, p) + 1 \land$$
$$@(MissileSensorActive, p) >= @(MissileReleased, p) + 1$$
$$\forall r\ @(MissileSensorActive, r) =< @(MissileJetOn, r) \land$$
$$@(MissileJetOn, r) =< @(MissileSensorActive, r) + 2$$
$$\forall s\ @(MissileTriggered, s) =< @(MissileJetOn, s) + 3$$
$$\forall t\ @(MissileJetOn, t) + 2 =< @(MissileTriggered, t)$$

Bezpečnostní tvrzení ve formě omezených RTL formulí:

$$\forall w\ @(\uparrow MissileAimed, w) =< @(\downarrow MissileAimed, w) + 5 \land$$
$$@(\downarrow MissileAimed, w) =< @(MissileReleased, w) \land$$
$$@(MissileReleased, w) =< @(\downarrow MissileAimed, w) + 2 \rightarrow$$
$$@(MissileTriggered, w) =< @(\uparrow MissileAimed, w) + 15$$

Specifikace systému ve formě programového zápisu:

```
V x (@(^ MissileAimed, x) =< @(^ MissilePitOpen, x) /\
@($ MissilePitOpen, x) =< @(^ MissileAimed, x) + 4);
V y (@(^ MissilePitOpen, y) + 2 =< @($ MissilePitOpen, y));
V z (@($ MissilePitOpen, z) =< @(% MissileReleased, z) /\
@(% MissileReleased, z) =< @($ MissilePitOpen, z) + 1);
V m (@(^ MissilePitClose, m) =< @(% MissileReleased, m) + 3 /\
@(^ MissilePitClose, m) >= @(% MissileReleased, m) + 3);
V n (@(^ MissilePitClose, n) + 2 =< @($ MissilePitClose, n));
V o (@(^ MissilePitClose, o) + 4 >= @($ MissilePitClose, o));
V p (@(% MissileSensorActive, p) =< @(% MissileReleased, p) + 1
/\ @(% MissileSensorActive, p) >= @(% MissileReleased, p) + 1);
V r (@(% MissileSensorActive, r) =< @(% MissileJetOn, r) /\
@(% MissileJetOn, r) =< @(% MissileSensorActive, r) + 2);
V s (@(% MissileTriggered, s) =< @(% MissileJetOn, s) + 3);
V t (@(% MissileJetOn, t) + 2 =< @(% MissileTriggered, t))
```

Bezpečnostní tvrzení ve formě programového zápisu:

```
V w ((@(^ MissileAimed, w) =< @($ MissileAimed, w) + 5 /\
@($ MissileAimed, w) =< @(% MissileReleased, w) /\
@(% MissileReleased, w) =< @($ MissileAimed, w) + 2) ->
(@(% MissileTriggered, w) =< @(^ MissileAimed, w) + 15))
```

Kulometný systém stíhacího letounu

Stíhačka je vybavena výkonným kulometem pro boj z blízka, ten se ovšem při střílení zahřívá a pokud zahřátí překročí únosnou mez, může dojít k jeho poškození a znefunkčnění. Této situaci je třeba zabránit, například tak, že kulomet bude obsahovat čidlo detekující jeho teplotu a při větším zahřátí zastaví jeho činnost. Je ale také nutné zajistit, aby po opětovném poklesu teploty kulometu bylo možné znova střílet, což může opět zajistit přítomné čidlo.

Specifikace systému:

- Kulomet se nezačne zahřívat dříve než po 20 sekundách střílení.
- Teplota kulometu překročí únosnou mez do 10 sekund od počátku zahřívání.
- Úplné zahřátí kulometu trvá vždy ale nejméně 7 sekund.
- Ochlazování kulometu začne probíhat ihned po jeho vypnutí.
- Zchladnutí kulometu na přijatelnou teplotu trvá podle klimatických podmínek alespoň 3 sekundy, ale maximálně 5 sekund.
- Ihned po detekci zchlazení kulometu dojde k opětovnému povolení střelby.

Bezpečnostní tvrzení:

- Pokud čidlo detekuje zahřátí kulometu do 5 sekund od počátku zahřívání a pokles teploty kulometu do 3 sekund od ustálení přijatelné teploty je zaručeno, že nikdy nedojde k přehřátí kulometu a k opětovnému povolení střelby kulometu po zchlazení dojde do 10 sekund od vypnutí kulometu.

Specifikace systému ve formě omezených RTL formulí:

$$\forall x\ @(CanonFireStart, x) + 20 =< @(\uparrow CanonWarming, x)\ \wedge$$

$$@(\uparrow CanonWarming, x) =< @(CanonFireStart, x) + 20$$

$$\forall y\ @(\uparrow CanonWarming, y) + 10 >= @(CanonOverheat, y)$$

$$\forall z\ @(\uparrow CanonWarming, z) + 7 =< @(CanonOverheat, z)$$

$$\forall p\ @(HeatDetection, p) =< @(\uparrow CanonCooling, p)\ \wedge$$

$$@(\uparrow CanonCooling, p) =< @(HeatDetection, p)$$

$$\forall r\ @(\uparrow CanonCooling, r) + 3 =< @(\downarrow CanonCooling, r)\ \wedge$$

$$@(\downarrow CanonCooling, r) =< @(\uparrow CanonCooling, r) + 5)$$

$$\forall s\ @(ColdDetection, s) =< @(CanonCanFire, s)\ \wedge$$

$$@(CanonCanFire, s) =< @(ColdDetection, s)$$

Bezpečnostní tvrzení ve formě omezených RTL formulí:

$$\forall t\ \forall u\ \forall v\ @(HeatDetection, u) =< @(\uparrow CanonWarming, t) + 5 \wedge$$

$$@(ColdDetection, v) =< @(\downarrow CanonCooling, u) + 3 \rightarrow$$

$$@(HeatDetection, u) < @(CanonOverheat, t) \wedge$$

$$@(CanonCanFire, v) =< @(HeatDetection, u) + 10$$

Specifikace systému ve formě programového zápisu:

```
V x (@(% CanonFireStart, x) + 20 =< @(^ CanonWarming, x) /\
@(^ CanonWarming, x) =< @(% CanonFireStart, x) + 20);
V y (@(^ CanonWarming, y) + 10 >= @(% CanonOverheat, y));
V z (@(^ CanonWarming, z) + 7 =< @(% CanonOverheat, z));
V p (@(% HeatDetection, p) =< @(^ CanonCooling, p) /\
@(^ CanonCooling, p) =< @(% HeatDetection, p));
V r (@(^ CanonCooling, r) + 3 =< @($ CanonCooling, r) /\
@($ CanonCooling, r) =< @(^ CanonCooling, r) + 5);
V s (@(% ColdDetection, s) =< @(% CanonCanFire, s) /\
@(% CanonCanFire, s) =< @(% ColdDetection, s))
```

Bezpečnostní tvrzení ve formě programového zápisu:

```
V t, u, v ((@(% HeatDetection, u) =< @(^ CanonWarming, t) + 5
/\ @(% ColdDetection, v) =< @($ CanonCooling, u) + 3)
-> (@(% HeatDetection, u) < @(% CanonOverheat, t) /\
@(% CanonCanFire, v) =< @(% HeatDetection, u) + 10))
```

EBNF omezených RTL formulí

```
// Logické spojky a kvantifikátory
TOK_NOT = 'not' | '!' ;
TOK_AND = 'and' | '&' | '/\' ;
TOK_OR = 'or' | '|' | '\/' ;
TOK_IMPLY = 'imply' | '->' ;
TOK_FORALL = 'forall' | 'V' ;
TOK_EXISTS = 'exists' | 'E' ;

// Predikáty a funkční symboly
TOK_LESS = '<' ;
TOK_LEQ = '=<' ;
TOK_EQ = '=' ;
TOK_GEQ = '>=' ;
TOK_GREATER = '>' ;
TOK_PLUS = '+' ;
TOK_MINUS = '-' ;
```

```
// Typy akcí
TOK_START_ACTION = '^' ;
TOK_STOP_ACTION = '$' ;
TOK_TRANSITION_ACTION = '%' ;
TOK_EXTERNAL_ACTION = '#' ;

// Konstanty, proměnné a akce
TOK_CONSTANT = ('-')?('0'..'9')+ ;
TOK_VARIABLE = ('a'..'z')('a'..'z' | 'A'..'Z' | '0'..'9')* ;
TOK_ACTION = ('A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9')* ;

// Bílé znaky
TOK_WS = (' ' | '\t' | '\n' | '\r')+ ;

// Pravidla pro syntaktický analyzátor
rrtlformula = eformula ;
eformula = tformula eformuladot ;
eformuladot = connective tformula eformuladot | epsilon ;
tformula = predicate | TOK_NOT tformula |
           quantifiedvars tformula | '(' eformula ')' ;
predicate = efunction predsymbol efunction ;
quantifiedvars = quantifier TOK_VARIABLE nextquantifiedvar ;
nextquantifiedvar = ',' TOK_VARIABLE nextquantifiedvar |
                    epsilon ;
efunction = tfunction efunctiondot ;
efunctiondot = funcsymbol tfunction efunctiondot | epsilon ;
tfunction = ofunction | TOK_CONSTANT ;
ofunction = '@(' actiontype TOK_ACTION ',' term ')' ;
term = TOK_VARIABLE | TOK_CONSTANT ;
connective = TOK_AND | TOK_OR | TOK_IMPLY ;
quantifier = TOK_FORALL | TOK_EXISTS ;
predsymbol = TOK_LESS | TOK_LEQ | TOK_EQ |
             TOK_GEQ | TOK_GREATER ;
funcsymbol = TOK_PLUS | TOK_MINUS ;
actiontype = TOK_START_ACTION | TOK_STOP_ACTION |
             TOK_TRANSITION_ACTION | TOK_EXTERNAL_ACTION ;
```

Použití nástroje

Verifikátor v aktuální verzi 0.1 podporuje pouze operační systém Linux, v brzké době ovšem bude tato podpora rozšířena i na operační systém Windows. Grafické uživatelské rozhraní je nezávislé na operačním systému a jeho použití záleží pouze na přítomnosti platformy Java.

Následující text lze považovat za jakousi rychlou uživatelskou příručku, jenž popisuje nejdůležitější položky konfigurace a popisuje uživateli základní postup pro provedení verifikace nějaké vlastnosti zadaného systému.

Verifikátor

Základní informace

Verifikátor je program pro příkazovu řádku, nevyžaduje tedy pro svůj chod žádnou podporu grafického rozhraní a lze jej jednoduše skriptovat. Po standardní instalaci jsou jeho soubory umístěny v adresáři `build/verifier` a jeho podadresářích `modules` a `plugins`.

Interní a externí moduly

Verifikátor obsahuje několik interních a externích modulů, které mohou být využity uživatelem. Tabulka 7.1 obsahuje podrobné informace o těchto modulech.

Název modulu	Typ	Implementované rozhraní	*SID*	Umístění	Popis
Standard Output Logger	Zásuvný modul	`ILogger`	*internal-stdout*	Interní modul obsažený v `libzetav-shared.a`	Protokolovací objekt, jenž zapisuje zprávy na standardní výstup
Standard Error Logger	Zásuvný modul	`ILogger`	*internal-stderr*	Interní modul obsažený v `libzetav-shared.a`	Protokolovací objekt, jenž zapisuje zprávy na standardní chybový výstup
File Logger	Zásuvný modul	`ILogger`	*internal-file*	Interní modul obsažený v `libzetav-shared.a`	Protokolovací objekt, jenž zapisuje zprávy do zadaného souboru
Xml Config Reader	Zásuvný modul	`IConfigReader`	*internal-xml-cfg-reader*	Interní modul obsažený v `libzetav-shared.a`	Konfigurační objekt, jenž využívá XML soubory pro uložení konfigurace
Corba Communicator	Zásuvný modul	`ICommunicator`	*corba-communicator*	`plugins/libzetav-plugin-corba-communicator.so`	Komunikační subsystém využívající pro komunikaci systém CORBA
Restricted RTL Formula Parser	Verifikační modul	`Module`	*rrtl-formula-parser*	`modules/libzetav-module-rrtlfparser.so`	Modul pro zpracování omezených RTL formulí
Restricted RTL Formula Converter	Verifikační modul	`Module`	*rrtl-formula-converter*	`modules/libzetav-module-rrtlfconverter.so`	Modul pro konverzi omezených RTL formulí
Constraint Graph Builder	Verifikační modul	`Module`	*cg-builder*	`modules/libzetav-module-cgbuilder.so`	Modul pro tvorbu grafu omezení
Search Tree Builder	Verifikační modul	`Module`	*search-tree-builder*	`modules/libzetav-module-stbuilder.so`	Modul pro tvorbu prohledávacího stromu

Tabulka 7.1: Interní a externí moduly verifikátoru

Základní konfigurace

Chod verifikátoru je řízen načtenou konfigurací. I přesto, že možnosti konfigurace jsou rozmanité, pro potřeby verifikace si uživatel vystačí pouze s malou částí. Základní položky konfigurace, které jsou důležité z hlediska procesu verifikace, jsou uvedeny v tabulce 7.2.

Datový typ *multi-řetězec*, jenž se vyskytuje v tabulce, je složeným typem řetězce, který se skládá z několika řetězců oddělených navzájem znaky středník, čárka nebo svilé lomítko. Datový typ *sid* pak zastupuje řetězec identifikující zásuvný nebo verifikační modul verifikátoru.

Název konfigurační položky	datový typ hodnoty
Popis	
`subsystem/extensions/path`	*multi-řetězec*
Specifikuje cestu k adresářům se zásuvnými a verifikačními moduly verifikátoru.	
`subsystem/logging/log-level`	*číslo*
Specifikuje úroveň protokolování. Každá úroveň zahrnuje i úrovně níže. Možné úrovně jsou 0 (neprotokolovat), 1 (chyby), 2 (varování), 3 - 7 (informace), přičemž úroveň 7 vypisuje ladící informace a neměla by se normálně používat.	
`verification/run/chain`	*multi-řetězec*
Specifikuje řetězení verifikačních modulů, které určuje verifikační proces. Každá položka *multi-řetězce* reprezentuje sekci v konfiguraci obsahující informace o tomto modulu. Moduly jsou spouštěny sekvenčně v pořadí jejich výskytu v *multi-řetězci*.	
verification/module/<*položka multi-řetězce*>/name	*sid*
Specifikuje *SID* modulu, kterému přísluší tato konfigurační sekce. Verifikátor vyhledá modul s tímto *SID* a spustí ho.	
verification/module/<*položka multi-řetězce*>/input/info	*řetězec*
Řetězec obsahující informace o vstupním řetězci předaném spuštěnému modulu.	
verification/module/<*položka multi-řetězce*>/input/data	*řetězec*
Vstupní řetězec předaný vstupnímu modulu.	

Tabulka 7.2: Základní konfigurační položky verifikátoru

Verifikace

Pro provedení verifikace vlastnosti systému vyjádřené ve formě omezených RTL formulí uložených v souboru `sa-formulas.in` v adresáři verifikátoru a za předpokladu specifikace systému popsaného omezenými RTL formulemi uloženými v souboru `sp-formulas.in` v adresáři verifikátoru, jsou potřebné následující soubory:

```
build/verifier/zetav-verifier
build/verifier/zetav.cfg
build/verifier/sp-formulas.in
build/verifier/sa-formulas.in
build/verifier/modules/libzetav-module-rrtlfparser.so
build/verifier/modules/libzetav-module-rrtlfconverter.so
build/verifier/modules/libzetav-module-cgbuilder.so
build/verifier/modules/libzetav-module-stbuilder.so
```

Soubor `zetav.cfg` je základní konfigurační soubor verifikátoru, který obsahuje potřebné informace pro provedení verifikace. Obsah toho souboru pro provedení popisované verifikace bude například následující:

```
<zetav-verifier>
  <subsystem>
    <extensions>
      <path>./modules</path>
    </extensions>
    <logging>
      <log-level>3</log-level>
    </logging>
  </subsystem>
  <verification>
    <run>
      <chain>rrtlfparser|rrtlfconv|cgbuilder|stbuilder</chain>
    </run>
    <module>
      <rrtlfparser>
        <name>rrtl-formula-parser</name>
        <input>
          <info>filenames:sp:sa</info>
          <data>./sp-formulas.in;./sa-formulas.in</data>
        </input>
      </rrtlfparser>
      <rrtlfconv>
        <name>rrtl-formula-converter</name>
      </rrtlfconv>
      <cgbuilder>
        <name>cg-builder</name>
      </cgbuilder>
      <stbuilder>
        <name>search-tree-builder</name>
      </stbuilder>
    </module>
  </verification>
</zetav-verifier>
```

Modul pro zpracování omezených RTL formulí vyžaduje jako vstupní data seznam souborů obsahujících omezené RTL formule. Formule v každém souboru musí vždy patřit k specifikaci systému nebo ověřovanému bezpečnostímu tvrzení. Řetězec popisující vstupní data (`info` položka) je interpretován modulem jako *multi-řetězec*, jehož první řetězec musí být hodnota *formulas*, jenž říká modulu, že vstupní data (`data` položka), jenž jsou také interpretována jako *multi-řetězec*, obsahují názvy souborů s omezenými RTL formulemi. Další řetězce popisu vstupních dat pak musí nabývat hodnoty *sp*, pokud $n - 1$ řetězec vstupních dat obsahuje název souboru s omezenými RTL formulemi specifikace systému, nebo *sa*, pokud $n - 1$ řetězec vstupních dat obsahuje název souboru s omezenými RTL formulemi bezpečnostního tvrzení.

Verifikace se pak provede spuštěním verifikátoru bez parametrů

```
./zetav-verifier
```

Výsledkem verifikace je poté odpověď

```
System valid
```

pokud je bezpečností tvrzení odvoditelné ze specifikace systému,

```
System NOT valid
```

pokud nelze prokázat platnost tohoto bezpečnostního tvrzení, případně

```
Verification process failed
```

pokud došlo k chybě při verifikaci. Podrobnější informace, včetně různých statistik, lze nalézt v protokolu událostí. V případě výchozí konfigurace jsou chyby protokolovány na standardní chybový výstup a ostatní zprávy (včetně varování) na standardní výstup.

Grafické uživatelské rozhraní

Základní informace

Grafické uživatelské rozhraní slouží pro jednodušší použití verifikátoru. Vyžaduje pro svůj běh přítomnost platformy Java, ovšem může běžet na jiném stroji než verifikátor a komunikovat s ním přes síť.

Verifikátor

V případě použití grafického uživatelského rozhraní musí být verifikátor spuštěn v tzv. *módu serveru*, ve kterém je schopen přijímat požadavky od grafického uživatelského rozhraní. V tomto módu je automaticky spuštěn komunikační subsystém, jehož konfigurace je prováděna pomocí několika nových konfiguračních položek zobrazených v tabulce 7.3.

Název konfigurační položky	datový typ hodnoty
Popis	
`subsystem/communication/communicator-name`	*sid*
Specifikuje *SID* modulu, který obsahuje konkrétní implementaci komunikačního subsystému. Výchozím modulem je modul *corba-communicator*.	
`subsystem/communication/host`	*řetězec*
Specifikuje název počítače nebo jeho IP adresu, kde bude naslouchat server verifikátoru na příchozí požadavky klientů. Pokud tato položka není definována, bude server naslouchat na všech rozhraních v počítači.	
`subsystem/communication/port`	*řetězec*
Číslo portu, na kterém bude server naslouchat na příchozí požadavky klientů. Výchozí port je `24680`.	

Tabulka 7.3: Konfigurační položky verifikátoru v *módu serveru*

Pro potřeby komunikace většinou postačuje ponechání výchozích nastavení, kdy se použije přiložený zásuvný modul pro komunikaci založenou na systému CORBA. Pro úspěšné spuštění verifikátoru v módu serveru jsou pak potřeba následující soubory:

```
build/verifier/zetav-verifier
build/verifier/zetav.cfg
build/verifier/modules/libzetav-module-rrtlfparser.so
build/verifier/modules/libzetav-module-rrtlfconverter.so
build/verifier/modules/libzetav-module-cgbuilder.so
build/verifier/modules/libzetav-module-stbuilder.so
build/verifier/plugins/libzetav-plugin-corba-communicator.so
```

Soubor `zetav.cfg` opět obsahuje základní konfiguraci verifikátoru, tentokrát pro spuštění v *módu serveru*. Obsah toho souboru v případě naslouchání pouze na lokálním stroji může být například následující:

```
<zetav-verifier>
  <subsystem>
    <extensions>
      <path>./modules;./plugins</path>
    </extensions>
    <communication>
      <communicator-name>corba-communicator</communicator-name>
      <host>localhost</host>
      <port>24680</port>
    </communication>
  </subsystem>
</zetav-verifier>
```

Nastartování verifikátoru v *módu serveru* se pak provádí jeho spuštěním s parametrem

```
./zetav-verifier --server-mode
```

Interní a externí moduly

<table>
<tr><td rowspan="2">Název modulu</td><td>Typ</td><td>Implementované rozhraní</td><td>SID</td></tr>
<tr><td colspan="3">Umístění</td></tr>
<tr><td colspan="4">Popis</td></tr>
<tr><td rowspan="2">Xml Config</td><td>Zásuvný modul</td><td><code>ZConfig</code></td><td>internal-xml-cfg</td></tr>
<tr><td colspan="3">Interní modul obsažený v <code>zetav-gui.jar</code></td></tr>
<tr><td colspan="4">Konfigurační objekt, jenž využívá XML soubory pro uložení konfigurace</td></tr>
<tr><td rowspan="2">Verifier Frontend</td><td>Zásuvný panel</td><td><code>ZGuiModule</code></td><td>verifier-frontend</td></tr>
<tr><td colspan="3">Interní modul obsažený v <code>zetav-gui.jar</code></td></tr>
<tr><td colspan="4">Grafické uživatelské rozhraní pro verifikátor</td></tr>
<tr><td rowspan="2">Corba Communicator</td><td>Zásuvný modul</td><td><code>ZCommunicator</code></td><td>corba-communicator</td></tr>
<tr><td colspan="3"><code>plugins/zetav-gui-plugin-corbacommunicator.jar</code></td></tr>
<tr><td colspan="4">Komunikační subsystém využívající pro komunikaci systém CORBA</td></tr>
</table>

Tabulka 7.4: Interní a externí moduly grafického uživatelského rozhraní

Grafické uživatelské rozhraní obsahuje pouze pár interních a externích modulů, které mohou být využity uživatelem. Tabulka 7.4 obsahuje podrobné informace o těchto modulech.

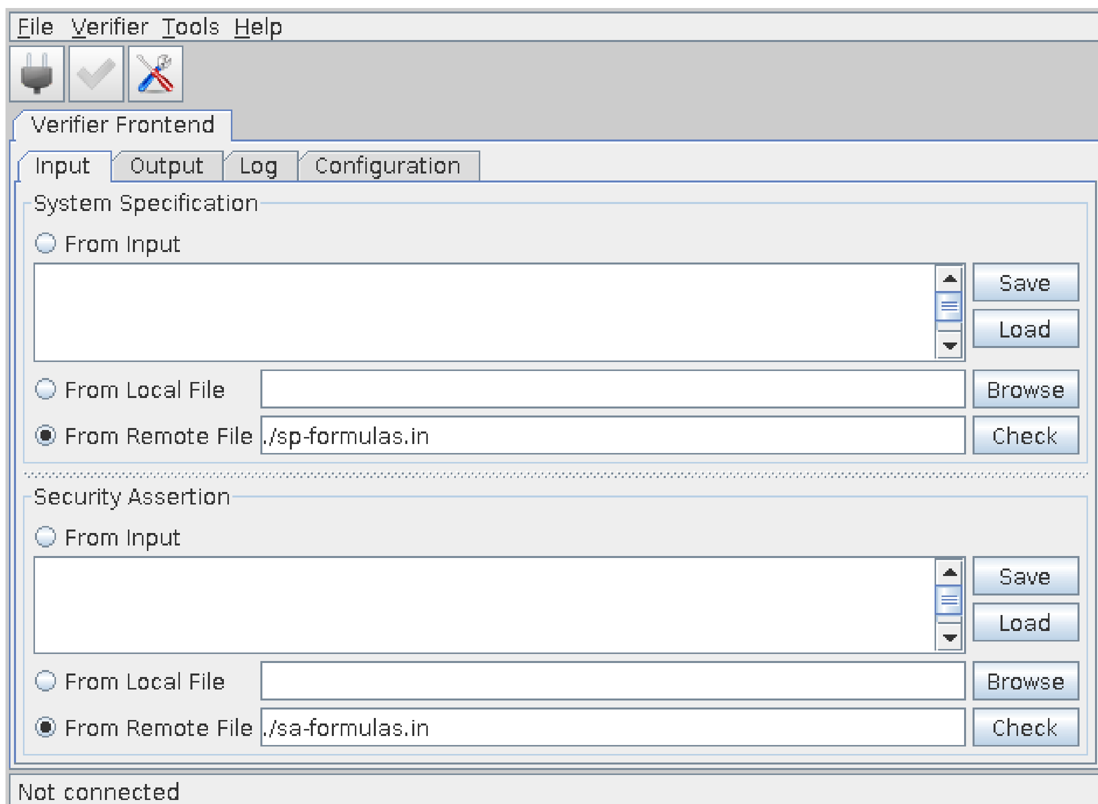
Základní konfigurace

Grafické uživatelské rozhraní nevyžaduje téměř žádnou konfiguraci. Konfigurace je uložena v souboru `zetav-gui.cfg`. Pokud není tento soubor nalezen, vytvoří se při startu grafického uživatelského rozhraní automaticky s výchozím nastavením konfigurace.

Základní položky konfigurace, jenž se vážou k subsystémům, jsou stejné jako u verifikátoru a lze je nalézt v tabulkách 7.2 a 7.3. Většina těchto položek lze nastavit v dialogu nastavení grafického uživatelského rozhraní (Tools → Settings).

Verifikace

Po spuštění grafického uživatelského rozhraní (soubor `zetav-gui.jar`) je nejprve potřeba připojit se k serveru verifikátoru. Připojení lze provést pomocí menu (Verifier → Connect) nebo nástrojové lišty (První tlačítko vlevo). Připojení vyžaduje přítomnost modulu `zetav-gui-plugin-corbacommunicator.jar` v adresáři se zásuvnými moduly.

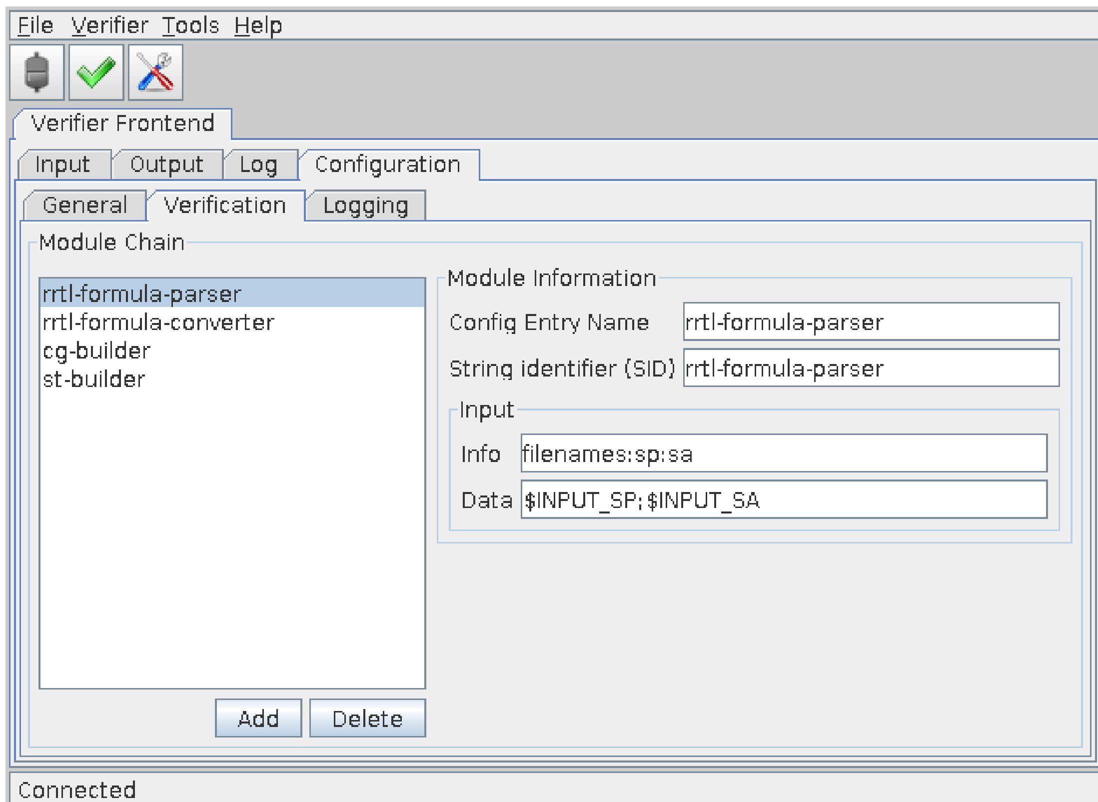


Obrázek 7.1: Specifikace vstupních formulí v grafickém uživatelském rozhraní

Po připojení se aktivuje možnost provádění verifikace. Než je ovšem možné úspěšně provést verifikaci, je potřeba provést několik nastavení. V panelu `input` (viz. 7.1) je potřeba

zadat informace o vstupních omezených RTL formulích. Panel rozlišuje omezené RTL formule, které náleží specifikaci systému a které patří k ověřovanému bezpečnostnímu tvrzení. Také umožňuje určit zdroj těchto formulí. Podporovány jsou tři typy zdrojů:

- **Input** - Vstupní formule jsou vloženy přímo do textového pole panelu `input`. Obsah tohoto textového pole je přenesen a uložen na straně verifikátoru jako soubor.
- **Local File** - Vstupní formule jsou uloženy ve specifikovaném souboru. Obsah tohoto souboru je přenesen a uložen na straně verifikátoru.
- **Remote File** - Vstupní formule jsou uloženy v souboru na straně verifikátoru.



Obrázek 7.2: Nastavení verifikačního procesu v grafickém uživatelském rozhraní

Konfigurace celého verifikačního procesu se provádí v panelu `configuration` v části `verification` (Viz. 7.2). Řetězení modulů je dáno jejich pořadím v seznamu v tomto panelu. U každého modulu lze pak specifikovat název konfigurační sekce a *SID* tohoto modulu a také vstupní data tohoto modulu. Pro specifikaci názvů souborů s vstupními formulemi lze využít proměnné (posloupnost alfanumerických znaků a podtržítka uvozená znakem dolar). Grafické uživatelské rozhraní poskytuje dvě proměnné. `INPUT_SP` obsahuje název souboru s vstupními omezenými RTL formulemi, které náleží specifikaci systému. `INPUT_SA` obsahuje název souboru s vstupními omezenými RTL formulemi ověřovaného bezpečnostního tvrzení.

Samotná verifikace se poté spouští přes menu (Verifier → Verify) nebo přes panel nástrojů (Druhé tlačítko zleva). Výsledek verifikace je pak vypsán do panelu `output` a zaprotokolované zprávy do panelu `log`.