Masarykova univerzita
Fakulta informatiky

LibFAUDES – knižnica pre prácu s disktrétnymi systémami udalostí

Bakalárska práca

Marián Pochyba

Brno, jeseň 2018

Masarykova univerzita
Fakulta informatiky

LibFAUDES – knižnica pre prácu s disktrétnymi systémami udalostí

Bakalárska práca

Marián Pochyba

Brno, jeseň 2018

Na tomto mieste sa v tlačenej práci nachádza oficiálne podpísané zadanie práce a vyhlásenie autora školského diela.

Vyhlásenie

Vyhlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Marián Pochyba

Vedúci práce: prof. RNDr. Ivana Černá, CSc.

Poďakovanie

Chcel by som poďakovať vedúcej mojej práce prof. RNDr. Ivane Černej, CSc. za záujem, rady a ochotu odpovedať na moje otázky. Takisto chcem poďakovať RNDr. Tomášovi Masopustovi, Ph.D. za poskytnuté vedomosti v problematike práce a konzultácie ohľadom implementácie, obsahu či samotného textu práce. Poďakovanie takisto patrí aj mojej rodine a blízkym, ktorí ma podporovali nielen počas písania práce, ale aj počas celého štúdia.

Zhrnutie

Práca popisuje libFAUDES, C++ softvérovú knižnicu pre reprezentáciu systémov s diskrétnymi udalosťami a pre prácu s nimi. Objasňuje prácu so samotnou knižnicou a jej základnými prvkami. Takisto predstavuje aplikácie založené na libFAUDES (skriptovací nástroj luafaudes, GUI DESTool) a možnosti rozšírenia tejto knižnice. Cieľom práce je tieto informácie sprostredkovať čitateľovi a predviesť rozšírenie knižnice o zásuvný modul. Ukázané rozšírenie implementuje algoritmy pre overenie vlastností pozorovateľnosti a relatívnej pozorovateľnosti systémov (jazykov) reprezentovaných pomocou konečných automatov.

Kľúčové slová

libFAUDES, luafaudes, lua, DESTool, pozorovateľnosť, DES

Obsah

Zoznam obrázkov

Úvod

Posledné desaťročia boli charakterizované rýchlym vývojom výpočtových, komunikačných a senzorových technológií. Tento vývoj mal za následok vznik a postupné šírenie dynamických systémov, z ktorých sú mnohé veľmi komplexné. Ich príklady nájdeme všade okolo nás: riadenie letovej prevádzky, systémy inteligentnej prepravy, systémy riadenia výroby v automobilovom priemysle či kontrolné systémy v automobiloch, distribuované systémy, počítačové a komunikačné siete a mnoho ďalších. Značná časť aktivity v týchto systémoch, niekedy aj celá, je riadená na základe ľuďmi definovaných pravidiel a ich dynamika je charakterizovaná asynchrónnym výskytom diskrétnych udalostí. Tieto udalosti môžu byť kontrolované (stlačenie klávesy klávesnice alebo tlačítka myši, zapnutie prístroja) či nekontrolované (strata paketu) a pozorované či nepozorované senzormi. Popísanú množinu dynamických systémov charakterizuje model systémov s diskrétnymi udalosťami (ďalej DES) [1].

Tieto systémy poskytujú veľa možností, ale s veľkou komplexitou, ktorú majú, je dôležitá ich dôkladná konštrukcia a práca s nimi ako prevencia voči zlyhaniam, ktoré môžu mať s takouto mierou komplexity katastrofické následky. Jednou z možností, ako prácu s časťou z nich uľahčiť, je reprezentácia DES pomocou niektorého z teoretických modelov. Tvorcovia softvérovej knižnice libFAUDES si vybrali model konečného automatu. Knižnica libFAUDES implementuje nielen tento model, ale aj škálu dátových typov, algoritmov či funkcií pre pohodlnejšiu manipuláciu s práve spracovávaným systémom. Práca popisuje najdôležitejšie z týchto implementovaných častí a priblížuje možnosti používania či rozšírenia, čomu sa venuje kapitola 1. Ďalej práca v kapitole 2 definuje pozorovateľnosť [1] a relatívnu pozorovateľnosť [2] a predstavuje algoritmy pre ich verifikáciu. Kapitola 3 potom popisuje funkcie, ktoré implementujú tieto algoritmy.

1 LibFAUDES a jej súčasti

LibFAUDES, skratka od Friedrich-Alexander University Discrete Event Systems library, je softvérová knižnica vyvíjaná na Katedre riadiacej techniky na Friedrich-Alexander University, nachádzajúcej sa v mestách Erlangen a Norimberg. Knižnica sa na DES pozerá z teoretickej perspektívy predstavenej v 80. rokoch minulého storočia dvojicou P. J. Ramadge a W. M. Wonham. Prvá funkčná verzia bola vydaná v rámci diplomovej práce Bernda Opitza v roku 2006.[1] Odvtedy k jej rozvoju prispelo a neustále prispieva veľa ľudí z oblasti DES. Knižnica pracuje so systémami s diskrétnymi udalosťami, konkrétne implementuje dátové štruktúry a algoritmy pre konečné automaty a regulárne jazyky. Jej zámerom je zredukovať úsilie potrebné k implementácii metód pre správu diskrétnych systémov a zároveň voľne poskytnúť tieto metódy verejnosti. Väčšina častí z tejto kapitoly prezentuje informácie dostupné z internetových stránok knižnice [3].

1.1 Technické informácie

Knižnica je napísaná v jazyku C++ s využitím STL (Standard Template Library) knižnice, pomocou ktorej sú definované triedy pre množiny udalostí, množiny stavov a prechodové vzťahy. Pomocou týchto množín sú definované triedy generátorov, ktoré sú založené na definícii deterministického konečného automatu. Vďaka použitiu tejto definície je implementácia algoritmov, ktoré originálne vznikli v oblasti regulárnych jazykov a automatov, jednoduchá a priamočiara. K tejto vlastnosti knižnice tiež prispieva jej infraštruktúra, ktorá ponúka možnosti ako súborový vstup a výstup či vizualizáciu generátorov pomocou nástroja dot z balíka Graphwiz. Môže sa zdať, že nepoužitie klasických polí jazyka C zbytočne zvyšuje pamäťovú náročnosť. STL však poskytuje efektívne základné algoritmy, ako je vyhľadávanie, triedenie, atď., kvôli ktorým boli autori toto nadbytočné pamäťové zaťaženie ochotní obetovať.

---

1. Methods of Supervisory Control: A Software Implementation, dostupné z https://www.rt.tf.fau.de/FGdes/archive/thesis_opitz.pdf.

Build systém knižnice dokáže spracovať a usporiadať dobrovoľnú používateľskú príručku, ktorá slúži k doplneniu informácií z C++ API dokumentácie, generovanej pomocou doxygenu. To znamená, že vývojári, ktorí chcú ilustrovať, vysvetliť či ukázať použitie svojich algoritmov pomocou príkladov, informatívneho textu a definícií, majú možnosť osloviť koncových používateľov aplikácií so základom libFAUDES.

Jednou z takýchto aplikácií, ktorá je súčasťou balíka libFAUDES, je aj skriptovací nástroj luafaudes. Dátové typy a funkcie libFAUDES sú viditeľné a použiteľné zo skriptovacieho jazyka Lua vďaka interpreteru luafaudes. Wrapper kód, ktorý spája knižnicu s jazykom Lua je automaticky generovaný build systémom s použitím nástroja SWIG.

Námaha, ktorá je potom potrebná na sprístupnenie libFAUDES prvkov, je minimálna. To je jedným z dôvodov, prečo je libFAUDES ideálnou platformou na predstavenie a implementáciu novo vyvinutých prístupov v oblasti riadenia systémov s diskrétnymi udalosťami. LibFAUDES je primárne vyvíjaná na platforme Linux, ale je bezproblémovo použiteľná aj na Microsoft Windows a Apple Mac OS X.

Zdrojové súbory knižnice sú distribuované zadarmo pod podmienkami licencie GNU LGPL (Lesser General Public License), v nádeji, že to pomôže výskumníkom v oblasti diskrétnych systémov. Okrem iného to znamená, že aplikácie vyvinuté s využitím tejto knižnice nemusia byť distribuované ako open source, ale môžu mať komerčný charakter. Knižnica podporuje mechanizmus zásuvných modulov, ktorý slúži k striktnému oddeleniu jadra knižnice od jej rozšírení a k vytváraniu a pridávaniu vlastných zásuvných modulov s organizovaným prístupom. Vývojári ďalších rozšírení sú podľa stránok knižnice vždy vítaní a vďaka licencii LGPL nie sú závislí na budúcich dizajnových rozhodnutiach tvorcov libFAUDES.

1.2 Použité nástroje

Knižnica okrem samotných dátových typov, ich metód či funkcií pre prácu s nimi poskytuje aj vlastnosti, na ktoré boli použité už existujúce nástroje. Týmito vlastnosťami sú podpora skriptovania a generovanie kódu, tak aby boli typy a funkcie libFAUDES dostupné zo skriptovacieho jazyka Lua. Takisto knižnica podporuje možnosť vizualizácie

a vstupno/výstupné operácie na súboroch. V tejto podkapitole sú stručne predstavené nástroje, vďaka ktorým sú tieto funkcionality možné.

1.2.1 Lua

Vývojári jazyka Lua o ňom tvrdia, že ide o mocný, efektívny a jednoduchý skriptovací jazyk, čomu nasvedčuje, že Lua bol použitý vo veľa priemyselných aplikáciach (napr. Adobe's Photoshop Lightroom), v stavaných systémoch (napr. middleware Ginga pre televíziu v Brazílii) či v svetovo známych hrách, ako sú World of Warcraft a Angry Birds. Jazyk bol vyvinutý na univerzite Pontifical Catholic University v Rio de Janeiro v Brazílii.

Lua okrem iného podporuje procedurálne, objektovo orientované a funkcionálne programovanie a prácu s dátami aj vo forme databáz. Lua kombinuje jednoduchú procedurálnu syntax a silné dátové konštrukcie založené na asociatívnych poliach. Lua je dynamicky typovaný jazyk, využíva interpretáciu do bajtkódu a technológiu virtuálneho stroja. Vývojárov, ktorí sa neradi starajú o pamäť, poteší automatické uvoľnovanie obsadenej nepoužívanej pamäte.[2] Integrácia Lua a libFAUDES je popísaná v podkapitole 1.5.

1.2.2 SWIG

Práve nástroj SWIG umožňuje vyššie spomínanú integráciu jazyka Lua s knižnicou libFAUDES. SWIG je softvérový vývojársky nástroj, ktorý spája programy napísané v jazyku C/C++ s možstvom vysokoúrovňových jazykov. Medzi ne patria aj bežné skriptovacie jazyky ako Javascript, Perl, PHP, Python, Ruby či práve skriptovací jazyk Lua. SWIG na základe deklarácií nájdených v hlavičkových súboroch generuje tzv. wrapper kód, ktorý umožňuje skriptovacím jazykom prístup k C/C++ kódu. SWIG taktiež umožňuje prispôsobiť generovanie kódu tak, aby vyhovovalo vlastnostiam aplikácie, v ktorej je použitý.[3]

2. Lua, dostupné z: https://www.lua.org/about.html.
3. SWIG, dostupné z: http://www.swig.org/exec.html

1.2.3 Graphwiz – dot

Pre vizualizáciu a výstup v podobe obrázkových súborov v libFAUDES je použitý nástroj dot. Tento nástroj vykresluje grafy a v libFAUDES je použitý pre vykresľovanie automatov vo forme grafov do súborov. Nachádza sa v open source balíku graphwiz.

1.2.4 Doxygen

Doxygen patrí medzi štandardne využívané nástroje použité pri generovaní dokumentácie z okomentovaných C++ zdrojových súborov. Taktiež podporuje generovanie dokumentácie pre ďalšie známe jazyky ako C, Objective-C, C#, Python, PHP alebo Java. Doxygen dokáže vygenerovať dokumentáciu vo formáte HTML, ale aj manuál použitím LaTeXu. Podporuje výstup aj vo formátoch PDF, RTF alebo vo forme stránok man systému Unix. Doxygen možno aj konfigurovať pre získanie štruktúry kódu z nezdokumentovaných zdrojových súborov.[4]

1.3 Jadro knižnice

Mechanizmus zásuvných modulov v knižnici má za následok aj oddelenie jej jadra od jej rozšírení. Dátové typy a funkcie z jadra knižnice sú využité v jej rozšíreniach a pokrývajú základnú prácu s automatmi a regulárnymi jazykmi. Táto podkapitola postupne vysvetluje základné pojmy pre zorientovanie sa v knižnici a popisuje základné dátové typy a funkcie definované jadrom knižnice.

1.3.1 Základné pojmy

Keďže knižnica implementuje teoretický model, je vhodné na porozumenie tomuto modelu a libFAUDES na úvod vysvetliť a zadefinovať niektoré pojmy. Pre niektoré z nich používa knižnica iné označenie, ako je bežne zaužívané v oblasti regulárnych jazykov, čo je ďalším dôvodom pre ich objasnenie.[5] Nasleduje prehľad základných definícií z dokumentácie knižnice.

4. Doxygen, dostupné z: http://www.doxygen.nl.
5. Niektoré, z týchto definícií sú potrebné aj pre porozumenie definíciam vlastností a algoritmom v kapitole 2.

Alphabet (abeceda). Abeceda Σ je množina znakov (symbolov). Keď nie je uvedené inak, abeceda v libFAUDES je považovaná za konečnú množinu. V kontexte systémov s diskrétnymi udalosťami, knižnice libFAUDES a tejto práce sa používa výraz *množina udalostí*.

Strings (slová). Slovo (reťazec) s nad abecedou Σ je ľubovoľná konečná postupnosť znakov $a_1 a_2 \dots a_n$ takých, že $a_i \in \Sigma$. Dĺžku slova s značíme $|s|$. Znaky abecedy Σ sú zároveň slová dĺžky 1 nad Σ.

Empty string (prázdne slovo). Prázdne slovo je klasicky značené ako ϵ, v libFAUDES manuáloch ako *epsilon*. Jeho dĺžka je nulová, $\epsilon = 0$.

Set of all strings (množina všetkých slov). Množina všetkých slov zo znakov abecedy Σ s dĺžkou väčšou ako 0 sa značí ako Σ^+. Podobne množina všetkých slov $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Concatenation (zreťazenie). Pre dve slová $s = a_1 a_2 \dots a_n$ a $r = b_1 b_2 \dots b_k \in \Sigma^*$ sa ich zreťazenie značí $s.r$, skrátene sr a $sr = a_1 a_2 \dots a_n b_1 b_2 \dots b_k$. Pre všetky $s \in \Sigma^*$: $s.\epsilon = s = \epsilon.s$.

Prefix. Slovo $s \in \Sigma^*$ je prefixom $r \in \Sigma^*$, ak existuje $t \in \Sigma^*$ také, že $st = r$. To, že s je prefix slova r, značíme ako $s \leq r$.

Formal languages (formálne jazyky). *Formálny jazyk* nad abecedou Σ je podmnožina $L \subseteq \Sigma^*$. Formálny jazyk, ktorý je možno reprezentovať konečným automatom sa nazýva regulárny.

Infinite length strings (nekonečne dlhé slová). Ak je daná abeceda Σ, potom sa pomocou Σ^w značí množina všetkých nekonečne dlhých postupností $w : \mathbb{N} \to \Sigma$. Pre všetky $w \in \Sigma^w$, sa značí $w_i \in \Sigma^*$, kde $w_i < w$, množina všetkých prefixov skladajúcich sa z prvých i znakov.

Omega languages (omega jazyky). Omega jazyk nad abecedou Σ je podmnožina $B \subseteq \Sigma^w$. Množina všetkých prefixov sa značí *Prefix(B)* $= \{s \in \Sigma^* \mid \exists w \in B : s < w\}$.

Generator (generátor). V libFAUDES sa pre deterministický konečný automat používa označenie generátor. Je to teda konečná reprezentácia pre (regulárne) formálne jazyky. V kontexte systémov diskrétnych udalostí sú generátory používané na modelovanie dynamického správania tak, že zodpovedajúci jazyk pozostáva zo všetkých slov z udalostí (symbolov), ktoré dokážu v systéme nastať.

Generátor je v knižnici definovaný ako pätica $G = (Q, \Sigma, \delta, Q_0, Q_m)$, kde:

- Q je množina stavov;
- Σ je množina udalostí (abeceda);
- δ je prechodová relácia s prechodmi z $Q \times \Sigma \times Q$;
- Q_0 je množina iniciálnych stavov;
- Q_m je množina koncových/akceptujúcich stavov.

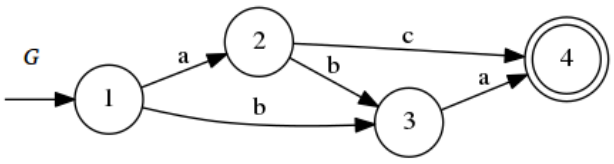
Relácia δ môže byť rozšírená tak, že jej druhým argumentom môže byť slovo. Pre všetky $q \in Q$, $a \in \Sigma$ a $s \in \Sigma^*$ nech:

- $\delta(q, a) := \{q' \in Q \mid (q, a, q') \in \delta\}$;
- $\delta(q, sa) := \delta(\delta(q, s), a)$.

Generovaný jazyk $L(G)$ a akceptovaný jazyk $L_m(G)$ pre automat G sú potom definované ako:

- $L(G) := \{s \in \Sigma^* \mid \delta(Q_0, s) \neq \emptyset\}$;
- $L_m(G) := \{s \in \Sigma^* \mid \delta(Q_0, s) \cap Q_m \neq \emptyset\}$.

Pre vysvetlenie, nech $G = (\{1, 2, 3, 4\}, \{a, b, c\}, \delta_G, \{1\}, \{4\})$ je automat s prechodovou reláciou δ_G definovanou tak, ako je znázornená na obr 1.1. Automat G generuje jazyk $L(G) = \{\epsilon, a, b, ac, ab, ba, aba\}$ a akceptuje jazyk $L_m(G) = \{ac, ba, aba\}$.



Obr. 1.1: Graf automatu G

1.3.2 Základné dátové typy

Knižnica definuje značný počet dátových typov, ktoré umožňujú reprezentovať abecedy, automaty a teda jazyky, systémy a pracovať s týmito prvkami DES. V tejto kapitole sú spomenuté dôležité typy pre pochopenie práce s libFAUDES. K tým najdôležitejším z nich je pridaný popis a prehľad podporovaných operácií.

Množinové typy

Template<class T, class Cmp = std::less<T>>
class faudes::TBaseSet<T, Cmp>
Táto šablónová trieda je implementovaná na základe STL množinovej šablóny. Poskytuje základy STL rozhrania dôležité pre libFAUDES, t. j. implementuje všetky klasické funkcie s množinami v jazyku C++ ako:

- vyhľadávanie a dotaz na existenciu prvku v množine;
- pridanie a odobranie prvku či množiny prvkov;
- zistenie počtu prvkov a dotaz na prázdnosť množiny;
- množinové operácie a porovnávanie pomocou operátorov.

Ako každá trieda v jazyku C++ má konštruktor, kopírovací konštruktor a deštruktor a podporuje prácu s iterátormi.

TBaseSet je predkom v hierarchii tried pre všetky kontajnery použité pri práci s automatmi v libFAUDES:

- IndexSet (indexy);
- TaIndexSet (indexy s atribútmi);
- SymbolSet (symbolické názvy);
- NameSet (indexy so symbolickými názvami);
- TaNameSet (indexy s atribútmi a symbolickými názvami);
- TTransSet (prechody v špecifickom poradí);
- TaTransSet (prechody v klasickom poradí s atribútmi).

Faudes::IndexSet
Trieda IndexSet je nadstavbou triedy TaBaseSet, ktorá pracuje s indexmi, t. j nezápornými hodnotami prirodzených čísiel. Index s hodnotou 0 je rezervovaný pre indikáciu neplatného indexu. Pokus vložiť index 0 vyvolá výnimku. Trieda takisto poskytuje funkcie čítania a zápisu pre súbory, typ std::string či konzolu. Z tejto triedy ďalej dedia triedy TaIndexSet, NameSet a TaNameSet.

Faudes::NameSet
Trieda NameSet reprezentuje množinu indexov spolu s ich symbolickými názvami, ktoré sú v rámci tejto triedy povinné. Obsahuje ukazateľ do symbolickej tabuľky, ktorá slúži na udržiavanie týchto mien. Štandardne sa použije statická tabuľka s názvom *SymbolTable*, ktorá je v rámci jednej relácie globálna. To znamená, že ak sa aj index nachádza vo viacerých množinách, jeho symbolický názov je vždy rovnaký a uložený práve v tejto tabuľke. Ak je definované makro **FAUDES_CHECKED**, práca s neznámym názvom alebo indexom bez názvu vyvolá výnimku. Keďže názvy sú povinné, vstupné a výstupné operácie preferujú prácu s nimi pred prácou s indexami.

Template<class Cmp = TransSort::X1EvX2>
class faudes::TTransSet< Cmp >
Táto trieda reprezentuje v libFAUDES prechodovú reláciu vo forme množiny prechodov. Trieda umožňuje špecifikovať alternatívne utriedenie množiny pomocou šablónového parametra. Základný parameter je TransSort::X1EvX2, ktorý určuje, že množina bude utriedená primárne podla stavov, z ktorých prechod vychádza, sekundárne podľa udalostí a na koniec podľa stavov, v ktorých prechod končí. TTransSet ďalej umožňuje pri množinových operáciach charakterizovať vlastnosti, ktoré musia dané prechody spĺňať, čo je využiteľné napr. pre zmazanie prechodov s určitou udalosťou.

Varianty množinových tried **TaIndexSet**, **TaNameSet** a **TaTransSet** poskytujú okrem vyššie zmienených funkcionalít aj možnosť pracovať s atribútmi jednotlivých prvkov. Ich šablónovým parametrom je trieda popisujúca tieto atribúty. Možno definovať vlastnú atribútovú triedu alebo použiť napríklad triedu **AttributeCFlags**, ktorá pre mno-

žiny udalostí poskytuje prácu s vlastnosťami, ako je kontrolovateľnosť, pozorovateľnosť, atď.

Najbežnejšie používané množinové triedy, ktoré postačia pre pochopenie a implementáciu základných algoritmov, sú pomenované typy:

- **EventSet** = NameSet je jednoduchá množina udalostí.
- **Alphabet** = TaNameSet<AttributeCFlags> je množina udalostí, ktorá podporuje prácu s atribútmi jednotlivých udalostí. Tieto atribúty sú typu AttributeCFlags.
- **StateSet** = IndexSet je množina pre prácu so stavmi generátora.
- **TransSet** = TTransSet<TransSort::X1EvX2> je množina pre manipuláciu s prechodovými vzťahmi generátora.

Triedy pre automaty

Keďže automat je pätica $G = (Q, \Sigma, \delta_G, Q_0, Q_m)$, tak základnými členmi tried reprezentujúcich automaty v libFAUDES sú:

- mpStates – množina stavov Q typu StateSet;
- mpAlphabet – abeceda Σ typu EventSet;
- mpTransRel – prechodová relácia δ_G typu TransSet;
- mpInitStates – množina iniciálnych stavov Q_0 typu StateSet;
- mMarkedStates – množina koncových stavov Q_m typu StateSet.

Faudes::Generator
Triedou, z ktorej všetky ostatné typy generátorov v knižnici dedia, je trieda vGenerator, taktiež prístupná pomocou pomenovaného typu Generator. Trieda poskytuje metódy pre vstup a výstup (vrátane výstupu vo forme obrázkového súboru), prácu so stavmi automatu, množinou udalostí či prechodovou reláciou. Toto je demonštrované na obr 1.2. Taktiež trieda obsahuje metódy pre dotazy na niektoré základné vlastnosti, ako je dosiahnuteľnosť stavov alebo kompletnosť automatu, či metódy pre úpravu automatu do požadovaného stavu, napr. odstránenie nedosiahnuteľných stavov.

Stavy, udalosti a prechody automatu reprezentovaného touto triedou môžu byť adresované 3 spôsobmi:

- indexom (efektívne) – zahŕňa vyhľadávanie na usporiadanej množine;
- názvom (neefektívne) – zahŕňa dve vyhľadávania na dvoch usporiadaných množinách;
- iterátorom (veľmi efektívne) – zahŕňa dereferenciu ukazateľov.

Okrem typu Generator obsahuje knižnica aj triedy:

- **faudes::System** = TcGenerator<AttributeVoid, AttributeVoid, AttributeCFlags, AttributeVoid>,
- **faudes::TaGenerator<GlobalAttr, StateAttr, EventAttr, TransAttr>**,

kde obe navyše poskytujú prácu s atribútmi globálnymi, stavovými, udalostnými či prechodovými. Tieto atribúty možno špecifikovať pomocou šablónových parametrov týchto typov. Trieda System pracuje primárne s atribútmi (typu AttributeCFlags) pre množinu udalostí.

1.3.3 Základné algoritmy a operácie

Pre lepšiu predstavu a pochopenie možností knižnice je nižšie uvedený krátky prehľad základných algoritmov a operácií pre prácu s triedami z menného priestoru faudes: Generator, System, EventSet a Alphabet. K operáciam je uvedený názov funkcie (bez parametrov), pod ktorým sa nachádza v knižnici:

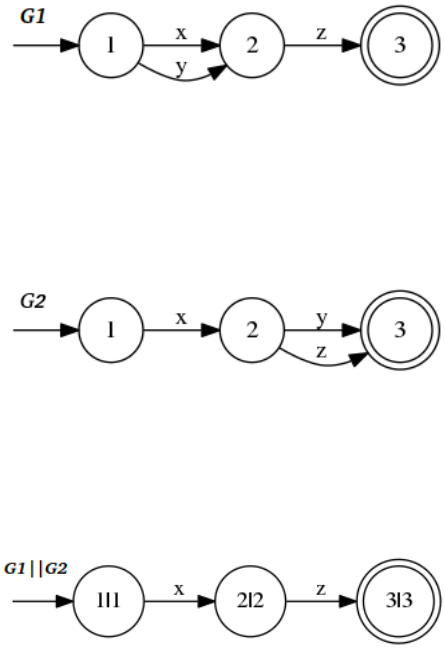
- test, či je automat deterministický – Deterministic,
- test, či automat neobsahuje nedosiahnuteľné stavy – IsAccessible;
- test, či automat neobsahuje nadbytočné stavy – IsCoaccessible;
- test, či je automat bez dosiahnuteľných a nadbytočných stavov – IsTrim;
- test na kompletnosť automatu – IsComplete.

Knižnica obsahuje aj varianty vyššie uvedených funkcií bez predpony Is, ktoré namiesto testu upravia automat na požadovaný stav. Napr. funkcia *Deterministic* implementuje algoritmus determinizácie nedeterministického automatu. Ďalšie základné operácie:

- test na prázdnosť jazyka $L_m(G)$ - IsEmptyLanguage,
- označenie všetkých stavov automatu za koncové - MarkAllStates;
- paralelná kompozícia automatov - Parallel;
- produkt automatov - Product;
- Kleeneho uzáver - KleeneClosure;
- prefixový uzáver - PrefixClosure.

```cpp
#include "libFAUDES/include/libfaudes.h"

int main() {
    using namespace faudes;

    // create event set variable
    EventSet alphabet_G1;
    // insert events "x" and "z" into set
    alphabet_G1.Insert("x");
    alphabet_G1.Insert("y");

    // create generator variable
    Generator G1;
    G1.Name("Generator G1");
    // insert initial state "1"
    G1.InsInitState("1");
    // insert states "2" and "3"
    G1.InsState("2");
    G1.InsState("3");
    // set state "3" as marked
    G1.SetMarkedState("3");
    // insert events from set "alphabet_G1"
    // to event set of G1
    G1.InsEvents(alphabet_G1);
    // insert event "z" to event set of G1
    G1.InsEvent("z");
    // set transition relation of G1
    G1.SetTransition("1", "x", "2");
    G1.SetTransition("1", "y", "2");
    G1.SetTransition("2", "z", "3");
    G1.GraphWrite("data/thesis_G1.png");

    // read generator from a file
    Generator G2("data/thesis_G2.gen");

    Generator result;
    // parallel composition  G1 and G2
    Parallel(G1, G2, result);

    // print result
    result.Write();

    // create image of result generator
    result.GraphWrite("data/thesis_G1||G2.png");

    return 0;
}
```

(a) C++ kód



(b) Automaty G1, G2 a G1 | | G2

Obr. 1.2: Ukážka použitia knižnice s grafickým výstupom

1.4 Zásuvné moduly

Ako už bolo uvedené v úvode tejto práce, knižnica pracuje s mechanizmom zásuvných modulov. To znamená, že hlavné komponenty knižnice (jadro – základné dátové typy a funkcie) sú striktne oddelené od jej rozšírení a postupne implementované časti, ktoré spolu nejakým spôsobom súvisia, sú rozdelené do častí - zásuvných modulov. Zároveň to umožňuje, bežný používateľ kedykoľvek rozšíril knižnicu o časti podľa jeho predstáv.

1.4.1 Prehľad zásuvných modulov

Synthesis Plug-In. Obsahuje funkcie relevantné pre syntézu a funkcie pre verifikáciu vlastností ako kontrolovateľnosť, normalita, relatívna uzavretosť, kompletnosť a obmien omega-kontrolovateľnosti.

Observer Plug-In. Implementuje funkcie pre syntézu a verifikáciu prirodzenej observer vlastnosti a jej variácií, ktoré nadväzujú na dizajn hierarchických architektúr a architektúr modulárneho riadenia.

Observability Plug-In. Tento zásuvný modul je vyhradený pre oblasť pozorovateľnosti jazykov, ale jeho implementácia je zatiaľ len experimentálna a distribuovaná len pre pozorovanie a prípadnú diskusiu s vývojármi.

Fault Diagnosis Plug-In. Tento modul implementuje postupy pre diagnostiku chýb v DES. Chyby v DES sú často spôsobené výskytom nepozorovateľných udalostí a ich nepredvídateľného správania.

Hierarchical I/O Systems Plug-In. Obsahuje dátové typy pre reprezentáciu hierarchických systémov so vstupmi a výstupmi a funkcie pre syntézu a verifikáciu takýchto systémov.

I/O Systems Plug-In. Implementuje funkcie pre abstrakciu založené na dizajne riadenia pre systémy s meniacimi sa vstupmi a výstupmi.

Multitasking Plug-In. Tento modul obsahuje definície pre dátové typy a funkcie pre automaty s možnosťou zafarbenia koncových sta-

vov, vrátane algoritmov pre synchrónnu kompozíciu, syntézu riadenia a observer dizajn.

Coordination Control Plug-In. Obsahuje funkcie relevantné pre syntézu koordinačného riadenia, vrátane výpočtu supremálneho podmienečne kontrolovateľného jazyka.

Timed Automata Plug-In. Rozširuje knižnicu o model časovaných automatov aj s triedou pre časové podmienky.

Simulator Plug-In. Tento zásuvný model poskytuje triedy pre podporu simulácie automatov v libFAUDES.

I/O Device Plug-In. Modul mapuje logické udalosti libFAUDES na fyzické udalosti, ktoré majú byť neskôr vykonané. Tento modul nie je vo finálnej verzii a je podporovaný len na platforme Linux.

Lua Bindings Plug-In. Tento modul implementuje podporu prístupu k libFAUDES dátovým typom a funkciám zo skriptovacieho jazyka Lua. Taktiež obsahuje aplikáciu luafaudes.

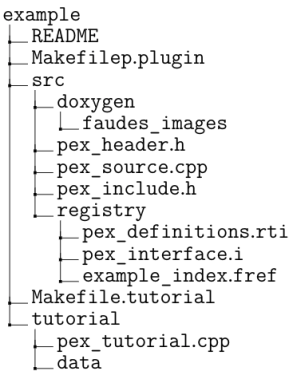
1.4.2 Rozšírenie knižnice o zásuvný modul

Rozšírenie knižnice o C++ zásuvný modul všeobecne zahŕňa nasledujúce kroky:

- implementácia nových dátových typov a algoritmov;
- zavedenie dokumentácie pomocou run-time rozhrania;
- zavedenie korešpondujúcich väzieb pre luafaudes;
- integrácia modulu s knižnicou pomocou jej build mechanizmu.

Táto časť postupne ukazuje, ako postupovať pri jednotlivých krokoch a teda rozšíriť knižnicu o vlastný zásuvný modul. Toto je ukázané na vzorovom module vytvorenom pre tieto účely. Každá distribúcia knižnice obsahuje podobný modul pre zjednodušenie procesu pridávania modulu do libFAUDES.

Adresárová hierarchia



Obr. 1.3: Adresárová hierarchia vzorového modulu

Názov najvyššieho adresára (**example**) implicitne definuje názov zásuvného modulu vkladaného do knižnice. Každý zo zdrojových a hlavičkových súborov začína prefixom (**pex_**), ktorý musí byť v rámci knižnice jedinečný. Implementácia dátových typov či algoritmov sa nachádza v adresári **example/src** spolu s korešpondujúcimi hlavičkovými súbormi a hlavičkovým súborom **pex_include.h**, pomocou ktorého pripojí build systém knižnice ostatné hlavičkové súbory do hlavného hlavičkového súboru knižnice. Adresárová hierarchia vzorového zásuvného modulu je ukázaná na obr 1.3

Doxygen dokumentácia

Dokumentácia je tvorená pomocou nástroja doxygen, ktorý je build systémom spustený na všetky *.cpp a *.h súbory. Dokumentácia sa potom objaví v C++ príručke knižnice.

Hlavičkový súbor **pex_include.h** by mal obsahovať doxygen sekciu s krátkym popisom obsahu pridávaného zásuvného modulu. Takisto každý hlavičkový súbor s deklaráciami typov a funkcií by mal obsahovať doxygen sekciu s technickým popisom a príslušnosťou k zásuvnému modulu. Ak tento popis obsahuje odkaz na obrázkový súbor, musí byť tento súbor uložený v adresári **src/doxygen/faudes_images**. Príklady vhodných hlavičkových súborov ukazuje obr 1.4.

```
/** @file pex_include.h
 * Include example plugin headers */

#ifndef FAUDES_PEX_INCLUDE_H
#define FAUDES_PEX_INCLUDE_H

#include "pex header.h"

#endif

/**
  @defgroup ExamplePlugIn Example PlugIn

  @ingroup AllPlugins

  @section Overview
  <p>
  PlugIn overview.
  </p>

  @section License
  <p>
  PlugIn license information.
  </p>
*/
```

(a) Súbor pex_include.h

```
/** @file pex_header.h
 * Example PlugIn */

#ifndef FAUDES_PEX_HEADER_H
#define FAUDES_PEX_HEADER_H

#include "pex corefaudes.h"

namespace faudes {

/**
 * Function tests if generator has more
 * than 2 states.
 *
 * @param rGen
 *    Input generator
 *
 * @return
 *    true if generator has more than
 *    2 states, false otherwise
 *
 * @ingroup ExamplePlugIn
 */

extern FAUDES API bool HasMoreThan2States(const Generator& rGen);

} //namespace faudes

#endif
```

(b) Súbor pex_header.h

Obr. 1.4: Ukážka hlavičkových súborov

Tutoriál

Každý zásuvný modul v knižnici demonštruje svoje použitie pomocou tutoriálu vo forme krátkeho C++ programu s použitím funkcií a dátových typov definovaných v tomto module. Ak tutoriály vyžadujú vstupné dáta, tieto dáta musia byť uložené v adresári **example/tutorial/data**. Vzorový tutoriál je ukázaný na obr 1.5.

```
/**
 * @file pex_tutorial.cpp
 * Tutorial, example plug-in. More text ...
 * @ingroup Tutorials
 * @include pex_tutorial.cpp
 */

#include <iostream>
#include "libfaudes.h"

int main() {

  // Apply algorithm to provided input data
  Generator gen("data/three_states.gen");
  if (HasMoreThan2States(gen)) {
      std::cout << "Gen has more than 2 states!";
  } else {
      std::cout << "Gen has less than 2 states!";
  }

  std::cout << endl;

  return 0;
}
```

Obr. 1.5: Ukážka zdrojového kódu tutoriálu

Závislosti pre vytvorenie objektových súborov tutoriálov sú definované v súbore **Makefile.tutorial**. Tento makefile musí splňovať formu tak, aby bolo možné integrovať tutoriál s build systémom knižnice. Táto forma je rovnaká ako pri súbore **Makefile.plugin** a je popísaná v časti 1.4.2.

Zdrojový kód tutorálu je potom uvedený v C++ API príručke, v sekcii *Tutorials*. To vďaka nástroju doxygen a špecifikácii, že tutoriál patrí do skupiny *Tutorials* pomocou anotácie **@ingroup Tutorials**.

Makefile.plugin

Hlavný makefile knižnice kopíruje hlavičkové súbory do adresára **libfaudes/include**, vytvára objektové súbory zo zdrojových a linkuje ich s libFAUDES. Počas spracovávania hlavného súboru makefile build systém pridáva **Makefile.plugin** každého z modulov. Všetky majú rovnakú formu ako ten zo vzorového modulu:

Definícia ciest a zdrojových súborov.

```
# Relevant paths
PEX_NAME   = example
PEX_BASE   = ./plugins/$(PEX_NAME)
PEX_SRCDIR = ./plugins/$(PEX_NAME)/src

# List source files
PEX_CPPFILES = pex_altaccess.cpp
PEX_INCLUDE = pex_include.h

# Generate paths
PEX_HEADERS = $(PEX_CPPFILES:.cpp=.h) $(PEX_INCLUDE)
PEX_SOURCES = $(PEX_CPPFILES:%=$(PEX_SRCDIR)/%)
PEX_OBJECTS = $(PEX_CPPFILES:%.cpp=$(OBJDIR)/%.o)
```

Pripojenie k automaticky generovaným hlavičkovým súborom.

```
# Append my overall include file to libfaudes.h
$(INCLUDEDIR)/$(PEX_INCLUDE): $(PEX_SRCDIR)/$(PEX_INCLUDE)
        cp -pR $< $@
        echo "#include \"$(PEX_INCLUDE)\"" >> $(INCLUDEDIR)/allplugins.h
        echo "/* example plugin configuration */" >> $(INCLUDEDIR)/configuration.h
        echo "#define  FAUDES_PLUGIN_EXAMPLE" >> $(INCLUDEDIR)/configuration.h
        echo " " >> $(INCLUDEDIR)/configuration.h
```

Pridanie modulu do build systému knižnice.

```
# Advertise plug-in to libFAUDES build system
SOURCES += $(PEX_SOURCES)
OBJECTS += $(PEX_OBJECTS)
HEADERS += $(PEX_HEADERS)
VPATH   += $(PEX_SRCDIR)
```

Run-time rozhranie

LibFAUDES poskytuje register dátových typov a funkcií ako základ pre aplikácie, cez ktoré sa používateľ môže s nimi stretnúť. Táto funkcionalita je v rámci pridávania zásuvného modulu dobrovoľná a build systém knižnice sa snaží minimalizovať úsilie pre zavedenie modulu do tohto registra. Zavedenie je vykonané pomocou XML súboru, ktorý spolu s deklaráciami funkcií obsahuje aj krátky popis, niekoľko kľúčových slov a odkaz na detailnejšiu HTML dokumentáciu. Na obr 1.6 možno vidieť súbor **src/registry/pex_definitions.rti**.

HTML dokumentácia vzorového modulu sa nachádza v súbore **src/registry/example_index.fref**. Nadpisy, hlavičky funkcií či ďalšie časti HTML stránky sú vygenerované pomocou špeciálnych tagov **ReferencePage** – nastavenie stránky, **ffnct_reference** – hlavička funkcie, **fdetails** – detailný popis a **fconditions** – podmienky pre vstupné či výstupné parametre funkcie. Ukážka tohto súboru je na obr 1.7.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE ReferencePage SYSTEM "http://www.faudes.org/dtd/1.0/referencepage.dtd">
<ReferencePage chapter="Reference" section="Example" page="Index" title="Example PlugIn">

<h1> Example PlugIn </h1>

<fsummary>
Example plug-in for demonstration purposes.
</fsummary>

<p>
This is the example plug-in that shows how to extend libFADUES.
</p>

<ffnct_reference name="HasMoreThan2States">

<fdetails/>
<p>
The function HasMoreThan2States...
</p>

<fconditions/>
<p>
Argument must be such that [...]
</p>

</ffnct_reference>

</ReferencePage>
```

Obr. 1.6: Súbor pex_definitions.rti

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE Registry SYSTEM "registry.dtd">
<Registry>

<FunctionDefinition name="Example::HasMoreThan2States"
ctype="faudes::HasMoreThan2States">

<Documentation ref="example_index.html">
Implementation of very important function.
</Documentation>
<Keywords>
Example       states
</Keywords>

<VariantSignatures>
<Signature name="Default">
<Parameter name="Gen" ftype="Generator" access="InOut"/>
</Signature>
</VariantSignatures>

</FunctionDefinition>
```

Obr. 1.7: Súbor example_index.fref

Aby build systém knižnice zaregistroval dátové typy a funkcie a vygeneroval HMTL dokumentáciu, je potrebné do súboru **Makefile.plugin** pripojiť riadky ako na obr 1.8.

```
# advertise rti defs
PEX_RTIDEFS = pex_definitions.rti
PEX_RTIFREF = example_index.fref

PEX_RTIDEFS := $(PEX_RTIDEFS:%=$(PEX_SRCDIR)/registry/%)
PEX_RTIHTML := $(PEX_RTIFREF:%.fref=%.html)
PEX_RTIFREF := $(PEX_RTIFRF:%=$(PEX_SRCDIR)/registry/%)

RTIPLUGINS += $(PEX_NAME)
RTIDEFS += $(PEX_RTIDEFS)
RTIFREF += $(PEX_RTIFREF)
RTIHTML += $(PEX_RTIHTML)
```

Obr. 1.8: Časť súboru Makefile.plugin

Väzby Lua

Aby boli dátové typy a funkcie libFAUDES dostupné zo skriptovacieho jazyka Lua, využíva knižnica nástroj SWIG. Tento nástroj automaticky konvertuje tzv. definície z rozhraní na kód jazyka C. Pre vygenerovanie väzieb pre jazyk Lua je nutné pripojiť do súboru **Makefile.plugin** riadky na obr 1.9a, ktoré oznamujú, že je potrebné spracovať jeden SWIG modul a jeden súbor s rozhraním. Súbor s rozhraním **pex_interface.i** je ukázaný na obr 1.9b.

```
# advertise luabindings
LBP_INTERFACES += $(PEX_SRCDIR)/registry/pex_interface.i
LBP_LUAMODULES += example
```

(a) Časť súboru Makefile.plugin

```
// Set SWIG module name
// Note: must match libFAUDES plug-in name
%module example

// Indicate plugin to rti function definitions
#ifndef SwigModule
#define SwigModule "SwigExample"
#endif

// Load std faudes interface
%include "faudesmodule.i"

// Extra Lua functions: copy to faudes name space
%luacode {
-- Copy example to faudes name space
for k,v in pairs(example) do faudes[k]=v end
}

// Add topic to mini help system
SwigHelpTopic("Example","Example plug-in function");

// Include RTI generated functioninterface
#if SwigModule=="SwigExample"
%include "../../../include/rtiautoload.i"
#endif
```

(b) Súbor pex_interface.i

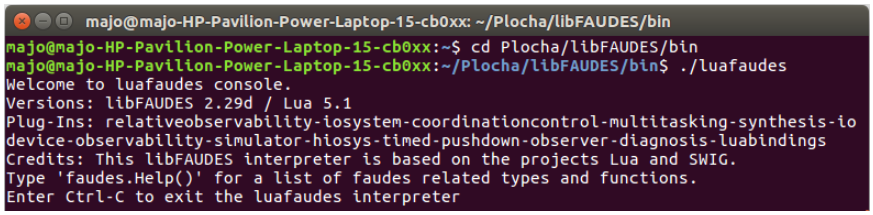
Obr. 1.9: Generovanie väzieb Lua

Záverečné kroky

1. Inštalácia knižnice, podrobne v podkapitole 1.7;
2. Vloženie adresárovej štruktúry popísanej v časti 1.4.2 do priečinka **libFAUDES/plugins**;
3. Zadanie príkazov do konzoly:
   (a) make dist-clean,
   (b) make configure,
   (c) make,
   (d) make tutorial,
   (e) make test.

1.5 Luafaudes

Luafaudes je skriptovacie prostredie založené na jazyku Lua, ktoré má prístup k dátovým typom a funkciám knižnice libFAUDES. Technicky je táto integrácia libFAUDES a jazyka Lua implementovaná v zásuvnom module *Luabindings PlugIn*. Zatiaľ čo dátové typy sú adoptované pomocou definícií rozhraní, funkcie libFAUDES sú automaticky viazané cez run-time rozhranie.



Obr. 1.10: Spustenie prostredia luafaudes

1.5.1 Použitie

Luafaudes je prirodzene spúštaná z príkazového riadku použitého operačného systému ako *./luafaudes*. Pri volaní luafaudes bez argumentov aplikácia zobrazí uvítaciu správu a je pripravená pracovať s akýmkoľvek zadaným valídnym Lua príkazom či výrazom. Spustenie luafaudes je demonštrované na obr 1.10.

Luafaudes vlastne rozširuje jazyk Lua o funckie a typy z libFAUDES a tie sú dostupné z menného priesotoru faudes, to znamená, že k žiadanej funkcii či typu sa možno dostať cez prefix faudes. Príkladom môže byť funkcia **faudes.Help()**, ktorá do konzoly vypíše prehľad základných možností vzťahujúcich sa ku knižnici. Pre prístup k prvkom knižnice bez potreby prefixu možno zavolať **faudes.MakeGlobal()**.

Luafaudes ďalej ponúka možnosť logovania príkazov zadaných do konzoly. Táto možnosť sa aktivuje volaním funkcie **InitLog(filnemae)**, kde **filename** je názov súboru, do ktorého sa budú logovať všetky postupne zadané príkazy, až pokým nie je zavolaná funkcia **CloseLog()** alebo je ukončená aplikácia. Súbor vzniknutý pri logovaní možno neskôr spustiť ako Lua skript.

Asi najdôležitejšia časť knižnice je práca s množinami udalostí a samotnými automatmi. Pre reprezentáciu udalostných množín v ja-

zyku Lua je použitá C++ trieda **EventSet** a pre generátory je to trieda **Generator**. Objekty z knižnice možno v jazyku Lua vytvoriť pomocou základného konštruktora, napr. príkaz **alph = EventSet()** vytvorí premennú alph typu EventSet. Pre volanie funkcií triedy sa používa znak dvojbodky, takže napr. pre vloženie udalosti gamma do množiny alph je potrebné do konzoly (alebo do súboru so skriptom) napísať: **alph:Insert("gamma")**.

1.5.2 Dátové typy

Jazyk Lua rozlišuje medzi základnými dátovými typmi (čísla, boolean, string) a používateľom definovanými typmi. Základné typy libFAUDES (Integer, Boolean, String) sú mapované na ich korešpondujúci typ v Lua. Premenná takéhoto typu drží hodnotu a operátor priradenia teda predáva hodnotu a možno pomocou neho kopírovať. Ostatné typy sú mapované na používateľom definované typy a premenná tohto typu drží referenciu a operátor priradenia teda predáva referenciu. Pre kopírovanie je potrebné zavolať metódu **Copy()**, prípadne metódu priradenia **Assign()**.

Jazyk Lua automaticky konvertuje hodnoty číselného typu na hodnoty typu string a opačne. Toto má svoje výhody, ale je potrebné dávať pozor pri preťažených funkciách, ktoré berú ako paramater hodnoty číselného typu aj typu string. Jednou z nich je metóda Insert triedy EventSet, kde volanie **Insert("2")**, môže byť interpretované ako volanie funkcie Insert pre číselný index 2. Pre vyhnutie sa podobným problémom je vhodné používať názvy, ktoré nezačínajú číslom.

1.5.3 Funkcie

Väčšina algoritmov implementovaných v libFAUDES je dostupných aj z prostredia luafaudes. Ich výpis možno zobraziť pomocou príkazu **Help("Functions")**. Na rozdiel od jazyka C++, funkcie v jazyku Lua predávajú parametre vždy hodnotou. To okrem iného znamená, že pomocou parametrov nemôže byť vrátená hodnota. Hodnotu možno vrátiť len pomocou výrazu **return**. Samotný jazyk Lua podporuje vrátenie viacerých hodnôt pomocou return výrazu, ale momentálna implementácia generátoru kódu podporuje generovanie Lua funkcií s jednou vrátenou hodnotou. Toto samozrejme platí len pre elemen-

tárne typy, keďže hodnota premenných používateľom definovaných typov je referencia k objektu. To znamená, že ak má funkcia z libFAUDES výstupný parameter elementárneho typu, je funkcia v Lua generovaná tak, že tento výsledok vráti pomocou return výrazu. Ak je výstupný parameter používateľom definovaného typu, vygenerovaná funkcia bude mať rovnakú deklaráciu ako v libFAUDES. Obr 1.11 demonštruje použitie dátových typov a funkcií z libFAUDES v luafades skripte.

V luafaudes skriptoch je možno definovať vlastné funkcie. Tieto funkcie môžu pracovať aj s objektami z libFAUDES či volať ďalšie funkcie z knižnice.

Tak ako aj libFAUDES aj luafaudes poskytuje funkcie pre tvorbu obrázkových súborov generátorov pomocou nástroja dot z balíka Graphviz.

```
-- Create plain generator
gen = faudes.Generator()

-- Have a name
gen:Name("simple machine")

-- Insert some states by name
gen:InsState("idle")
gen:InsState("busy")

-- Insert anonymous states
didx = gen:InsState()

-- Set name of anonymous state
gen:StateName(didx,"down")

-- Insert some events
gen:InsEvent("alpha")
gen:InsEvent("beta")
gen:InsEvent("mue")
gen:InsEvent("lambda")

-- Insert some transitions by names
gen:SetTransition("idle","alpha","busy")
gen:SetTransition("busy","beta","idle")
gen:SetTransition("busy","mue","down")

-- Indicate initial and marked states
gen:SetInitState("idle")
gen:SetMarkedState("idle")

-- Print
gen:Write()
```

Obr. 1.11: Ukážka luafaudes skriptu

1.5.4 Inštalácia

Interpreter luafaudes je časťou balíka libFAUDES s predkompilovanými verziami pre operačné systémy Linux a MS Windows. Alternatíva k spúštaniu skriptov cez príkazový riadok je grafické rozhranie DESTool, ktorému sa venuje podkapitola 1.6. Pre používanie luafaudes, je po inštalácii libFAUDES nutné nastaviť systémovú premennú **PATH** príkazom: *export PATH=$PATH:$(pwd)/libfaudes-current/bin*,kde *libfaudes-current* je názov priečinka s knižnicou, inak je potrebné ju spúšťať z adresára *libfaudes-current/bin* . Pre otestovanie inštalácie je vhodné zavolať niektorý zo vzorových skriptov z tutoriálovej sekcie Luabindings modulu. Dostupné skripty čítajú dáta z podadresára **data**, ktorý sa nachádza v priečinku so skriptom. Preto je potrebné zmeniť pracovný adresár na ten s volaným skriptom.

1.6 DESTool

DESTool je nástroj pre syntézu a analýzu systémov s diskrétnymi udalosťami. Technicky je to implementácia grafického prostredia pre knižnicu libFAUDES. Pracuje s typmi a funkciami z knižnice v tzv. projektoch, ktoré sa skladajú z premenných (týchto typov), na ktorých možno aplikovať funkcie z knižnice vo forme skriptov.

Nástroj je taktiež ako libFAUDES vyvíjaný na Friedrich-Alexander University. Koordinátorom a autorom projektu je Thomas Moor. Rozhranie a hierarchia pracovných objektov boli navrhnuté Ruedigerom Berndtom, avšak do vývoja DESToolo už prispelo mnoho ľudí.[6]

1.6.1 Inštalácia a licencia

DESTool nie je súčasťou balíka libFAUDES a je potrebné ho stiahnuť z internetových stránok nástroja a inštalovať ho osobitne. Dostupný archív s nástrojom obsahuje len binárne súbory, pretože nástroj nie je distribuovaný ako open source. To znamená, že nie je povolené modifikovať a redistribuovať DESTool, ale možno s ním pracovať bez obmedzení, dokonca aj pre komerčné účely.

6. DESTool, dostupné z :https://www.rt.tf.fau.de/FGdes/destool/index.html

Postup inštalácie a spustenie:

1. stiahnutie archívu na stránkach *https://www.rt.tf.fau.de/FGdes/download.html*;
2. extrahovanie archívu na ľubovoľné miesto v systéme;
3. inštalácia dotatočných knižníc (LSB);[7]
4. spustenie – zadaním príkazu *./faudes-destool/bin/destool* do konzoly, kde *faudes-destool* je názov extrahovaného priečinku.

1.6.2 Použitie a ovládanie

Projekty

Projekty sú entity najvyššej úrovne v DESTool, pomocou ktorých sa spracovávajú DES dizajny alebo vykonávajú analytické úlohy. DESTool projekt sa skladá z:

- vstupných a výstupných dát vo forme premenných, ktoré sú libFAUDES typu (napr. System alebo Alphabet);
- zoznamu operácií vo forme DESTool skriptu;
- DESTool simulátora, ktorý slúži na pozorovanie vzniknutého dizajnu či výsledku úlohy.

Pre prístup k týmto prvkom projektu slúžia záložky v hornej časti okna pre prehliadanie projektov, čo je ukázané na obrázku obr 1.12. Projekty môžu byť uložené a otvorené pomocou menu *File*. Aby boli všetky dáta, ktoré súvisia s projektom, v jednom súbore, je vhodné najprv projekt do tohto súboru uložiť a až potom pridávať premenné či skripty.

Premenné

Pre ukladanie operandov a výsledkov operácií služia v DESTool premenné. K premenným možno pristúpiť pomocou záložky *Variables* v prehliadači projektov. Pre vytvorenie novej premennej treba kliknúť

7. sudo apt install lsb; knižnicu LSB je potrebné nainštalovať aj pre chod libFAUDES

na možnosť *New Variable* v menu *File* alebo stlačiť Shift + Return v zozname premenných. Každá z premenných musí mať jedinečné meno a má priradené vlastnosti konfigurácie:

- Premenné sú špecifického typu z libFAUDES (napr. System alebo Alphabet); zoznam dostupných typov sa nachádza v libFAUDES používateľskej príručke; pri zmene typu premennej sa DESTool pokúsi konvertovať hodnotu premennej na hodnotu nového typu.
- Premenná môže byť umiestnená v rámci projektového súboru (možnosť *Persistent*), čo je veľmi praktické, ale pri projekte s veľkým počtom veľkých automatov nemusí byť možné.
- Premenná môže byť uložená v externom súbore (možnosť *External file*); načítanie a uloženie premennej z externého súboru možno z menu *File*, ale táto možnosť je dostupná až po uložení projektu do priečinka, keďže DESTool ukladá všetky relevanté súbory v priečinku s projektom.
- Premenné môžu mať priradenú vizuálnu reprezentáciu, ale len v prípade, ak sa pracuje s dostatočne malými automatmi (možnosť *Visual Representation*).
- Premenné môžu mať priradenú vlastnosť *Supervisor* alebo *Plant-Model*, čo určuje, ktoré generátory budú simulované.

Po vytvorení majú premenné nastavené vlastnosti *Persistent* a *Visual Representation*.

Skripty

V nástroji DESTool sú skripty zoznamy funkcií z libFAUDES, ktoré pracujú s DESTool premennými. Pri vytváraní skriptu možno:

- vložiť operácie pomocou kontextového menu;
- vybrať signatúru z rozbaľovacieho menu;
- vybratím z rozbaľovacieho menu nastaviť operandy a premenné, do ktorých budú uložené výsledné hodnoty.

Po vytvorení možno vykonanie skriptu krokovať alebo vykonať všetky kroky naraz.

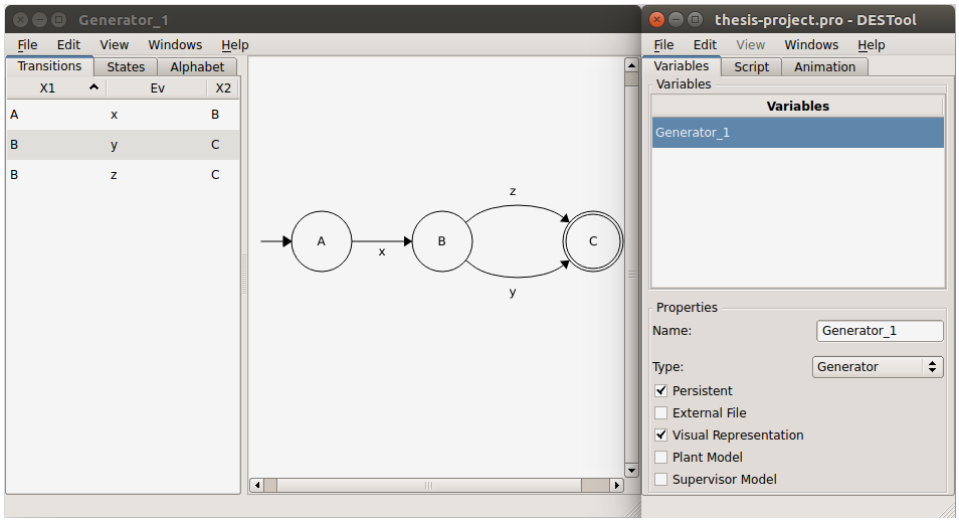
Simulácia

DESTool simulácia interpretuje generátory so synchronizovanými udalosťami pomocou tried v zásuvnom module *Simulator* v libFAUDES. Hlavným cieľom je simulovať dizajn DES a analyzovať efekty z hľadiska syntézy. DESTool simulácia neposkytuje plnú simuláciu systémov s diskrétnymi udalosťami.

V záložke *Animation* možno pomocou tlačítka *Configure* konfigurovať vlastnosti simulácie či nastaviť súbor pre jej logovanie. Túto konfiguráciu možno aj exportovať do konfiguračného súboru z menu *File*.

Samotnú simuláciu možno ovládať pomocou troch tlačítok:

- *Init*, resetuje aktuálny stav generátora na jeho iniciálny stav;
- *Step*, vykoná práve jeden prechod;
- *Revert*, vráti posledne vykonaný prechod.



Obr. 1.12: Prostredie DESTool

1.7 Inštalácia libFAUDES

V tejto podkapitole je ukázaný postup inštalácie knižnice libFAUDES pre systém Ubuntu. Predpokladá sa tu, že inštalácia knižnice prebehne na čistej inštalácii systému Ubuntu, aby sa ukázala inštalácia aj dodatočných knižníc a nástrojov pre správne fungovanie knižnice. Postup inštalácie:

1. stiahnutie a inštalácia dodatočných knižníc a nástrojov:
   (a) knižnica libreadline – *sudo apt install libreadline*,
   (b) knižnica libncurses5 – *sudo apt install libncurses5*,
   (c) knižnica LSB – *sudo apt install lsb*,
   (d) nástroj doxygen – *sudo apt install doxygen*,
   (e) nástrog SWIG – *sudo apt install swig*,
   (f) nástroj dot – *sudo apt install graphwiz*;
2. stiahnutie archívu s knižnicou (obsahuje aj binárne súbory pre Linuxové systémy);
3. extrahovanie stiahnutého archívu;
4. presunutie sa do adresára s knižnicou;
5. kompilácia knižnice zadaním príkazov do konzoly:
   (a) *make* – kompilácia jadra knižnice a zásuvných modulov,
   (b) *make tutorial* – kompilácia tutoriálov,
   (c) *make test* – kontrola priebehu kompilácie.

2 Implementované algoritmy

Pre ukážku postupu pri tvorbe zásuvných modulov pre libFAUDES sú v tejto práci použité definície vlastností z oblasti supervízneho riadenia pre systémy s diskrétnymi udalosťami. V tejto kapitole sú definované vlastnosti pozorovateľnosti a relatívnej pozorovateľnosti a vysvetlené algoritmy pre verifikáciu týchto vlastností a výpočet supremálneho relatívne pozorovateľného jazyka.

2.1 Úvodné poznámky

Pre vysvetlenie vlastností a algoritmov je potrebné pre zvyšok práce zaviesť značenie a zadefinovať použité vlastnosti či konštrukcie. Nech teda:

- $\delta(q, s)!$ a $\delta(q, s)\not!$ značí, že prechod $\delta(q, s)$ je definovaný, resp. nedefinovaný.
- $\overline{L}$ značí prefixový uzáver jazyka L, t. j. jazyk, ktorý je tvorený všetkými prefixmi všetkých slov z jazyka L.
- Automat G je neblokujúci, ak $L(G) = \overline{L_m(G)}$.
- Množinu Σ možno rozdeliť na množinu kontrolovateľných Σ_c a množinu nekontrolovateľných udalostí Σ_{uc}, resp. na množinu pozorovateľných Σ_o a nepozorovateľných udalostí Σ_{uo}.
- $P_o : \Sigma^* \to \Sigma_o^*$ značí projekciu, kde:
  1. $P_o(\epsilon) = \epsilon$;
  2. $P_o(a) = a$, ak $a \in \Sigma_o$;
  3. $P_o(a) = \epsilon$, ak $a \in \Sigma_{uo}$;
  4. $P_o(sa) = P_o(s)P_o(a)$, pre $s \in \Sigma^*$ a $a \in \Sigma$.
- Inverzná projekcia je definovaná ako:
  $P_o^{-1}(t) = \{s \in \Sigma^* : P_o(s) = t\}$.
- Projekcia aj inverzná projekcia môžu byť rozšírené na jazyky aplikovaním $P_o(s)$ a $P_o^{-1}(s)$ na všetky slová s z tohto jazyka.

- $Ac(G)$ značí všetky časti (stavy) automatu G dosiahnuteľné z iniciálneho stavu.
- $CoAc(G)$ značí všetky časti (stavy) automatu G, z ktorých je dosiahnuteľný koncový stav.
- Pre dva automaty G_1 a G_2, $G_1 \times G_2$ a $G_1 \| G_2$ značí ich produkt, resp. paralelnú kompozíciu.
- $Obs(G, \Sigma_o)$ je zdeterminizovaný automat G, na ktorý je aplikovaná projekcia P_o.

2.2 Pozorovateľnosť

Definícia pozorovateľnosti a algoritmus na jej verifikáciu nie sú dielom tejto práce a sú prebrané z citovanej literatúry [1].

Pozorovateľnosť intuitívne znamená:

Ak nemožno rozlíšiť medzi dvoma slovami, potom by tieto slová mali vyžadovať riadenie rovnakou akciou.

Definícia 2.2.1. Nech K a $M = \overline{M}$ sú jazyky nad abecedou Σ. Nech Σ_c je určená podmnožina Σ kontrolovateľných udalostí. Nech Σ_o je určená podmnožina Σ pozorovateľných udalostí a P_o projekcia zo Σ^* do Σ_o^*.

Jazyk K je potom pozorovateľný vzhľadom na M, Σ_o a Σ_c, ak pre všetky $s \in \overline{K}$ a pre všetky $a \in \Sigma_c$ platí:

$$(sa \notin \overline{K}) \wedge (sa \in M) \Rightarrow P^{-1}[P(s)]\{a\} \cap \overline{K} = \emptyset.$$

Poznámky

Výraz $P^{-1}[P(s)]\{a\} \cap \overline{K}$ na pravej strane implikácie definície 2.2.1 identifikuje množinu všetkých slov v $\overline{K}$, ktoré majú rovnakú projekciu ako s a po zreťazení s udalosťou a stále patria do jazyka $\overline{K}$. Ak je táto množina neprázdna, t. j. jazyk K nie je pozorovateľný, znamená to, že $\overline{K}$ obsahuje slová s a s' také, že $P_o(s) = P_o(s')$, kde $(sa \notin \overline{K})$ a $(s'a \in \overline{K})$.

Ak je parameter Σ_c vynechaný z definície pozorovateľnosti, potom je ekvivalentný Σ.

Vlastnosť pozorovateľnosti jazyka je závislá iba na prefixovom uzávere tohto jazyka. Takže K je pozorovateľný práve vtedy, keď $\overline{K}$ je pozorovateľný.

Príklad 2.2.1

Na obr. 2.1a je ukázaný automat H_1 s $L_m(H_1) = K_1$ a na obr 2.1b automat G s $L(G) = M$. Teda $K_1 = \{bu\}$ a $M = \overline{\{ub,\, bu\}}$ sú jazyky nad abecedou $\Sigma = \{u,\, b\}$. Ďalej nech je určená množina nepozorovateľných udalostí $\Sigma_{uo} = \{u\}$ a množina nekontrolovateľných udalostí $\Sigma_{uc} = \{b\}$. Je jazyk K_1 pozorovateľný vzhľadom na M, Σ_o a Σ_c?

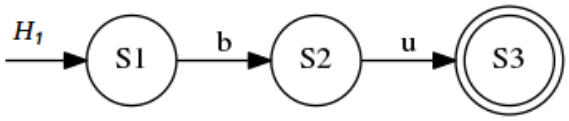
Podľa definície pozorovateľnosti treba skontrolovať všetky slová z jazyka $\overline{K_1}$. Prvým takým slovom je $s = \epsilon$. Toto slovo možno zreťaziť s kontrolovateľnou udalosťou $a = u$ tak, že $sa \notin \overline{K_1}$ a $a \in M$. Avšak v K_1 nie je ďalšie také slovo, ktoré začína slovom s rovnakou naturálnou projekciou ako ϵ a končí udalosťou u. To znamená, že pre toto slovo s a udalosť a platí, že $P^{-1}[P(s)]\{a\} \cap \overline{K} = \emptyset$, čo je v súlade s vlastnosťou pozorovateľnosti.

Ďalším slovom je $s = b$. Pre toto slovo neexisutuje kontrolovateľná udalosť taká, že po jej prireťazení k tomuto slovu by platilo: $sa \in M$ a $sa \notin \overline{K_1}$, takže podmienka pozorovateľnosti je nutne splnená. Toto isté platí pre $s = bu$ a preto je jazyk K_1 pozorovateľný.

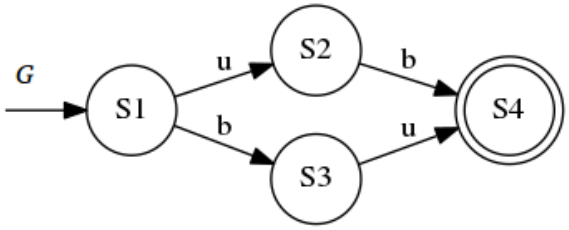
Príklad 2.2.2

Na obr 2.1c je ukázaný automat H_2 s $L_m(H_2) = K_2$ a na obr 2.1b automat G s $L(G) = M$. Teda $K_2 = \{ub\}$ a $M = \overline{\{ub,\, bu\}}$ sú jazyky nad abecedou $\Sigma = \{u, b\}$. Ďalej nech je určená množina pozorovateľných udalostí $\Sigma_{uo} = \{u\}$ a množina kontrolovateľných udalostí $\Sigma_c = \{b\}$. Je jazyk K_2 pozorovateľný vzhľadom na M, Σ_o a Σ_c?

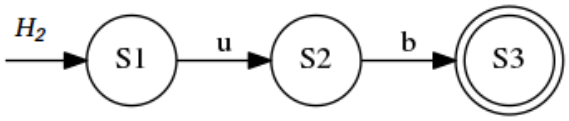
V tomto prípade pre slovo $s = \epsilon$ a udalosť $a = b \in \Sigma_c$, platí: $sa = b \in M \setminus \overline{K_2}$. Ďalej platí, že $s' = u \in P^{-1}[P(s)]$, keďže $u \in \Sigma_{uo}$ a $s'a = ub \in \overline{K_2}$, čo porušuje definíciu pozorovateľnosti. Jazyk K_2 teda nie je pozorovateľný vzhľadom na M, Σ_o a Σ_c.



(a) Automat H_1



(b) Automat G



(c) Automat H_2

Obr. 2.1: Grafy automatov z príkladov 2.2.1 a 2.2.2

2.2.1 Verifikácia pozorovateľnosti

Verifikácia pozorovateľnosti kontrolou každého jedného slova z jazyka $\overline{K}$ sa javí ako ťažkopádna a vo väčšine prípadoch ani nie je možná, keďže jazyky sú zvyčajne nekonečné množiny. Existuje algoritmus pre verifikáciu pozorovateľnosti s polynomiálnou zložitosťou, ktorý využíva konštrukciu konečného automatu. Pre tento test nie je dôležitá množina koncových stavov, keďže pozorovateľnosť je vlastnosť prefixového uzáveru jazyka, ale vo vstupných parametroch sú uvedené pre dodržanie definície deterministického konečného automatu.

Vstupné argumenty testu:

- $G := (Q_G, \Sigma, \delta_G, q_{0_G}, Q_{m_G})$, s $L(G) = M$;
- $H := (Q_H, \Sigma, \delta_H, q_{0_H}, Q_{m_H})$, s $L(H) = \overline{K}$, kde $K \subseteq M$;
- $\Sigma_o \subseteq \Sigma$;
- $\Sigma_c \subseteq \Sigma$.

Test je založený na konštrukcii deterministického automatu, ktorý zachytí všetky porušenia podmienky pozorovateľnosti. Myšlienka tohto testu je taká, že skonštruovaný automat sleduje páry slov (s_1, s_2) z $L(H)$, ktoré majú identické projekcie a zároveň sleduje dosiahnutý stav v automate G pod slovom s_2. Inak povedané, sleduje páry slov s identickou projekciou, zatiaľ čo jedno z nich je rozšíriteľné udalosťou a v $\overline{K}$ a druhé nie (a je rozšíriteľné v M udalosťou a), čo porušuje definíciu 2.2.1.

Výsledkom prechodu je trojica stavov (q_1, q_2, q_3). Rovnako prechody medzi trojicami stavov sú značené ako trojice udalostí. Nech automat nesie meno $ObsTest(H, G)$. Pre značenie prechodov medzi trojicami stavov je potrebné prázdne slovo, nech teda $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$[1].

Výstup testu:

$$ObsTest(H, G) := Ac((Q_H \times Q_H \times Q_G) \cup \{dead\}, \Sigma_\epsilon \times \Sigma_\epsilon \times \Sigma_\epsilon, \delta_{test}, (q_{0_H}, q_{0_H}, q_{0_G}), dead)$$

$ObsTest(H, G)$ je deterministický automat s prechodovou funkciou zahŕňajúcou trojice udalostí. V definícii prechodovej funkcie je použitá podmienka terminácie (VO) definovaná pre trojicu stavov (q_1, q_2, q_3) a pre kontrolovateľnú udalosť a ako:

$$(VO) \Leftrightarrow (a \in \Sigma_c) \wedge (\delta_H(q_1, a))! \wedge (\delta_H(q_2, a))\, \not! \wedge (\delta_G(q_3, a))!$$

Prechodová funkcia

$$\delta_{test} : [(Q_H \times Q_H \times Q_G) \cup \{dead\}] \times [\Sigma_\epsilon \times \Sigma_\epsilon \times \Sigma_\epsilon] \rightarrow [(Q_H \times Q_H \times Q_G) \cup \{dead\}]$$

je potom definovaná nasledovne:

- Prechod do trojice stavov je zahrnutý kedykoľvek, keď na základe čiastkových stavov sú definované $\delta_H(q_1, a), \delta_H(q_2, a)$ a $\delta_G(q_3, a)$.
- Prechod do stavu *dead* je zahrnutý práve vtedy, keď je terminačná podmienka (VO) pravdivá v aktuálnom stave (q_1, q_2, q_3) a pre kontrolovateľnú udalosť a.

1. V grafoch automatov v tejto práci a vo výsledných grafoch automatov implementovaných funkcií je prázdne slovo značené znakom podčiarkovníka (_).

1. Pre $a \in \Sigma_o$:

- $\delta_{test}((q_1, q_2, q_3), (a, a, a)) = (\delta_H(q_1, a), \delta_H(q_2, a), \delta_G(q_3, a))$;
- $\delta_{test}((q_1, q_2, q_3), (a, \epsilon, a)) = dead$.

2. Pre $a \in \Sigma_{uo}$:

- $\delta_{test}((q_1, q_2, q_3), (a, \epsilon, \epsilon)) = (\delta_H(q_1, a), q_2, q_3)$;
- $\delta_{test}((q_1, q_2, q_3), (\epsilon, a, a)) = (q_1, \delta_H(q_2, a), \delta_G(q_3, a))$;
- $\delta_{test}((q_1, q_2, q_3), (a, \epsilon, a)) = dead$.

K je teda podľa definície 2.2.1 pozorovateľný vzhľadom na M, Σ_o, Σ_c práve vtedy keď, $L_m(ObsTest(H, G)) = \emptyset$. Vidno, že pre zvýšenie efektivity algoritmu možno zastaviť konštrukciu $ObsTest(H, G)$, kedykoľvek je pravdivá terminačná podmienka (VO) v jeho dosiahnuteľnom stave s výsledkom, že jazyk K nie je pozorovateľný. Pri konštrukcii celého automatu $ObsTest(H, G)$ sú všetky porušenia podmienky pozorovateľnosti v $\overline{K}$ zachytené slovami, ktoré dosiahnu stav *dead*.

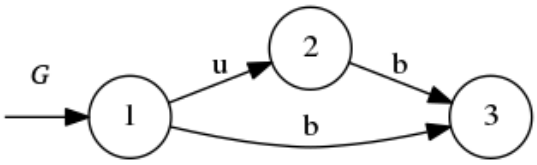
Príklad 2.2.3

Nech G na obr 2.2a je automat s $L(G) = \{u, b, ub\}$ a H_1 na obr 2.2b automat s $L(H_1) = \{b\}$. Ďalej nech sú určené množiny $\Sigma_{uc} = \{u\}$ a $\Sigma_{uo} = \{u\}$. Z obr 2.2c vidno, že graf $ObsTest(H_1, G)$ neobsahuje prechod do stavu *dead* a teda jazyk $L(H_1)$ je pozorovateľný. Je vhodné si všimnúť, že v grafe nie je prechod zo stavu (A/A/1) do stavu *dead*, pretože udalosť u nepatrí medzi kontrolovateľné.

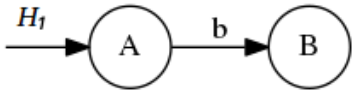
Príklad 2.2.4

Nech G na obr 2.2a je automat s $L(G) = \{u, b, ub\}$ a H_2 na obr 2.2d automat s $L(H_2) = \{u, ub\}$. Ďalej nech sú určené množiny $\Sigma_{uo} = \{u\}$ a $\Sigma_{uc} = \emptyset$. Graf $ObsTest(H_2, G)$ na obr 2.2e obsahuje prechody do stavu *dead* a teda jazyk $L(H_2) = \{u, ub\}$ nie je pozorovateľný. Podmienku pozorovateľnosti porušujú slová $s_1 = u$, $s_2 = s_3 = \epsilon$ a kontrolovateľná udalosť b. Ak udalosť b prireťazíme k s_1, toto slovo stále patrí do jazyka $L(H_2)$, ale ak ju prireťazíme k s_2, toto slovo patrí

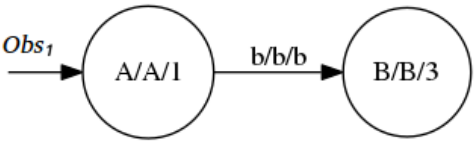
do $L(G)$, ale nepatrí do $L(H_2)$. Toto porušenie je zachytené prechodom zo stavu (D/C/1) do stavu *dead* v automate $ObsTest(H_2, G)$.



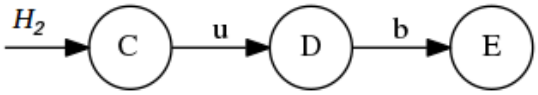
(a) Automat G



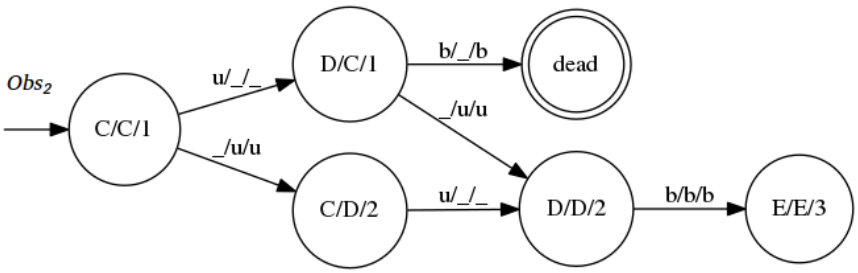
(b) Automat H_1



(c) Automat $ObsTest(H_1, G)$



(d) Automat H_2



(e) Automat $ObsTest(H_2, G)$

Obr. 2.2: Grafy automatov z príkladov 2.2.3 a 2.2.4

Poznámka

Nech n je počet prechodov z iniciálneho stavu skonštruovaného automatu $ObsTest(H, G)$ do stavu *dead*. Nech sú tieto prechody značené

trojicami udalostí (a_1, a_2, a_3). Výstupné grafy algoritmu možno potom interpretovať nasledovne:

- postupnosť udalostí $a_{1_1} a_{1_2} \dots a_{1_{n-1}}$ reprezentuje slovo $s_1 \in \overline{K}$;
- postupnosť udalostí $a_{2_1} a_{2_2} \dots a_{2_{n-1}}$ reprezentuje slovo $s_2 \in \overline{K}$;
- postupnosť udalostí $a_{3_1} a_{3_2} \dots a_{3_{n-1}}$ reprezentuje slovo $s_3 \in M$;
- $s_2 = s_3$;
- $P_o(s_1) = P_o(s_2)$.

Podmienku pozorovateľnosti potom porušuje kontrolovateľná udalosť $a = a_{1_n} = a_{3_n}$ spolu so slovom s_2, pretože $s_2 a \notin \overline{K}$ a zároveň $s_2 a \in M$, zatiaľ čo existuje slovo s_1 a tak $(P^{-1}[P(s_2)]\{a\} \cap \overline{K} \neq \emptyset)$.

2.3 Relatívna pozorovateľnosť

Ďalšie dva implementované algoritmy súvisia s vlastnosťou relatívnej pozorovateľnosti. Táto kapitola postupne predstavuje definíciu tejto vlastnosti, algoritmus pre jej verifikáciu a tiež algoritmus pre výpočet supremálneho relatívne pozorovateľného jazyka. Definícia relatívnej pozorovateľnosti [2] a ani predstavené algoritmy [4] nie sú dielom tejto práce a sú prebrané z citovanej literatúry.

Definícia 2.3.1. Nech Σ je množina udalostí a Σ_o jej určená podmnožina pozorovateľných udalostí s projekciou $P_o : \Sigma^* \to \Sigma_o^*$. Ďalej jazyk $C \subseteq L_m(G)$ a $K \subseteq C$.

Potom jazyk K je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o ($\overline{C}$-pozorovateľný), ak pre všetky slová s, $s' \in \Sigma^*$ také, že $P_o(s) = P_o(s')$, a pre všetky udalosti $a \in \Sigma$ platí, že:

$$(sa \in \overline{K}) \wedge (s' \in \overline{C}) \wedge (s'a \in L(G)) \Rightarrow s'a \in \overline{K}$$

Ak $C = K$, tak sa jedná o pozmenenú, ale ekvivalentnú definíciu pozorovateľnosti (definícia 2.2.1). Ďalej je dôležité poznamenať, že v algoritme pre verifikáciu relatívnej pozorovateľnosti predstaveného v časti 2.3.1 sa využíva poznatok, že K je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o práve vtedy, keď je K relatívne pozorovateľný vzhľadom na $\overline{C_s} = (\overline{K}\Sigma_{uo}^* \cap \overline{C})$, G a P_o, kde $C_s \subseteq C$.

Definícia 2.3.2. Relatívnu pozorovateľnosť možno ekvivalentne zadefinovať s použitím zredukovaného jazyka C_s nasledovne. Jazyk K je $\overline{C}$-pozorovateľný, ak:

$$(\forall(s, a) \in \overline{K} \times \Sigma)\, sa \in \overline{K} \Rightarrow (\nexists s' \in \overline{C_s}) \\ [(s'a \in L(G) \setminus \overline{K}) \wedge (P_o(s) = P_o(s'))]$$

2.3.1 Verifikácia relatívnej pozorovateľnosti

Podľa definície 2.3.2 možno vidieť, že $\overline{C}$-pozorovateľnosť je porušená vtedy, keď existuje $sa \in \overline{K}$ a $s'a \in \overline{C_s}\Sigma \cap \overline{K}^C \cap L(G)$ také, že $P_o(s) = P_o(s')$.[2] Toto pozorovanie naznačuje algoritmus pre verifikáciu pozorovateľnosti založenú na porovnaní projekcií jazykov $\overline{K}$ a $\overline{C_s}\Sigma \cap \overline{K}^C \cap L(G)$. Tento alogritmus má polynomiálnu zložitosť.

Pre tento algoritmus je nutné ešte rekurzívne zadefinovať funkciu premenovania $R : \Sigma^* \to \Sigma_R^*$, kde:

1. $R(a) := a$, ak $a \in \Sigma_o$;
2. $R(a) := a\text{-}r$, ak $a \notin \Sigma_o$;
3. $R(sa) := R(s)R(a)$ pre $s \in \Sigma^*$ a $a \in \Sigma$.

Vstup testu:

- $G = (Q_G, \Sigma, \delta_G, q_{0_G}, Q_{m_G})$: deterministický automat s akceptovaným jazykom $L_m(G)$;
- $A = (Q_A, \Sigma, \delta_A, q_{0_A}, Q_{m_A})$: deterministický neblokujúci automat s akceptovaným jazykom $L_m(A) = C$;
- $H = (Q_H, \Sigma, \delta_H, q_{0_h}, Q_{m_H})$: deterministický neblokujúci automat s akceptovaným jazykom $L_m(H) = K$;
- $\Sigma_o \subseteq \Sigma$.

Výstup testu:

K je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o: áno/nie.

2. $\overline{K}^C$ značí komplement jazyka $\overline{K}$ vzhľadom na Σ^*.

Postup testu:

1. Skonštruovanie automatu G_m označením všetkých jeho stavov za koncové, t. j. $G_m := (Q_G, \Sigma, \delta_G, q_{0_G}, Q_{m_G})$.

2. Skonštruovanie automatu $M := (Q_A \cup \{q_d\}, \Sigma, \delta_M, q_{0_A}, Q_A \cup \{q_d\})$ z automatu A, kde:

   (a) $\forall(q, a) \in Q_A \times \Sigma : \delta_M(q, a) = \delta_A(q, a)$, ak $\delta_A(q, a)!$, v opačnom prípade $\delta_M(q, a) = q_d$;

   (b) $\delta_M(q_d, a)$ nie je definované $\forall a \in \Sigma$.

3. Skonštruovanie automatu $M_g := M \times G_m$.

4. Skonštruovanie automatu N z automatu H,
   $N := (Q_H \cup \{q_{d1}, q_{d2}\}, \Sigma, \delta_N, q_{0_H}, \{q_{d1}, q_{d2}\})$, kde:

   (a) $\forall(q, a) \in Q_H \times \Sigma$:
   - $\delta_N(q, a) = \delta_H(q, a)$, ak $\delta_H(q, a)!$,
   - $\delta_N(q, a) = q_{d1}$, ak $((a \in \Sigma_{uo}) \wedge (\delta_h(q, a)\not!))$,
   - $\delta_n(q, a) = q_{d2}$, ak $((a \in \Sigma_o) \wedge (\delta_h(q, a)\not!))$;

   (b) $\delta_N(q_{d1}, a) = q_{d1}$, ak $a \in \Sigma_{uo}$ a $\delta_N(q_{d1}, a) = q_{d2}$, ak $a \in \Sigma_o$;

   (c) $\delta_N(q_{d2}, a)$ je nedefinované $\forall a \in \Sigma$.

5. Skonštruovanie automatu $H_c := CoAc(M_g \times N)$.

6. Skonštruovanie automatu $H_m^R := (Q_H, R(\Sigma), \delta_R, q_{0_H}, Q_H)$, kde $\delta_R(q, R(a)) = delta_h(q, a)$.

7. Skonštruovanie overovacieho automatu V.
   $V := H_m^R \| H_c = (Q_V, \Sigma \cup \Sigma_R, \delta_V, q_{0_V}, Q_{m_V})$.

8. Pre všetky $(q, a) \in Q_V \times \Sigma$ také, že $\delta_V(q, a) \in Q_{m_V}$, overiť či platí jedna z nasledujúcich podmienok:

   (a) $a \in \Sigma_o$;

   (b) $(a \notin \Sigma_o) \wedge \delta_V(q, R(a))!$.

Ak vo verifikačnom automate V existuje prechod $\delta_V(q, a)$, pre ktorý platí podmienka (a) alebo (b), potom K nie je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o. V opačnom prípade je K $\overline{C}$-pozorovateľný.

V 1. kroku algoritmu vznikne automat G_m z automatu G tak, že $L_m(G_m) = L(G)$. V 2. kroku sa skonštruuje automat M z automatu A, ktorého akceptovaný jazyk je $L_m(M) = \overline{C}\Sigma \cup \{\epsilon\}$. V nasledujúcom kroku vznikne automat $M_g = M \times G_m$, ktorý akceptuje jazyk $(\overline{C}\Sigma \cup \{\epsilon\}) \cap L(G)$. V 4. kroku vznikne automat N z automatu H, pre ktorý $L_m(N) = (\overline{K}\Sigma_{uo}^*\Sigma \cap \overline{K}^C)$. V 5. kroku sa skonštruuje automat $H_c = M_g \times N$, pre ktorý platí, že $L_m(Hc) = (\overline{K}\Sigma_{uo}^* \cap \overline{C})\Sigma \cap \overline{K}^C \cap L(G)$. V 6. kroku vznikne automat H_m^r z automatu H aplikovaním funkcie premenovania R na jeho množinu udalostí a označením všetkých jeho stavov za koncové. Posledné dva kroky zahŕňajú konštrukciu verifikačného automatu $V = H_m^r \| H_c$ a kontrolujú, či je jazyk K relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o.

2.3.2 Výpočet supremálneho relatívne pozorovateľného jazyka

Supremálny relatívne pozorovateľný jazyk z jazyka K vzhľadom na $\overline{C}$, G a P_o je najväčšia možná podmnožina jazyka K, ktorá splňuje podmienku relatívnej pozorovateľnosti vzhľadom na $\overline{C}$, G a P_o.

Algoritmus pre výpočet tohto jazyka je založený na myšlienke odstránenia všetkých prechodov z automatu (aceptujúceho jazyk K) korešpondujúcich s prechodmi vo verifikačnom automate V, ktoré porušujú podmienku relatívnej pozorovateľnosti. Tento algoritmus má exponencálnu zložitosť.

Vstup testu:

- $G = (Q_G, \Sigma, \delta_G, q_{0_G}, Q_{m_G})$: deterministický automat s akceptovaným jazykom $L_m(G)$;
- $A = (Q_A, \Sigma, \delta_A, q_{0_A}, Q_{m_A})$: deterministický neblokujúci automat s akceptovaným jazykom $L_m(A) = C$;
- $H = (Q_H, \Sigma, \delta_H, q_{0_h}, Q_{m_H})$: deterministický neblokujúci automat s akceptovaným jazykom $L_m(H) = K$;
- $\Sigma_o \subseteq \Sigma$.

Výstup testu:

H_{sup}: neblokujúci automat, ktorého akceptovaný jazyk je supremálny relatívne pozorovateľný jazyk z jazyka K vzhľadom na $\overline{C}$, G a P_o.

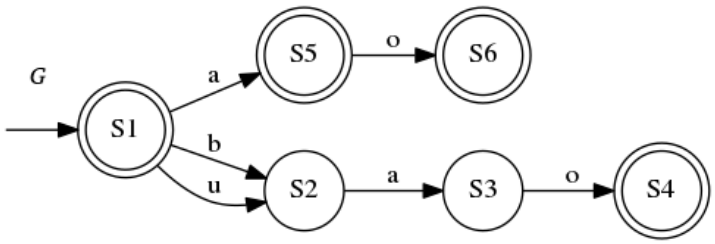
Postup testu:

1. Konštrukcia automatu $H_{sp} := H \| Obs(H, \Sigma_o) = (Q_{sp}, \Sigma, \delta_{sp}, q_{0_{sp}}, Q_{m_{sp}})$.
2. $H_s := H_{sp}$.
3. Konštrukcia verifikačného automatu V použitím algoritmu pre verifikáciu relatívnej pozorovateľnosti so vstupmi G, A a H_s, kde $V = (Q_V, \Sigma \cup \Sigma_R, \delta_V, q_{0_V}, Q_{m_V})$.
4. Ak V nie je prázdny automat, skonštruovanie množiny: $Q\Sigma = \{(q_V, a) \in Q_V \times \Sigma : (\delta_V(q_V, a) \in Q_{m_V}) \wedge ((a \in \Sigma_o) \vee ((a \notin \Sigma_o) \wedge (\delta_v(q_v, R(a))!)))\}$.
5. Ak $Q\Sigma \neq \emptyset$, potom:
   (a) $\forall (q_V, q) \in Q\Sigma$, odobrať prechod $(q, a, \delta_s(q, a))$ z automatu H_s, kde q je stav H_s rovný prvému komponentu q_V;
   (b) $H_s \leftarrow Trim(H_s)$;
   (c) vrátiť sa ku kroku 3.
6. $H_{sup} \leftarrow H_s$.

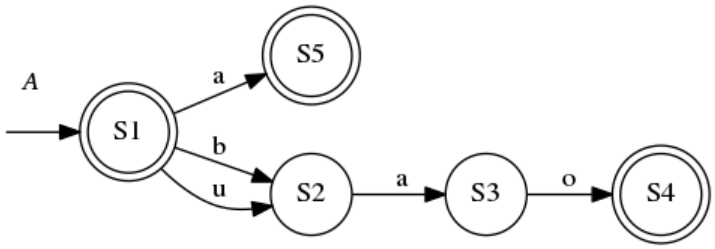
V 1. kroku algoritmu sa skonštruuje automat s rozdelením stavov $H_{sp} := H \| Obs(H, \Sigma_o)$, ktorý umožní z jazyka K odstrániť len tie slová, ktoré porušujú podmienku relatívnej pozorovateľnosti. Ďalej kroky 3 až 5 sú vykonávané iteratívne, kde v 3. kroku je skonštruovaný verifikačný automat V a v 4. kroku množina $Q\Sigma$, ktorá reprezentuje všetky páry stavov a udalostí, zodpovedné za porušenie podmienky relatívnej pozorovateľnosti. V kroku 5 sa potom odstránia zodpovedajúce prechody z automatu H_s, odstránia nedosiahnuteľné stavy a algoritmus sa vráti ku kroku 3.

Príklad 2.3.1

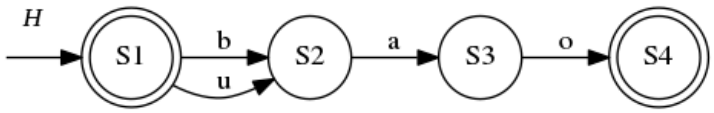
Na obr 2.3 sú ukázané vstupné automaty G (akceptuje $L_m(G)$, A (akcteptuje C) a H (akceptuje K) algoritmu pre verifikáciu relatívnej pozorovateľnosti predstaveného v časti 2.3.1. Ďalej $\Sigma = \{a, b, o, u\}$ a $\Sigma_o = \{a, b, o\}$. V 7. kroku algoritmu vznikne verifikačný automat V ukázaný na obr 2.4. Z grafu tohto automatu vidno, že prechod $((S3|S1), a, (S4|S2))$ vyhovuje podmienke (a) v 8. kroku algoritmu predstaveného v časti 2.3.1, čo znamená, že jazyk K nie je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o. Automat V generuje slová $s = u \in \overline{K}$, $s' = \epsilon \in \overline{C}$. Potom $sa = ua \in \overline{K}$, $s'a = a \in L(G) \setminus \overline{K}$ a $P_o(s) = P_o(s')$, čo porušuje definíciu 2.3.1.



(a) Automat G

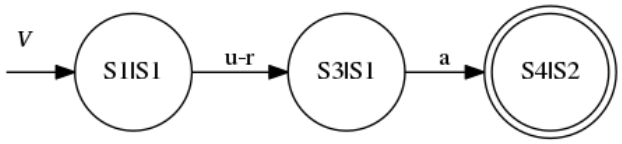


(b) Automat A



(c) Automat H

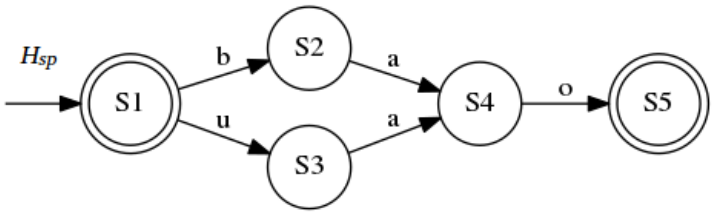
Obr. 2.3: Grafy vstupných automatov z príkladu 2.3.1



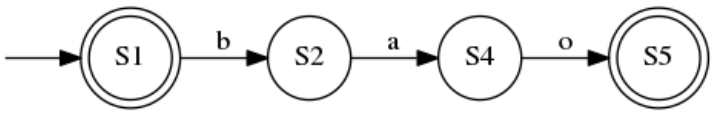
Obr. 2.4: Graf verifikačného automatu z príkladu 2.3.1

Pre skonštruovanie supremálneho relatívne pozorovateľného jazyka z jazyka K sa ponúka možnosť odstrániť prechod $(S2, a, S3)$ z automatu H na obr 2.3b. Týmto však okrem slova ua odstránime z jazyka K aj slovo ba, ktoré neporušuje definíciu 2.3.1 a nemôže byť odstránené. Preto je pri výpočte tohto jazyka použitý automat s rozdelením stavov $H_{sp} = H||Obs(H, \Sigma_o)$ ukázaný na obr 2.5, z ktorého možno odstrániť slovo ua a zachovať slovo ba.

Tento automat a verifikačný automat na obr 2.4 vzniknú v prvej iterácii algoritmu predstaveného v časti 2.3.2. V 4. kroku tohto algoritmu vznikne množina $Q\Sigma = \{((S3|S1), a)\} \neq \emptyset$. To znamená že v krokoch 5(a) až 5(c) musí byť odstránený prechod $(S3, a, S4)$ z automatu H_s ($H_s = H_{sp}$), pretože skrz tento prechod je generované slovo ua, ktoré porušuje $\overline{C}$-pozorovateľnosť. V druhej iterácii je množina $Q\Sigma$ prázdna a automat $H_{sup} = H_s$ na obr 2.6 teda akceptuje supremálny $\overline{K}$-pozorovateľný jazyk z jazyka K vzhľadom na G a P_o.



Obr. 2.5: Graf automatu H_{sp} z príkladu 2.3.1



Obr. 2.6: Graf automatu H_s z príkladu 2.3.1

3 Zásuvný modul Relative Observability

Úlohou praktickej časti tejto práce je implementovať zásuvný modul do knižnice libFAUDES. Tento modul nesie názov **Relative Observability** a implementuje niektoré algoritmy z oblasti supervízneho riadenia popísané v kapitole 2. Modul je vytvorený a pridaný podľa postupu ukázaného v časti 1.4.2 a obsahuje povinné aj dobrovoľné časti (väzby Lua, HTML dokumentáciu). V tejto kapitole sú predstavené funkcie, o ktoré modul rozširuje libFAUDES.

3.1 IsObservable

Funkcia **isObservable** testuje jazyk na pozorovateľnosť (definícia 2.2.1) a implementuje algoritmus predstavený v časti 2.2.1. Funkcia teda testuje, či je jazyk K pozorovateľný vzhľadom na M, Σ_o a Σ_c.

3.1.1 Hlavičky

- bool isObservable(Generator H, Generator G, const EventSet &obsEvents, const EventSet &conEvents)
- bool isObservable(Generator H, Generator G, Generator &obsTest, const EventSet &obsEvents, const EventSet &conEvents)

3.1.2 Parametre

- H je vstupný automat s generovaným jazykom $L(H) = \overline{K}$, kde $K \subseteq M$;
- G je vstupný automat s generovaným jazykom $L(G) = M$;
- *obsTest* je výstupný deterministický automat, ktorý zachytáva všetky porušenia definície pozorovateľnosti;
- *obsEvents* je vstupná množina pozorovateľných udalostí $\Sigma_o \in \Sigma$;
- *conEvents* je vstupná množina kontrolovateľných udalostí $\Sigma_c \in \Sigma$.

3.1.3 Poznámky

Obe funkcie testujú, či sú množiny Σ_o a Σ_c podmnožinou Σ, či $K \subseteq M$, či sú vstupné automaty deterministické. Ak niektorá z týchto podmienok nie je splnená, funkcia vyvolá výnimku s príslušným popisom.

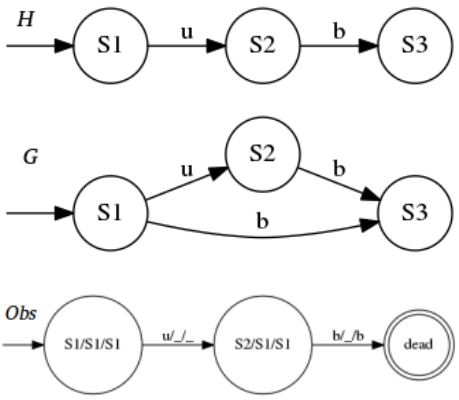
Funkcia s prvou hlavičkou zastaví hneď ako narazí na porušenie podmienky pozorovateľnosti a vráti hodnotu *false*. Ak funkcia takéto porušenie nenájde, t. j. jazyk K je pozorovateľný vzhľadom na M, Σ_o a Σ_c, vráti hodnotu *true*. Funkcia s parametrom *obsTest* navyše konštruuje deterministický konečný automat, ktorý zachytáva všetky porušenia vlastnosti pozorovateľnosti v jazyku K vzhľadom na M, Σ_o a Σ_c.

Príklad 3.1

Použitie funkcie isObservable je demonštrované na obr 3.1, kde je ukázaný zdrojový kód s volaním funkcie isObservable a automaty H, G a $ObsTest(H, G)$. V tomto prípade funkcia testuje, či je jazyk $L(H) = \{u, b\}$ pozorovateľný vzhľadom na jazyk $L(G) = \{u, b, ub\}$ a množiny udalostí $\Sigma_o = \{b\}, \Sigma_c = \{u, b\}$. Výsledkom tohto testu je, že $L(H)$ nie je pozorovateľný, čo je možné vidieť aj z grafu $ObsTest(H, G)$. Interpretácia výsledných grafov tejto funkcie je podrobne vysvetlená v podkapitole 2.2.

```cpp
#include "libfaudes.h"

using namespace faudes;

int main() {

    // read generators from files
    Generator h("data/obs_h.gen");
    Generator g("data/obs_g.gen");

    // read event sets from files
    EventSet obsEvents("data/obs_obs_events.alph");
    EventSet conEvents("data/obs_con_events.alph");

    // test if K is observable with respect to M, Eo
    // and Ec and construct generator ObsTest(H, G)
    Generator obsTest;
    if (isObservable(h, g, obsTest, obsEvents, conEvents)) {
        std::cout << "Language K is observable "
                     "w.r.t. M, Eo and Ec!" << std::endl;
    } else {
        std::cout << "Language K is not observable "
                     "w.r.t. M, Eo and Ec!" << std::endl;
    }

    // now we can further work with ObsTest(H, G)
    obsTest.GraphWrite("obs_obstest_hg");

    return 0;
}
```

(a) C++ kód



(b) Automaty H, G a $obsTest(H, G)$

Obr. 3.1: Ukážka použitia funkcie isObservable

3.2 IsRelativelyObservable

Funkcia **isRelativelyObservable** testuje jazyk na relatívnu pozorovateľnosť (definícia 2.3.1) a implementuje algoritmus predstavený v časti 2.3.1. Funkcia teda testuje, či je jazyk K relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o.

3.2.1 Hlavička

- bool isRelativelyObservable(const Generator &G, const Generator &A, const Generator &H, const EventSet &obsEvents)

3.2.2 Parametre

- G je vstupný automat s akceptovaným jazykom $L_m(G)$;
- A je vstupný neblokujúci automat s akceptovaným jazykom $L_m(A) = C$;
- H je vstupný neblokujúci automat s akceptovaným jazykom $L_m(H) = K$;
- *obsEvents* je vstupná množina pozorovateľných udalostí $\Sigma_o \in \Sigma$.

3.2.3 Poznámky

Funkcia testuje, či $\Sigma_o \in \Sigma$, či sú vstupné automaty deterministické a z nich A a H neblokujúce. Ďalej funkcia testuje podmienku: $L_m(H) \subseteq L_m(A) \subseteq L_m(G)$. Ak niektorá z týchto podmienok nie je splnená, funkcia vyvolá výnimku s príslušným popisom.

Funkcia skonštruuje verifikačný automat, ktorý zachytáva dvojice stavov a udalostí, ktoré porušujú vlastnosť relatívnej pozorovateľnosti. Ak tento automat žiadne také dvojice nezachytí, funkcia vráti hodnotu *true*, t. j. jazyk K je relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o. Inak vráti hodnotu *false*.

3.3 SupRelativelyObservable

Funkcia **supRelativelyObservable** implementuje algoritmus výpočtu supremálneho relatívne pozorovateľného jazyka, ktorý bol predstavený v časti 2.3.2. Funkcia teda odstráni časti z jazyka K, ktoré sú zodpovedné za porušenia vlastnosti relatívnej pozorovateľnosti vzhľadom na $\overline{C}$, G a P_o a vypočíta jazyk relatívne pozorovateľný vzhľadom na tieto parametre.

3.3.1 Hlavička

- void supRelativelyObservable(const Generator &G, const Generator &A, const Generator &H, Generator &result, const EventSet &obsEvents)

3.3.2 Parametre

- G je vstupný automat s akceptovaným jazykom $L_m(G)$;
- A je vstupný neblokujúci automat s akceptovaným jazykom $L_m(A) = C$;
- H je vstupný neblokujúci automat s akceptovaným jazykom $L_m(H) = K$;
- *result* je výstupný automat, ktorý akceptuje supremálny relatívne pozorovateľný jazyk z jazyka K vzhľadom na $\overline{C}$, G a P_o;
- *obsEvents* je vstupná množina pozorovateľných udalostí $\Sigma_o \in \Sigma$.

3.3.3 Poznámky

Funkcia testuje, či $\Sigma_o \in \Sigma$, či sú vstupné automaty deterministické a z nich A a H neblokujúce. Ďalej funkcia testuje podmienku: $L_m(H) \subseteq L_m(A) \subseteq L_m(G)$. Ak niektorá z týchto podmienok nie je splnená, funkcia vyvolá výnimku s príslušným popisom.

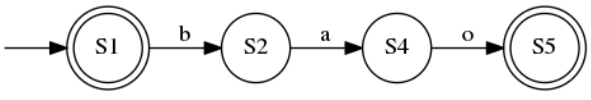
Funkcia skonštruuje verifikačný automat, ktorý zachytáva dvojice stavov a udalostí, ktoré porušujú vlastnosť relatívnej pozorovateľnosti. Potom na základe týchto dvojíc vypočíta supremálny relatívne pozorovateľný jazyk a vráti ho pomocou paramteru *result*.

Príklad 3.2

Použitie funkcií isRelativelyObservable a supRelativelyObservable je demonštrované na obr 3.2, kde je ukázaný kód s ich volaním. V tomto prípade najprv ukážka testuje, či je jazyk K relatívne pozorovateľný vzhľadom na $\overline{C}$, G a P_o. Automaty G, A a H na obr 2.3 postupne reprezentujú jazyky $L_m(G)$, C a K. Ďalej sú určené množiny $\Sigma = \{a, b, o, u\}$ a $\Sigma_o = \{a, b, o\}$. Výsledok testu je negatívny. V ukážke je potom volaná funkcia supRelativelyObservable s rovnakými vstupmi, ktorej výstupom je automat na obr 3.3, ktorý akceptuje supremálny relatívne pozorovateľný jazyk z jazyka K vzhľadom na $\overline{C}$, G a P_o.

```
#include "libfaudes.h"

using namespace faudes;

int main() {

    // read generators from files
    Generator g("data/rel_obs_g.gen");
    Generator a("data/rel_obs_a.gen");
    Generator h("data/rel_obs_h.gen");

    // read the set of observable events from file
    EventSet obsEvents("data/rel_obs_obs_events.alph");

    // test if Lm(H) is realtively observable with
    // respect to prefix-closure(Lm(A)), G and Po.
    if (isRelativelyObservable(g, a, h, obsEvents)) {
        std::cout << "Language K is relatively observable"
                     " w.r.t. prefix-closure(C),G and Po!"
                  << std::endl;
    } else {
        std::cout << "Language K is not relatively observable"
                     " w.r.t. prefix-closure(C), G and Po!"
                  << std::endl;
    }

    // compute the supremal relatively observable sublanguage
    // of Lm(H) w.r.t. prefix-closure(C), G and Po;
    // funtcion returns it via output generator parameter

    Generator supRelObsLan;
    supRelativelyObservable(g, a, h, supRelObsLan, obsEvents);

    // now we can further work with generator
    // representing the supremal language
    supRelObsLan.Write();

    return 0;
}
```

Obr. 3.2: Ukážka použitia funkcií isRelativelyObservable a supRelativelyObservable



Obr. 3.3: Graf výsledného automatu funkcie supRelativelyObservable

Záver

Zadaním práce bolo zoznámiť čitateľa s knižnicou libFAUDES a poskytnúť mu základné informácie, potrebné k pochopeniu práce s ňou. Aby bolo toto možné, musel som sám najprv preskúmať základnú manipuláciu s knižnicou a jej ďalšie možnosti. Zhromaždil som informácie, ktoré uľahčujú novému používateľovi začať používať libFAUDES s jej všetkými dôležitými časťami. Práca informuje o zameraní knižnice, o jej základných objektoch, o jej inštalácií a použití. Takisto informuje aj o príbuzných aplikáciach, ktoré využívajú jej dátové typy či funkcie.

Po prečítaní kapitoly 1 by mal byť čitateľ z oblasti informatiky schopný nielen rozumieť prostrediu knižnice a vedieť aktívne s ňou pracovať, ale aj rozšíriť knižnicu o zásuvný modul podľa vlastnej potreby. Práca vysvetľuje postup rozšírenia knižnice o zásuvný modul a demonštruje ho na súboroch vzorového modulu, čo bolo ďalšou jej úlohou.

Práca ďalej predstavuje vlastnosti a algoritmy, ktoré boli základom zásuvného modulu, o ktorý som mal knižnicu rozšíriť. Vlastnosti sú tu definované a vysvetlené na vybraných príkladoch. Takisto sú tu vysvetlené aj algoritmy pre verifikáciu týchto vlastností a výpočet supremálneho relatívne pozorovateľného jazyka. Táto časť práce je nevyhnutná pre pochopenie implementovaných algoritmov a ich samotnej implementácie, čo umožnilo knižnicu rozšíriť o tieto prvky.

Praktickou úlohou práce bolo teda rozšíriť knižnicu o vlastný zásuvný modul a zároveň demonštrovať možnosti rozšírenia knižnice na reálnom príklade. Vyššie spomenuté algoritmy som úspešne naimplementoval a pridal ich pomocou zásuvného modulu s menom *Relative Observability* do knižnice. Ich pridanie umožňuje jednoducho testovať tieto vlastnosti v rámci libFAUDES. Pre pridanie tohto rozšírenia som sa musel zoznámiť s nástrojmi použitými v knižnici, čo bolo jednou z častí zadania. Tieto nástroje sú krátko predstavené aj v texte práce. Archív s implementovaným modulom je súčasťou elektronickej prílohy a obsahuje všetky súbory potrebné pre rozšírenie knižnice o zásuvný modul aj spolu s dobrovoľnými časťami (HTML dokumentácia, väzby Lua).

Bibliografia

1. CASSANDRAS, C. G.; LAFORTUNE, S. *Introduction to Discrete Event Systems, Second Edition*. Springer, 2008. ISBN 978-0-387-33332-8. Dostupné z DOI: 10.1007/978-0-387-68612-7.
2. CAI, K.; ZHANG, R.; WONHAM, W. M. Relative Observability of Discrete-Event Systems and Its Supremal Sublanguages. *IEEE Transactions on Automatic Control*. 2015, roč. 60, č. 3, s. 659–670. ISSN 0018-9286. Dostupné z DOI: 10.1109/TAC.2014.2341891.
3. *LibFAUDES* [online]. Erlangen, Germany: Friedrich-Alexander University Erlangen-Nürnberg, 2006–2018 [cit. 2018-11-13]. Dostupné z: http://www.rt.eei.uni-erlangen.de/FGdes/faudes/index.html.
4. ALVES, M. V. S.; CARVALHO, L. K.; BASILIO, J.C. New algorithms for verification of relative observability and computation of supremal relatively observable sublanguage. In: *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*. 2016, s. 526–531. Dostupné z DOI: 10.1109/CCA.2016.7587883.