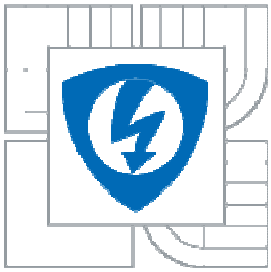
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV MATEMATIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF MATHEMATICS

TVORBA OPERAČNÍHO SYSTÉMU ZALOŽENÉHO NA EVOLUČNÍCH A GENETICKÝCH ALGORITMECH

DEVELOPMENT OF OPERATING SYSTEM BASED ON EVOLUTIONARY AND GENETIC ALGORITHMS

DIZERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE Ing. PETR SKORKOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE prof. RNDr. JAN CHVALINA, DrSc.
SUPERVISOR

BRNO 2013

Abstrakt

Hlavním cílem této práce je představit nové myšlenky, jak obvyklé postupy pro návrh operačního systému a přidruženého software mohou být vylepšeny aby se staly součástí automatizovaného vývoje software. Obecně se předpokládá, že algoritmy nalezené pomocí genetického programování nemohou být použity pro přesné výpočty, ale jen pro přibližná řešení. Je představeno několik příkladů jak lze při evoluci software přesto dosáhnout přiměřené přesnosti. Pro dosažení tohoto cíle jsou vlastnosti stromově orientovaných struktur spolu s postupy používanými u buněčných automatů spojeny do nového slibného přístupu, který slučuje výhody obou metod. Byla vyvinuta aplikace založená na těchto nových postupech a předpokládá se její budoucí využití v procesu automatizovaného vývoje software.

Klíčová slova

genetické programování, buněčný automat, automatizovaný vývoj software

Abstract

The main goal of the work is to introduce new ideas how traditional approaches for designing an operation system and associated software can be improved to be a part of automatic software evolution. It is generally supposed that algorithms found by the genetic programming processes cannot be used for exact calculations but only for approximate solutions. Several examples of software evolution are introduced, to show that quite precise solutions can be achieved. To reach this goal, characteristics of tree-like structures with approaches based on cellular automata features are combined in a new promising technique of algorithm representation, joining benefits of both concepts. An application has been developed based on these new genetic programming concepts and it is supposed it can be a part of a future automatic software evolution process.

Keywords

genetic programming, cellular automaton, automatic software development

Bibliografická citace

SKORKOVSKÝ, P. *Tvorba operačního systému založeného na evolučních a genetických algoritmech*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav matematiky, 2013. 105 s., 6 s. příloh. Dizertační práce. Vedoucí práce: prof. RNDr. Jan Chvalina, DrSc.

Prohlášení

Prohlašuji, že svou dizertační práci na téma „*Tvorba operačního systému založeného na evolučních a genetických algoritmech*“ jsem vypracoval samostatně pod vedením vedoucího dizertační práce prof. RNDr. Jana Chvaliny, DrSc. a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené dizertační práce dále prohlašuji, že v souvislosti s vytvořením této dizertační práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: …… ……

Ing. Petr Skorkovský

Poděkování

Na tomto místě bych rád poděkoval mému školiteli panu prof. RNDr. Janu Chvalinovi, DrSc. za veškerou pomoc, podporu a ochotu, která mi byla věnována po celou dobu mého doktorského studia a během přípravy této dizertační práce.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

Každodenní řešení problémů v mnoha různých odvětvích lidské činnosti se výrazně usnadnilo a zefektivnilo díky masivnímu využívání prostředků výpočetní techniky. Zejména v posledních letech dosáhly informační technologie takového rozmachu, že stále více narůstá jejich důležitost a prohlubuje se jejich význam, jak postupně nahrazují starší pracovní postupy. Dostávají se do popředí zájmu a výrazně ovlivňují veškeré pracovní i mimopracovní aktivity každého z nás. Tento trend však přináší na jedné straně úsporu rutinních pracovních postupů, které by bylo nutné jinak provádět ručně, na druhé straně ale znamenají výrazný nárůst práce pro programátory. Situace je v současné době taková, že i přes dostupnost moderních vývojářských nástrojů a přes významné zefektivnění vývoje programového vybavení, tvorba algoritmů a další příbuzné činnosti s tím spojené zatím nebyly plně zautomatizovány a tak veškeré úsilí spočívá na programátorech a vývojářích samotných. Navíc zdaleka neznamená, že již jednou hotový softwarový produkt bude sloužit ve všech ohledech a k nejvyšší spokojenosti uživatele i v budoucnu. Technické vybavení postupem času stárne, nároky uživatelů na výkonnost se zvyšují a se změnou hardware vznikají i nové verze operačních systémů, pro které je často nutné vytvářet i nové verze již existujících programů. Další údržba již hotových produktů, jejich vývoj a rozšiřování tak patří k běžnému životnímu cyklu, který je patrný u většiny softwarového vybavení. Tyto skutečnosti znamenají nepřetržitou a nikdy nekončící práci pro programátory. Je vhodné se zamyslet nad tím, zda není možné programátorům jejich práci více usnadnit. Nedá se zřejmě přímo říci, že by jejich účast na celém procesu vývoje softwaru v budoucnu nebyla vůbec nutná, ale alespoň by se výrazně omezila jen na nejnutnější úkony spojené s administrativou a byly by tak alespoň ušetřeni časově i pracovně nejnáročnějších etap.

Jako vhodný směr pro automatizaci řešení různých tříd problémů, který by mohl časem splňovat potřebná kritéria, se zdají být slibné současné studie a výzkumy na poli genetického programování. Ač zatím jejich plné užití znamená nutnost překonávat různá jejich omezení a není zatím možné je využívat v plné míře pro řešení všech známých typů problémů, byly již přesto jisté úspěchy zaznamenány a to nejen pouze v teoretické rovině, ale i v technické praxi. Pro firmy, kterým se tento úspěch podařil, zpravidla znamenal výrazný přísun a obrovské ekonomické úspory [11]. Technické řešení jistého obtížného problému, které je posléze nalezeno pomocí evolučních technik (mezi které patří i genetické programování), bývá často takové povahy, že člověk by jinou cestou k němu nejspíš ani nedospěl. Nejvíce se tedy vyplatí užívat tyto postupy na řešení vysoce náročných problémů, pro které ani neexistují jiné známé klasické postupy a výpočty. Přitom bývají s pomocí genetického programování nalezeny takové algoritmy (nebo matematické rovnice), které svou komplexností přesahují technický rozhled běžného inženýra (nebo matematika). Mnohdy není snadné zpětně zjistit a spolehlivě určit jak a proč nalezené řešení funguje. Jedním z cílů může být tedy i další automatizace převodu komplexního řešení problému na srozumitelnější a přehlednější popis, pokud je to samozřejmě v daném případě vůbec možné.

2. Cíle dizertace

V rámci dizertační práce vyvíjená aplikace je založena na aplikování již známých poznatků genetického programování, kde použitým jádrem procesu je určitý mechanismus odvozený od vlastností podobných vlastnostem buněčného automatu. Nejedná se přímo o buněčný automat, ale spíše o procesor logických funkcí, který má některé vlastnosti odvozeny od chování buněčného automatu.

Rozsahu dizertační práce odpovídá zpracování těchto bodů:

- Rešerše současného stavu zkoumané oblasti genetického programování, již dosažené výsledky v oboru, ze kterých vlastní práce vychází.
- Navrhnout vlastnosti nového operačního systému (nové generace), který by využíval pro svůj vývoj a vývoj pro něj určených aplikací techniky genetického programování.
- Navrhnout architekturu operačního systému nové generace.
- Analýza požadavků na nový (programovací) jazyk a jeho vlastností (nejspíš neprocedurálního typu), kterým lze vhodným způsobem kódovat algoritmy, aby bylo možné zautomatizovat jejich vývoj.
- Matematický popis nalezeného jazyka pro kódování algoritmů.
- Vývoj aplikace, která je schopna po zadání vhodně kódovaných definic popisujících chování hledaného algoritmu (jeho vstupy, výstupy, dynamické chování, přechodové stavy, atd.), automaticky nalézt tento algoritmus a jeho reprezentaci vyjádřenou „novým jazykem“. Nalezený algoritmus je poté možno kdykoliv a opakovaně spouštět a používat ho na řešení problémů pro které byl určen.
- Nalézt způsob jak popis získaného algoritmu převést do jiného, lidem srozumitelnějšího formátu (např. na popis logickými rovnicemi číslicové techniky), místo pouhé řady binárních čísel, které mnoho neřeknou o fungování algoritmu a jsou nepřehledné.
- Detailní popis vyvíjené aplikace, programové bloky, podprogramy, moduly, syntaxe a formát vstupních a výstupních souborů, popis datových typů, atd.
- Praktické příklady a ukázka hledaných a nalezených algoritmů s použitím vyvinuté aplikace.
- Návrhy pro navazující výzkum.

3. Rozsah a význam řešeného problému

Zde je nutné představit rozsah a dopad popisovaného výzkumu v širším kontextu a nastíníme, kam až by mohly poznatky získané v rámci dizertační práce vést při pozdějším uplatnění v praxi. Význam dosažených výsledků lze tak především ocenit při dalším rozšiřování a prohloubení již započatého výzkumu, který momentálně směřuje pouze do jistého dílčího bodu v rámci dizertační práce a skutečný cíl leží tak mnohem dál, avšak v tuto chvíli je ještě příliš vzdálen, aby se dal do dizertační práce zahrnout celý.

V současné době jsme svědky prudkého rozmachu výpočetní techniky. I trh se softwarovým vybavením je rozvinutý do té míry, že pokrývá potřeby většiny zákazníků a lze pořídit téměř jakoukoliv aplikaci zabývající se téměř libovolným zaměřením uživatelů. Jedny z kritérií, která určují úspěšnost softwarového produktu jsou: množství funkcí které poskytují, ergonomičnost ovládání a popřípadě jeho výkonnost (rychlost a množství zpracování dat, apod.). Všechny softwarové produkty mají ale jednu společnou vlastnost a tou je vývoj v režii softwarové firmy zaměstnávající mnohdy značné množství programátorů, kteří tráví spousty hodin návrhem a vývojem algoritmů dle specifikace, která je sestavena tak, aby co nejlépe naplnila potřeby zákazníka či cílové skupiny uživatelů. Většinu těchto aplikací je nutno i nadále ručně vyvíjet spolu s tím jak narůstají časem potřeby zákazníků, jak se mění hardware počítače či při přechodu na nový operační systém. Je tedy vhodné zabývat se tím, zda je možné vyvíjet software i jiným způsobem, jestli lze při vhodně navržené a propracované specifikaci a s použitím odlišných přístupů, než na které jsme zvyklí, dosáhnout stavu, kdy se z velké části software vyvíjí sám.

Většina aplikací se v současné době vyvíjí ručně s pomocí procedurálního programovacího jazyka a pochopitelně nelze počítat s tím, že by se daly pro automatizovaný vývoj software použít stejné programovací jazyky, jaké jsou dnes hojně používány při ručním vývoji, neboť pro účely automatizovaného vývoje nejsou tyto příliš vhodné. Od základů musíme změnit představu o tom co to vlastně software je, jak funguje, respektive jakých různých podob může nabývat. Rovněž důležitým parametrem je způsob, jak je software prováděn (interpretován) systémem, pro který byl napsán, a jak svým následným chováním (často dynamickým) ovlivňuje své okolí (hardware), jak zpracovává různé hodnoty na vstupech a jak dle předem připraveného předpisu (dle specifikace) daný algoritmus generuje očekávané hodnoty na výstupech. V případě, že by se podařilo vybudovat metodologii pro automatizovaný vývoj algoritmů (nebo automatizovaný vývoj software, což může být totéž), mělo by být teoreticky možné vytvořit i operační systém, který dynamicky mění svou podobu podle aktuálních potřeb svého uživatele, neustále se zdokonaluje, vyvíjí a rozšiřuje se sám o funkce, které jsou potřeba. Kdykoliv by to bylo nutné, stačilo by, aby se jen změnily parametry specifikace a už by se mohl vývoj ubírat jiným, aktuálně potřebným směrem a možná i přímo za chodu. Každý uživatel by pak disponoval operačním systémem, který by v maximální možné míře naplňoval jeho požadavky a potřeby a sám by se podle toho i aktualizoval, kdyby se požadavky časem změnily.

Dosáhnout těchto vzdálených cílů v rámci dizertační práce pochopitelně není v krátkém čase možné a proto je potřeba stanovit takový cíl, který je možné stihnout a uspokojivě vyřešit. Přesto by ale mohl tento bližší cíl znamenat dobrý základ pro návazný výzkum a přiblížení se v započatém směru ke vzdálenému cíli.

4. Operační systém pro automatizovaný vývoj aplikací

4.1 Vymezení pojmů

Operačním systémem v rámci této dizertační práce je myšlen počítačový systém založený na softwarovém vybavení, sloužícím ke snadné správě, údržbě, ovládání, spouštění a přístupu k dalšímu softwarovému vybavení.

Veškeré softwarové vybavení slouží pro ovládání periferních zařízení, zajištění interních systémových služeb operačního systému, nebo pro:

- vytváření,
- správu,
- údržbu,
- užívání,
- a distribuci

různého typu dat, informací a jiného programového vybavení v lokálních, externích, nebo vzdálených datových úložištích a médiích.

Podle způsobu integrace do počítačového systému je softwarové vybavení tohoto typu:

- Je součástí operačního systému samotného a je spouštěno lokálně,
- bylo později přidáno podle individuálních potřeb uživatele a je spouštěno lokálně,
- lze jej spouštět jako hostitel pro jiného uživatele přistupujícího ze vzdáleného místa a distribuovat výstupy do vzdáleného místa,
- lze jej spouštět na vzdáleném místě na jiném hostitelském systému a přejímat jeho výstupy do lokálního místa,
- ve chvíli použití lze softwarové vybavení stáhnout ze vzdáleného místa ale dále ho spustit lokálně
- ve chvíli použití lze softwarové vybavení poskytnout jinému uživateli pro stažení do vzdáleného místa, aby se spustilo na jiném, vzdáleném systému.

Ve chvíli, kdy se nějaké softwarové vybavení zavádí a spouští operačním systémem, obvykle se nazývá procesem. Je běžné, že v jeden okamžik běží v operačním systému více procesů najednou. Pro bezproblémový chod je nutné správně procesy

přepínat a na vyžádání jim přidělovat různé, v tu chvíli dostupné, systémové zdroje, které jsou poskytovány systémovými službami. Dříve přidělené systémové zdroje jsou ve chvíli, kdy nejsou předchozím procesem dále využívány, opět uvolněny pro budoucí užití jiným procesem.

4.2 Funkce současného operačního systému

Operační systém lze rozdělit na části zajišťující:

- správu systémových zdrojů na systémové úrovni,
- správu systémových služeb,
- správu vstupních a výstupních uživatelských rozhraní na uživatelské úrovni,
- správu grafických prvků pro vizuální komunikaci s uživatelem

Systémové zdroje, se kterými operační systém pracuje a které spravuje na systémové úrovni, bývají:

- správa dočasné systémové paměti typu RAM,
- správa trvalé paměti realizované pevným diskem nebo pamětí typu flash pro data zapsaná v lokálním souborovém systému,
- správa a přístup k datům uloženým na externích, nebo vzdálených datových úložištích a médiích
- sběrnice pro sériový přenos dat po kabelu
- sběrnice pro paralelní přenos dat po kabelu
- sběrnice pro sériový přenos dat bezdrátově přes infra port
- sběrnice pro sériový přenos dat bezdrátově přes bluetooth
- USB sběrnice a periferní zařízení k nim připojené
- interní systémové sběrnice na základní desce počítače pro komunikaci s připojenými kartami
- zjištění aktuální polohy pomocí systému GPS
- zjištění aktuální polohy pomocí detektoru zrychlení, naklonění, pádu
- zjištění reálného času
- síťová karta připojená do okolí pomocí kabelu: Zajišťuje připojení do lokální počítačové sítě LAN, do vnější počítačové sítě WAN, do Internetu.
- síťová karta připojená do okolí bezdrátově – Wifi: Zajišťuje připojení do lokální počítačové sítě LAN, do vnější počítačové sítě WAN, do Internetu.

Systémové služby, se kterými operační systém pracuje na systémové úrovni a které spravuje, bývají:

- zavádění, spouštění, provoz a pozdější bezpečné ukončení celého operačního systému
- zavádění a spouštění nových procesů, přidělování paměti a dalších systémových zdrojů, které si procesy vyžádají pro svůj chod
- bezpečné ukončování běžících procesů, uvolňování paměti a dalších systémových zdrojů, které si procesy předtím vyžádali pro svůj chod
- vytváření, správa a časování událostí pokud jejich vznik nějaký proces vyžádal nebo naplánoval
- sledování a distribuce informace o vzniklých událostech procesům, které událost očekávají
- pozastavení procesu, který čeká na událost, nebo pokračování běhu procesu pokud očekávaná událost přišla
- přepínání procesů a jejich časování pro souběžný běh procesů dle nastavených priorit a správa těchto priorit
- správa přístupu ke zdrojům aby nedocházelo ke kolizím, pokud dva procesy vyžadují stejný zdroj ve stejném čase - řízení dle priorit nebo kvót
- izolace procesů mezi sebou aby nemohli úmyslně nebo neúmyslně narušit vzájemnou integritu
- správa víceuživatelského a vzdáleného přístupu, správa hesel, biometrických údajů a uživatelských oprávnění
- sledování výkonnostních parametrů, tvorba statistiky
- diagnostika správné funkčnosti, samo-opravné mechanizmy, sledování integrity systému, zajištění bezpečnosti, ochrana proti ztrátě kontroly nad systémem kvůli poruše, nebo kvůli neautorizovanému přístupu

Vstupní uživatelská rozhraní na uživatelské úrovni, která operační systém spravuje, bývají:

- vstup z klávesnice: ovládání klávesnicí pro pohyb kurzoru nebo skrze psaný text pomocí vkládání textu na místo kde se nachází kurzor, ovládání klávesnicí bez psaní textu
- vstup z polohovacího zařízení skrze ukazatel (šipku), ruku, nebo uživatelův prst: počítačová myš, dotyková plocha, pákový ovladač
- vstup skrze posuv nebo rotací ovládacího prvku: kolečko na myši
- vstup skrze zvukové záznamové zařízení: hlasové ovládání, převod informace ze zvuku
- vstup skrze obrazové záznamové zařízení: kamera, skener, převod informace z obrazu

- jiné vstupy: autorizace uživatele pomocí otisku prstů, autorizace uživatele pomocí čipové karty, autorizace uživatele pomocí obrysů dlaně

Výstupní uživatelská rozhraní na uživatelské úrovni, která operační systém spravuje, bývají:

- výstup na obrazovku nebo na více obrazovek pomocí změny barvy pixelů v matici zobrazovacího panelu
- tiskový výstup
- zvukový výstup
- světelná signalizace pomocí LED, nebo pomocí displejových segmentů na doplňkových panelech zobrazující jiný druh informace než hlavní obrazový panel složený z pixelů: čísla nebo text složený ze segmentů - rozsvěcování nebo zhasínání jednotlivých segmentů, které tvoří informaci

Grafické prvky pro vizuální komunikaci s uživatelem, případně pro ovládání systému uživatelem, které operační systém spravuje, bývají:

- pracovní plocha
- nabídky pro spouštění zabudovaného, nebo uživatelem přidaného programového vybavení
- grafická okna,
- ikony,
- okna pro operace se soubory a adresáři,
- systémová hlášení,
- nabídky pro volbu výběrem,
- textová pole,
- tlačítka,
- posuvníky,
- zaškrtávací políčka,
- výběry z menu,
- změna barev, fontů a velikostí písma
- zobrazování nápovědy pro uživatele
- ovládací lišty,
- panely nástrojů
- zobrazení textového kurzoru, nebo šipky polohovacího zařízení
- systémová lišta pro zobrazení důležitých provozních údajů, varování uživatele, upozornění na vzniklé události, žádosti o provedení akcí ze strany uživatele
- jiné grafické ovládací prvky

4.3 Dnešní praxe realizace softwarových produktů

Operační systémy, tak jak je známe dnes, bývají realizovány s pomocí vývojových nástrojů pro vývoj softwarového vybavení. Stejně tak další softwarové vybavení, které je pro tyto operační systémy určeno a je v nich později spouštěno, bývá rovněž vyvíjeno s pomocí stejných vývojových nástrojů. Práce s vývojovými nástroji je prováděna programátory, kteří musí mít dokonalou znalost o programu, který právě vyvíjejí, aby nenastala situace, že se program později chová jinak než uživatel, nebo objednavatel programu očekává.

Nejprve budoucí uživatel, nebo objednavatel programu stanoví potřebné chování, nadefinují se typy a formáty dat, se kterými má vyvíjený program pracovat a dále se stanoví další požadavky a kritéria jako je vzhled, statické a dynamické chování algoritmů, atd. Programátor potom s pomocí programovacího jazyka napíše a odladí zdrojové kódy podle předchozích požadavků na software. Tyto zdrojové kódy slouží později pro kompilaci a sestavení požadovaného softwarového vybavení. Vytvořené softwarové vybavení má po jeho spuštění v operačním systému zajistit svůj očekávaný vzhled a chování pokud programátor neudělal během vývoje nějakou chybu. Chyby v programech jsou často nevyhnutelné, a přestože by většina z nich měla být nalezena a odstraněna již ve fázi vývoje, často tomu tak není. Později nalezené chyby se odstraňují obvykle dodatečně s pomocí celých nových verzí programů, aktualizacemi jen chybných částí programů, nebo drobnými opravami (takzvanými záplatami). Jak už bylo uvedeno výše, stejný postup platí pro jakékoliv programové vybavení, tedy i pro vývoj operačního systému samotného - tento je jen speciálním typem programového vybavení.

Pro zjednodušení můžeme uvažovat o tomto sledu kroků, které jsou nutné pro vytvoření nového programového vybavení (a i nového operačního systému), tak jak je to prováděno v současnosti:

1. Stanovení cíle, účelu a oblasti využití nového programového vybavení
2. Určení systému, na kterém programové vybavení má běžet dle hardwarové specifikace počítačového systému a dle dostupných prostředků cílového operačního systému včetně kompatibility s hostitelským prostředím
3. Pokud využití programového vybavení přesahuje hostitelský systém a očekává se spolupráce na úrovni počítačových sítí, nebo externích zařízení, je nutné specifikovat pravidla, podle kterých k interakcím s blízkým a vzdáleným okolím dochází.
4. Je třeba vyjasnit, jakou roli hraje uživatel. Jestli programové vybavení vyžaduje interakci s uživatelem (a jakou), jestli interaguje se skupinou uživatelů ve stejnou dobu, zdali dochází k výměně dat s jiným programovým vybavením, nebo jestli může pracovat zcela autonomně a nezávisle na svém okolí.
5. Zadavatel, nebo budoucí uživatel musí předložit úplnou specifikaci o požadavcích, co má softwarové vybavení umět. Během samotného vývoje se tato specifikace obvykle ještě více upřesňuje, rozšiřuje, nebo opravuje, pokud programátor zjistil nějaké nedostatky v popisu.

6. Ze specifikace vyplývají typy a formáty dat nutné pro řešení problému, dále se musí popsat algoritmy pracující s těmito daty. Nedílnou součástí je také omezení a rozsahy dat, stanovení okrajových podmínek, kdy ještě programové vybavení spolehlivě a přesně funguje (bez ztráty přesnosti, nebo celkové funkčnosti).
7. V této chvíli již může programátor začít vytvářet zdrojové kódy podle předložené specifikace. Během samotného vývoje programátor již hotové části testuje a odlaďuje, pokud jejich funkčnost lze ověřit nezávisle na jiných částech programového vybavení, které nejsou ještě hotové.
8. Nedílnou součástí vývoje je i nezávislé testování, které by měl provádět někdo jiný než vyvíjející programátor. V nejlepším případě by nezávislé testování mělo probíhat souběžně během vývoje, aby se zabránilo příliš pozdnímu nalezení závažných chyb, které by mohly ovlivnit funkčnost finálního produktu. Nedoporučuje se nechat testování až nakonec, může být už potom pozdě a dopad na případné opravy a práci programátora by byl v některých případech neúnosný. Kromě souběžného vývoje a testování by se verifikace měla provádět už ve fázi vytváření specifikace. Některé chyby mohou totiž vzniknout i kvůli špatně napsané specifikaci a bohužel takové chyby odhalí zřejmě až uživatel, který se později diví, proč nějaká funkce dělá něco jiného, než očekával, přestože podle specifikace je vše v pořádku.
9. Po dokončení vývoje zdrojových kódů následuje kompilace a sestavení finálního softwarového produktu. Hotový produkt je před dodáním testován v konfiguraci, kterou bude později používat i uživatel. Zamezí se tak chybám, které nemohly být v předchozích fázích nalezeny, protože nebyl produkt sestaven do finální podoby. Tyto chyby by mohly totiž souviset až s hotovým produktem nebo s jeho finální konfigurací.
10. Životní cyklus programového vybavení dále pokračuje i během dodání produktu uživateli a během jeho používání. Uživatel komunikuje s dodavateli softwarového produktu a informuje je o případných nalezených chybách, nebo vytváří další požadavky na doplnění funkcí, které by v již hotovém produktu potřeboval v budoucnu.
11. Ze sesbíraných informací od uživatele lze opět sestavit novou specifikaci a nastartovat vývoj nové verze softwarového vybavení, nebo pouze provést drobnou aktualizaci či opravu takzvanou záplatou (podle rozsahu a náročnosti změny). Dalším důvodem pro vytvoření nové verze bývá změna hostitelského prostředí nebo hardwarové specifikace, kdy už stávající softwarové vybavení nelze uspokojivě provozovat v novém prostředí (například nefunguje, nelze vůbec spustit, nebo je zastaralé).

4.4 Vývoj operačního systému nové generace

Dnešní operační systémy prošly předchozím dlouholetým vývojem a obsahují již mnoho funkcí, které mohou využívat jak profesionální uživatelé, tak i široká laická veřejnost. Díky jejich zjednodušenému ovládání a velkému množství užitečných funkcí se stávají operační systémy dostupné (občas dokonce atraktivní) i lidem, kteří by se jinak o výpočetní techniku zřejmě nezajímali.

Z uživatelského hlediska toho nelze mnoho operačním systémům vytknout, jejich stabilita, bezpečnost a použitelnost se již dostala na vysokou úroveň. Problematická však zůstává jejich údržba, aktualizace a následný další vývoj nových verzí. To samé se týká i ostatního programového vybavení, které je určeno pro provoz v těchto operačních systémech. Celý cyklus zaměstnává stále více lidí a bohužel je třeba konstatovat, že některé softwarové produkty se vyvíjejí pořád dokola jen proto, že se během několika let několikrát změnil operační systém, nebo v případě nového operačního systému se změnil hardware. Celý tento proces sice umožňuje zaměstnat velké množství lidí, ale občas se pouze nějaký softwarový produkt pouze „zrecykluje“ s drobnými úpravami, aby se mohl znovu prodat pro nový operační systém, a generuje tak další zisk svým dodavatelům. Uživatelé by jistě více uvítali, kdyby nebylo nutné stále dokola kupovat nové verze téhož software. Také není příjemné software neustále aktualizovat a záplatovat, protože se nový software vydává často předčasně, kdy obsahuje mnoho chyb. Oprava probíhá sice zdarma, ale je to často provázeno i dalšími problémy, které uživatele obtěžují.

Proces vývoje nového operačního systému, nebo jakéhokoliv dalšího programového vybavení by se velmi zkrátil, pokud by se vytvořila pouze vhodná specifikace s popisem očekávaného chování, popsal by se formát používaných dat a statické i dynamické chování programu by bylo také popsáno příslušnými algoritmy. Tyto informace by samozřejmě musely být vhodným způsobem kódovány, aby bylo možno s nimi pracovat i automatizovaně bez účasti člověka. Tohoto cíle by mělo být dosaženo s pomocí znalostního inženýrství, které například v oblasti umělé inteligence u expertních systémů doznalo jistého pokroku.

Takto vytvořená specifikace by mohla být použita k automatizovanému vývoji programového vybavení s menší lidskou účastí, než jak je tomu doposud. Další velkou výhodou tohoto přístupu by bylo i to, že testování takto vyvíjeného programového vybavení by se provádělo rovněž automatizovaně a za běhu. Velká důležitost by potom spočívala na dobře vytvořené specifikaci, která by musela být již sama bez chyb a musela by být dobře použitelná pro následný automatizovaný vývoj. Pokud by se později zjistilo, že je nutné specifikaci rozšířit o popis dalších funkcí, nebo opravit později v ní nalezené chyby, nemělo by být už obtížné programové vybavení vygenerovat znovu s opravenou specifikací včetně opětovného automatizovaného testování za běhu při automatizovaném vývoji. Naopak změny ve specifikacích dnešních vyvíjených softwarových produktů většinou znamenají obrovské problémy, neboť je někdy nutné část již hotové práce zahodit. Bohužel tyto změny mívají i velký dopad na testování, a pokud se některé hotové části vyvíjeného produktu zahazují, byly testovány zbytečně. Často i malá změna v softwarovém produktu generuje velkou práci pro testovací týmy, neboť tyto musí obvykle otestovat větší část programu a jeho uživatelských funkcí, než jaký byl původní dopad do zdrojového kódu.

4.5 Vlastnosti operačního systému nové generace

Nová generace operačního systému by mohla mít tyto vlastnosti, které prozatím postrádáme u současných operačních systémů:

- Automatizovaný vývoj operačního systému by mohl probíhat podle vhodně kódované specifikace.
- Správné chování systému by mohlo být ověřováno automatizovaným testováním, které porovnává aktuální chování systému se specifikací.
- Při změně specifikace v jakékoliv fázi, ať už na počátku, při hotovém produktu, nebo na přání uživatele kdykoliv později, by mělo být možné v softwarovém produktu touto změnou zasažené části znovu vyvinout (vygenerovat) a automatizovaně otestovat aby opět odpovídaly chování které je popsané ve změněné specifikaci.
- Operační systém by v sobě mohl obsahovat celý mechanizmus automatizovaného vývoje programového vybavení. Bylo by tedy možné, aby i sám uživatel, pokud by to uměl, si navrhnul svou vlastní novou specifikaci, která by popisovala nějaký nový program, který by rád používal a tento by byl následně sám a automatizovaně bez další účasti uživatele vytvořen.
- Tak jako programátoři používají různé knihovny funkcí napsané ve zdrojovém jazyce, ve kterém sami programují, mohlo by být možné v operačním systému spravovat knihovny specifikací, které by bylo možné sdílet mezi uživateli. Tyto volně šířené specifikace by si každý uživatel mohl upravit, aby výsledný automatizovaně vyvinutý program měl právě ty funkce, které sám nejvíce potřebuje a naopak by mohl vypustit funkce pro něj nepotřebné.
- Obdobně by bylo možné upravovat i funkce samotného operačního systému podle potřeby uživatele, jen pouhou změnou ve specifikaci. Pochopitelně by bylo nutné dodržet určitá předem daná pravidla, aby mezi jednotlivými systémy zůstala základní míra kompatibility a nedocházelo k tomu, že každý uživatel by měl sice systém, který by byl jemu ušitý na míru a choval se, tak jak on sám potřebuje, ale nebylo by možné sdílet a přenášet data mezi ním a ostatními uživateli.

5. Architektura operačního systému pro automatizovaný vývoj

Spíše než vymýšlet vše znovu od začátku, je rozumné vycházet z architektury a uspořádání dnešních operačních systémů, neboť jejich vzhled, vlastnosti, chování a uživatelské standardy prošly mnohaletým vývojem a ustálily se již na jistém bodě, který mnoha uživatelům vyhovuje. Nemá smysl tedy začínat od začátku, vymýšlet vše znovu a opakovat vývoj který máme již za sebou.

5.1 Rozdělení podle typu integrace v nadřazeném systému

Podle míry integrace do existujícího počítačového systému, se lze na počátku vývoje zamyslet nad těmito způsoby provedení:

- Lze začít i tak, že se vybere již existující operační systém a v něm by se vytvořilo prostředí, které se bude spouštět jen jako jeho nadstavba. Uvnitř této nadstavby lze pak vymodelovat nosné médium, které už může mít svá vlastní pravidla a své vlastní uspořádání, nezávislé na vnějších podmínkách panujících v nadřazeném operačním systému. Vznikla by tak jakási skořápka, uvnitř které by mohl běžet operační systém nové generace. Tento přístup se zdá být pro začátek nejjednodušší a méně zasahuje do zvyklostí uživatele, který by jinak přišel o současný operační systém na který je zvyklý. V případě vytvoření pouhé skořápky běžící pod běžným operačním systémem, by stačilo vytvořit podobnou skořápku pro několik různých konkurenčních operačních systémů, uvnitř kterých by potom nosné médium mělo stejné vlastnosti. Tento přístup by mohl být jednodušší, než vytvářet operační systém, který by se dal provozovat současně na různých hardwarových architekturách.
- Druhou možností je úplné nahrazení současného operačního systému novým, pokročilejším operačním systémem. Výhodou by bylo získání všech systémových prostředků pro své účely a tím i vyšší výkon. Nový operační systém by byl hlavním systémem a žádný nadřazený systém by mu nebránil ve výkonu přidělováním jen omezených systémových prostředků. Nevýhodou by byla vyšší obtížnost vývoje takového systému, protože by musel zajistit sám všechny systémové služby, o které by se v případě použití nadřazeného systému nemusel už starat – byly by mu na vyžádání poskytovány. Další nevýhodou by bylo omezení jen na určitou hardwarovou architekturu a konfiguraci, protože nelze bez předchozích úprav tentýž operační systém provozovat na různých počítačových architekturách, které mají jinou hardwarovou specifikaci (jiný procesor, jinak uspořádanou paměť, jiná rozhraní, atd.).
- Určitým kompromisem proti úplnému nahrazení starého operačního systému novým, by mohlo být i řešení, při kterém by se nový operační systém

zaváděl jen při startu počítače podle potřeby z vyměnitelného datového média a tím by nenarušil původní operační systém dostupný opět po restartování počítače z hlavního datového úložiště.

5.2 Rozdělení podle stupně nahrazení původní integrace

Na současný stav poznání v oblasti operačních systémů je možné z hlediska stupně nahrazení současného modelu integrace navazovat těmito způsoby:

- Nejméně náročná by byla ta varianta, vhodná jako nejsnáze dosažitelný přechodný cíl, že bychom se pro začátek zaměřili jen na automatizovaný vývoj uživatelských aplikací, kdy samotný operační systém – nosné médium pro naše aplikace, by byl vyvinut klasickými metodami a nebylo by tak možné jeho vlastnosti snadno měnit. Chování by bylo pevně dané a během provozu neměnné, jako je tomu u současných operačních systémů. Toto řešení je v rozporu s cílem uvedeným v titulu této práce, neboť se do budoucna předpokládá i nahrazení celého klasického operačního systému systémem vyvinutým novými vývojovými metodami. Je zřejmé, že pro začátek je omezení jen na samotné uživatelské aplikace cílem snáze dosažitelným.
- Pokročilejší variantou je postup, kdy se vytvoří pevné vývojové prostředí. Toto pevné prostředí by bylo schopné využívat předdefinované specifikace definující formáty dat a algoritmy které s těmito daty pracují. Z těchto specifikací lze pomocí připraveného vývojového prostředí dále automatizovaně vyvíjet libovolné softwarové komponenty. Tyto softwarové komponenty by po jejich sestavení a propojení tvořili samotný operační systém nové generace. Tento nový operační systém by se zapouzdřil sám do sebe a uvnitř něj by bylo opět možné spouštět, vytvářet, vyvíjet, propojovat a provozovat další automatizovaně vyvíjené operačnímu systému podřazené uživatelské aplikace. Vznikla by tak hierarchie automatizovaně vyvinutého operačního systému, uvnitř kterého by bylo možné vyvíjet další podřazené automatizovaně vyvíjené uživatelské aplikace.

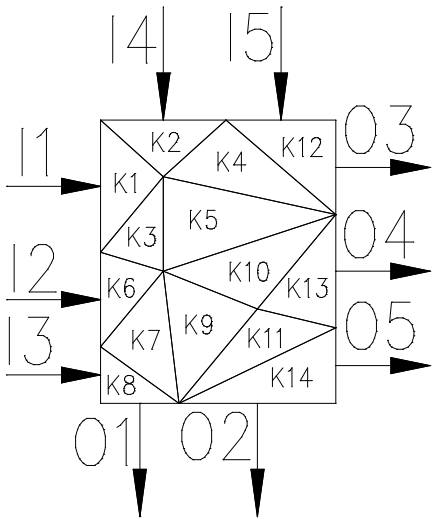
5.3 Předpokládané části operačního systému nové generace

Předpokládané části nového operačního systému by mohly být tyto:

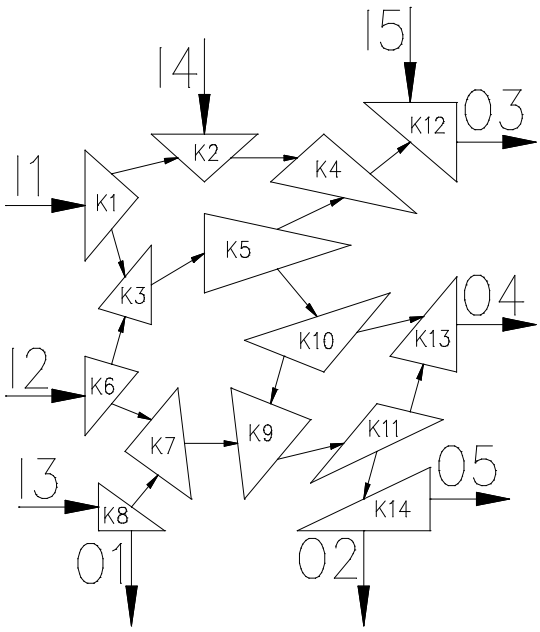
- **Nalezená cílová aplikace:** Tato aplikace je cílem, který by byl nadefinován uživatelem a který by měl operační systém za úkol nalézt. Samotným cílem může být i aplikace, která odpovídá nové verzi operačního systému. Nalezená cílová aplikace je složena z komponent a její komponenty jsou složeny z množiny implementací (komponenty a implementace viz. popis níže). V Plánovači cílů (viz. popis níže) může být zadáno, že uživatel chce nalézt automatizovaně, a s pomocí všech dostupných funkcí, aplikaci jedné z těchto typů:
  - **Zcela nová aplikace:** Všechny její vlastnosti musí být uživatelem nadefinovány před samotným spuštěním hledání. Veškeré

vlastnosti jsou uživatelem stanoveny s pomocí zdrojových specifikací popisujících vlastní aplikaci, její komponenty a implementace použité uvnitř komponent.

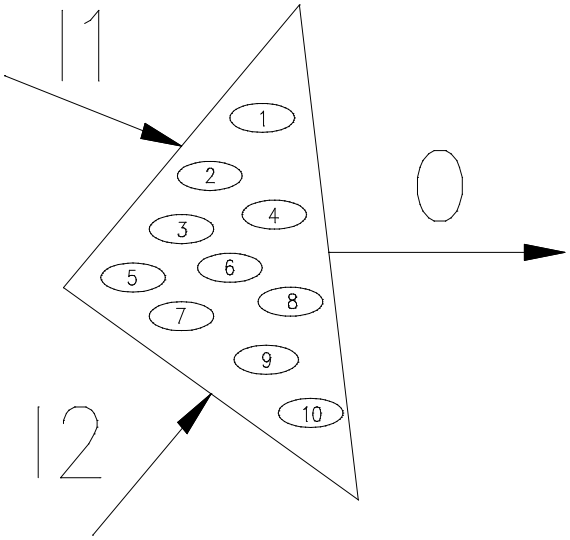
- **Nová verze již existující aplikace:** Uživatel upraví podle své vůle pouze některé již hotové zdrojové specifikace již existující aplikace a spustí hledání nové verze. Docílí se stavu, kdy pouze některá část aplikace, komponenty, nebo specifikace změní své chování, které bude pak po skončení hledání blíže očekávanému chování.
- **Nový operační systém, nebo nová verze existujícího systému:** Analogicky to samé co platí pro aplikace, platí i pro samotný operační systém, který je sám aplikací.



Obr. 5.3.1 Znázornění hledané aplikace složené z více komponent K, obsahující několik vstupů I a několik výstupů O

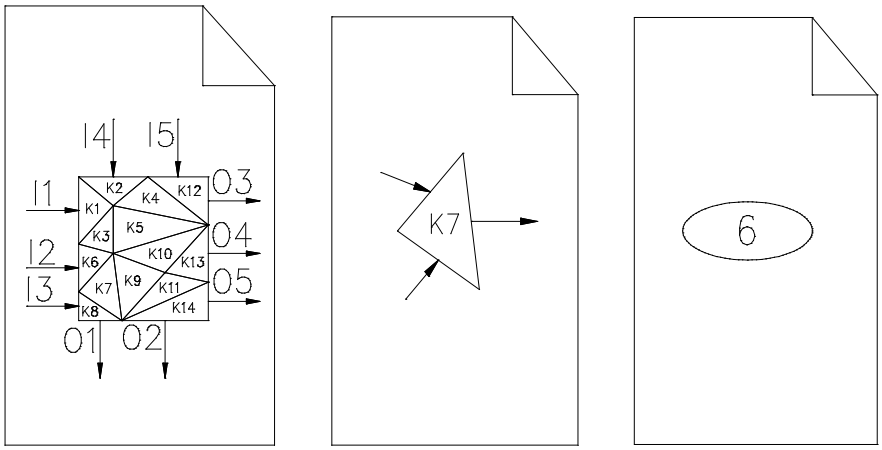


Obr. 5.3.2 Znázornění komponent K a jejich propojení pro přenos dat mezi sebou, které jako celek tvoří aplikaci.



Obr. 5.3.3 Znázornění jedné komponenty K a jejich vstupů I a výstupu O, jejíž chování je utvořeno spoluprací skupiny vnitřních implementací (např. 1-10).

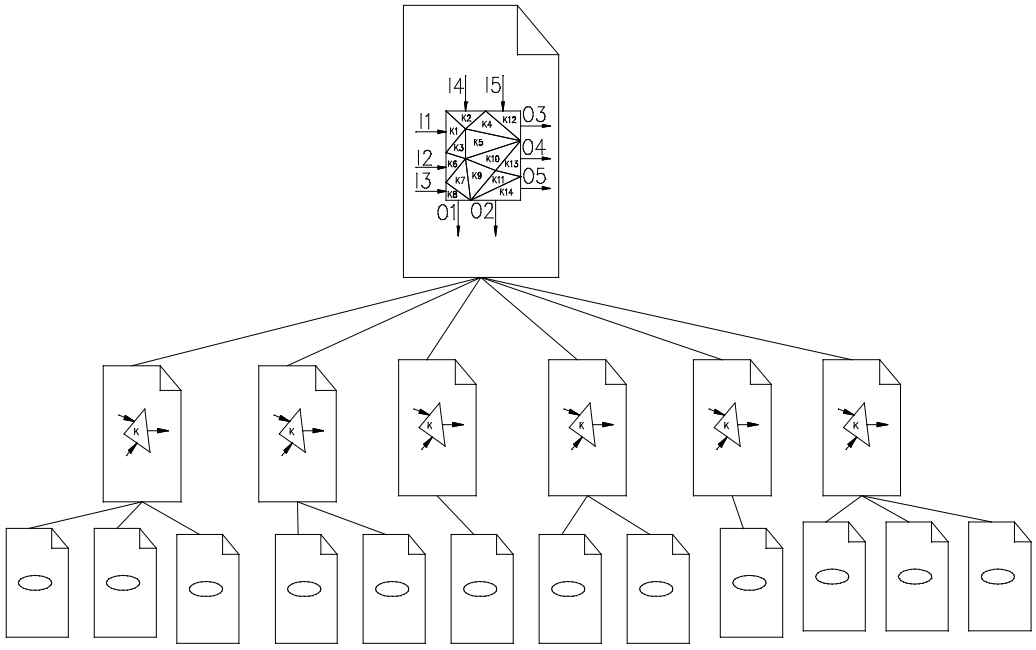
- **Editor zdrojových specifikací:** Slouží k nadefinování specifikací uživatelem podle přesně daných pravidel, tak aby bylo možné k těmto zdrojovým specifikacím automatizovaně nalézat jejich implementace, se kterými by se daly párovat, propojovat je ve složitější komponenty a následně tyto komponenty spojovat do aplikací. Existovalo by několik typů zdrojových specifikací:
  - **Zdrojová specifikace popisující aplikaci:** Slouží k popisu, jaké komponenty jistá aplikace obsahuje a jakým způsobem jsou uvnitř ní spolu komponenty propojeny (jak si předávají komponenty mezi sebou data, jaké mají vstupy a jaké výstupy)
  - **Zdrojová specifikace popisující komponentu:** Slouží k popisu, jaké implementace jsou obsaženy v nějaké komponentě a jakým způsobem jsou uvnitř ní spolu implementace propojeny (jak si předávají implementace mezi sebou data, jaké mají vstupy a jaké výstupy).
  - **Zdrojová elementární specifikace:** K této zdrojové specifikaci nejnižší úrovně se nalézají (párují) vnitřní implementace s pomocí Párovacího systému (popsáno níže).



Obr. 5.3.4 Znázornění 3 typů specifikací, které na různých úrovních popisují složení a chování aplikace (vlevo), složení komponenty (uprostřed) a chování vnitřní implementace, nebo skupiny vnitřních implementací (vpravo).

- **Strukturovaná databáze zdrojových specifikací:** Vhodně kódovaný registr specifikací popisující základní datové typy a algoritmy které s nimi umí pracovat. Součástí by byly i specifikace popisující komponenty umožňující ostatním specifikacím po nalezení a spárování s jejich implementacemi se propojovat do větších celků – složitějších komponentů. Skupina komponentů by potom po propojení a spuštění tvořila celou aplikaci (nebo operační systém). V případě že by tvořila jinou aplikaci než operační systém, byla by tato určena pro spuštění v operačním systému. Rekurze tvorby operačních systémů kdy jeden operační systém přispěje k vytvoření svého nástupce je možná, tak jako je to možné například provést kompilátorem nějakého obvyklého programovacího jazyka. Jako příklad lze uvést kompilátor, který by byl sám napsán v nějakém programovacím jazyce a po sestavení tohoto kompilátoru, by s tímto novým kompilátorem šlo vytvořit novou verzi kompilátoru téhož programovacího jazyka a nahradit ten předchozí kompilátor.
- **Strukturovaná databáze nalezených implementací:** Ke každé elementární specifikaci by náležela, buď jedna upřednostněná implementace, nebo skupina více či méně rovnocenných variant implementací. Nacházení a párování cílových implementací s jejich zdrojovými specifikacemi se předpokládá být automatizované – toto je hlavním bodem celé úvahy a tomuto bodu je zde věnováno nejvíce prostoru. Skupina příbuzných implementací, jejichž spolupráce je podrobně popsána ve specifikaci komponenty tvoří potom chování komponenty. Spolupráce skupiny implementací vztahující se ke stejnému chování popsaného ve specifikaci, je řízeno pomocí většinové volby. V některých případech se chování nalezených implementací může lišit, především pokud se na vstupech objeví data, nebo kombinace dat, které nejsou ve specifikaci definovány. Zde se uplatní zákony pravděpodobnosti a skupina paralelně zapojených implementací vypočítá pro zadaný vstup nějaký výsledek a tento výsledek je porovnán a který výsledek vyšel stejně u více implementací, ten bude zvolen pro výstup. Simuluje se zde princip dedukce, kdy ze známého

chování pro známé případy se odvozuje chování pro případy, které nejsou popsané. Tento princip se dá přirovnat k chování umělé neuronové sítě, která je natrénovaná na určitou množinu dat. Pokud se vyskytnou data, která nejsou obsažena v trénovací množině, hledaný výsledek je dopočítán s vědomím, že může být, ale nemusí být správný. Čím více paralelních implementací se podílí na výpočtu a volbě výsledku v jednom okamžiku, tím větší je pravděpodobnost správného výsledku.



Obr. 5.3.5 Znázornění hierarchie jednotlivých zdrojových specifikací podle toho jak popisují cílovou aplikaci na různých úrovních: Nahoře se nachází specifikace celé aplikace, uprostřed specifikace jednotlivých komponent a dole specifikace jednotlivých implementací.

- **Administrátor komponent:** Slouží pro určení a naplánování s pomocí databáze zdrojových specifikací, které zdrojové specifikace jsou nutné pro vytvoření určité komponenty vyvíjeného software. Nalezené, k nim spárované implementace dle jejich specifikací pak administrátor komponent pospojuje do větších celků, které potom tvoří celou komponentu. Následně jednotlivé komponenty automaticky vyvíjeného software administrátor komponent propojí tak, aby spojené komponenty tvořili kompletní softwarové vybavení a plnili tak svou očekávanou funkci.
- **Interpret implementací:** Tato část systému je zodpovědná za provádění (interpretaci, exekuci) jednotlivých implementací z databáze podle obsahu jednotlivých komponent. To by spočívalo v převzetí hodnot vstupů do implementace a provedení algoritmu (dříve nalezeného automatizovaně ze specifikace) zakódovaného uvnitř implementace, který má určitý počet kroků a ty by se iterativně opakovaly. Následoval by přenos nově vypočítané informace na výstupy z implementace. Tyto odevzdané výstupní hodnoty by byly opět využity jako vstupy do nové navazující zřetězené implementace. Celý řetězec interpretovaných implementací (jehož struktura by byla daná nějakou specifikací) by tvořil chování celé komponenty. Mezi sebou propojené komponenty tvoří celou aplikaci tím

způsobem, že jsou na té nejnižší úrovni realizované interpretem iterativně interpretujícím podřazené implementace.

- **Párovací systém, generátor variant implementací**: Jedná se o genetickým programováním řízený systém hledání a párování implementací se zdrojovými specifikacemi. Při tomto procesu se obvykle nalézá více variant implementací téže specifikace. K jedné zdrojové specifikaci je pak obvykle možné nalézt více variant implementací a spravovat je jako množinu různých variant implementací. Důvodem může být i to, že ve chvíli kdy se určité varianty implementací naleznou, nemusí být ještě zřejmé, která z nich bude pro pozdější použití nejvhodnější. Výběr lze odložit na pozdější dobu až do chvíle, kdy například jedna varianta bude poskytovat během provozu přesnější a správnější výsledky než jiná varianta. V jiném případě to může být zase odlišná varianta než prvně, tedy je vhodné jich spravovat více najednou. Pro určení správného výsledku lze v danou chvíli použít i většinový volební systém, kdy máme lichý počet variant a část z nich hlasuje například pro hodnotu 1 a část z nich pro hodnotu 0. Vybere se potom jako nejpravděpodobnější ta hodnota pro výstup, pro niž se rozhodlo více variant jedné implementace. Nabízí se i pokročilejší řešení: Podobně jako existují hardwarově řešené akcelerátory počítačové 3D grafiky obsahující mnoho jader běžících paralelně na jednom čipu, šlo by jistě vytvořit i specializované, podpůrné hardwarově řešené karty zasunuté do základní desky počítače, které by celý proces genetickým programováním řízeného hledání a párování specifikací s implementacemi urychlily díky paralelním výpočtům. Tyto paralelní výpočty zkracující významně dobu hledání by mohly být umožněny díky specializovaným mnohojaderným procesorovým čipům na kartě.
- **Plánovač cílů**: Uživatel, který by byl momentálně spokojen s konkrétním daným stavem, nebo verzí operačního systému a nepotřeboval by vyvíjet žádné nové softwarové vybavení (jak operační systém samotný, nebo aplikace v něm běžící), ten by zřejmě neměl ani žádné cíle a nepotřeboval by tudíž ani Plánovač cílů. Předpokládá se naopak, že v případě že by uživatel potřeboval, aby se něco v systému vyvinulo do jiného stavu, než ve kterém se právě systém nachází, nebo aby systém dospěl k nějaké nové aplikaci, stačilo by nadefinovat nový cíl. Mohlo by existovat souběžně několik naplánovaných cílů s různými prioritami a systém by se snažil s pomocí nadefinovaných zdrojových specifikací přidružených k těmto cílům (které by dostatečně přesně popisovaly detaily cílů), spárovat postupem času s pomocí Párovacího systému tyto specifikace s v té chvíli neexistujícími (a později nalezenými) implementacemi. Ve chvíli, kdy by byly postupně všechny cílové implementace nalezeny, následuje sestavení komponent s pomocí nástroje zvaného Administrátor komponent, který propojí všechny související implementace do větších celků a v dalším kroku se tyto komponenty propojí dohromady a vytvoří cílovou aplikaci. Může to být i cílová nová verze samotného operačního systému. Během celého procesu jsou v každé jednotlivé fázi sestavování všechny dílčí kroky automatizovaně kontrolovány tak, že na úrovni implementací, na úrovni sestavených komponent a na úrovni sestavených aplikací se musí splňovat všechna pravidla, která se k nim vztahují, popsaná ve zdrojových

specifikacích. Pokud nebyla nalezena žádná neshoda se žádnou přidruženou zdrojovou specifikací, předpokládá se, že cíl byl splněn a výsledek se předloží uživateli s upozorněním, že došlo ke splnění zadaného cíle. V opačném případě hledání probíhá dál, dokud nebudou splněna všechna pravidla. Předpokládá se, že počítačový systém obsahující a provozující zde popisovaný operační systém nové generace, by běžel nepřetržitě, neboť po naplánování cílů by bylo spuštěno hledání, které však pravděpodobně bude časově velmi náročné. Postupně se musí nalézt všechny implementace pro všechny zdrojové specifikace a to je úkol velmi časově náročný, když víme, že se využívá hledání řízené genetickým programováním. Dává tedy smysl, aby hledání probíhalo na pozadí, kdy uživatel mezitím používá různé aplikace pro své vlastní účely. Jakmile by uživatel práci ukončil, nechal by systém běžet, aby dále pokračoval v hledání, dokud nebudou všechny implementace nutné pro sestavení cílové aplikace nalezeny. Systém by se pak podle potřeby sám vypnul. Realisticky vzato je spíše ale pravděpodobné, že by se systém nevypínal nikdy a vlastně by se neustále dynamicky vyvíjel, podle právě stanovených cílů.

- **Kontrola shody**: Na všech úrovních se neustále automaticky kontroluje shoda nalezených implementací, složených komponent a chování sestavené aplikace s popsaným chováním ve všech zdrojových specifikacích všech úrovní, které jsou momentálně přidružené k danému cíli. Shoda všech zdrojových specifikací s celkovým chováním je kritériem pro dosažení cíle a ukončení hledání.

- **Rozhraní pro propojení s vnějším prostředím:** Tato část operačního systému obsahuje pevně naprogramovanou složku, u které se příliš nepředpokládá změna běžným uživatelem a není ani součástí cílů které plánuje uživatel. Využívá se pouze informace o napojení na konkrétní poskytované části rozhraní. Prostřednictvím tohoto rozhraní by bylo možné propojit vnitřní, automatizovaně vyvíjené aplikace, se skutečnými systémovými zdroji počítače. Podle míry a stupně integrace do systému to mohou být jak vazby na systémové služby nadřazeného, klasického operačního systému, který vnitřnímu systému zprostředkovává např. čtení znaků z klávesnice, změnu pixelů na obrazovce, posílání dat přes síťové rozhraní, tak i využívání služeb operačního systému pro vytváření a ovládání oken, tlačítek, apod. Pokud není použit žádný nadřazený operační systém a jsou dostupné veškeré systémové zdroje počítače v nejvyšší možné míře hned po spuštění počítače a načtení systému do paměti, tak toto rozhraní může zprostředkovávat komunikaci přímo s hardwarovými funkcemi počítače a jeho systémovými službami. Logicky právě toto rozhraní musí být pevně nadefinováno, jinak by nebylo možné vytvářet automatizovaně vyvíjené aplikace, které by uměly obsluhovat klávesnici, snímat pozice myši, zapisovat na obrazovku, číst z pevného disku, zapisovat na disk, apod.

5.4 Rozsah a hloubka popisu, směr zaměření práce

Vzhledem k rozsáhlosti tématu bylo zvoleno se zaměřit v této práci jen na ty části operačního systému nové generace, které by mohly být pro jeho realizaci důležitější než jiné části (méně zajímavé části by byly řešeny obecně známými a běžnými postupy) a u nichž se předpokládá, že jejich hlubší analýza bude přínosem. Nalezneme zde především popis vztahující se k:

- Zdrojovým specifikacím,
- Nalezeným implementacím,
- Interpretu implementací,
- Párovacímu systému,
- Kontrole shody nalezené implementace se specifikací,
- Rozhraní pro propojení s vnějším prostředím

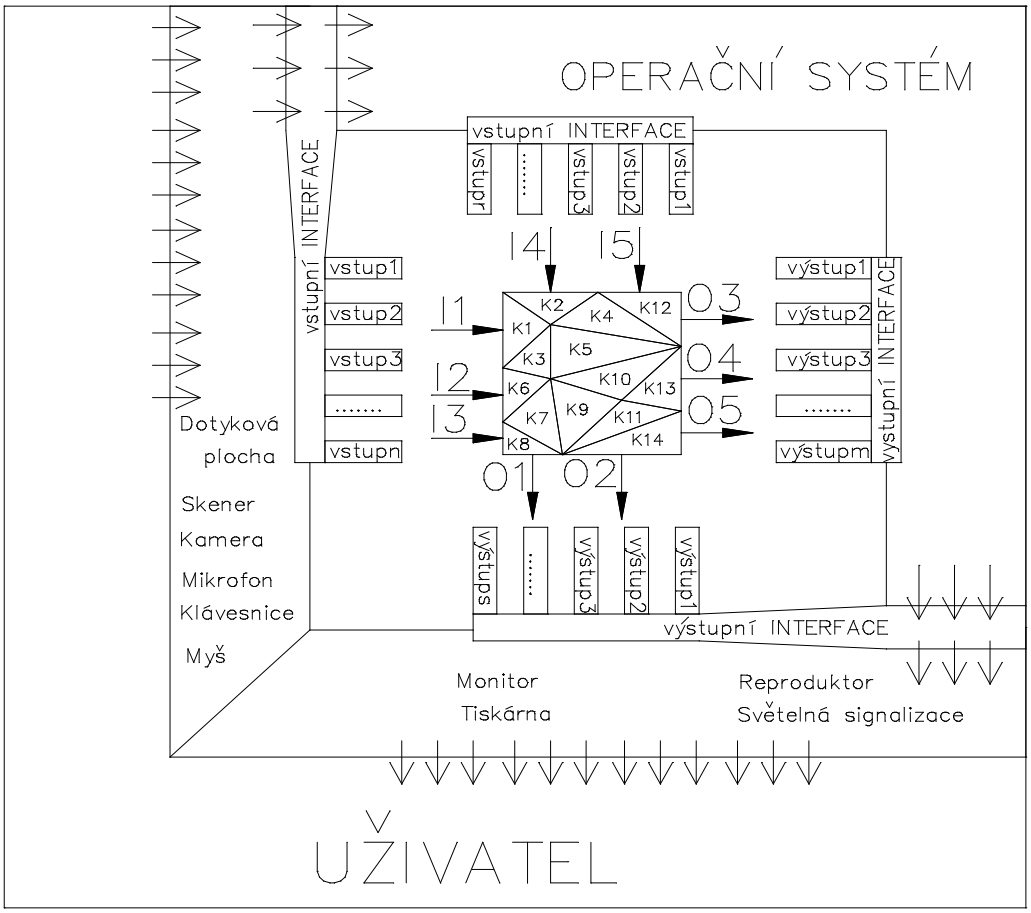
Tyto zasluhují hlubší pozornost a další popis ve zbytku práce je věnován už jen těmto částem.

6. Rozhraní pro propojení s vnějším prostředím

Operační systém komunikuje s uživatelem prostřednictvím svých rozhraní (interface), která jsou napojena na periferní zařízení. Přístup k těmto periferním zařízením je rovněž zprostředkován pomocí týž rozhraní i uživatelské aplikaci. Z pohledu aplikace je spojení s vnějším světem přímé, ve skutečnosti je ale zprostředkované operačním systémem. Uživatel může rovněž nabýt dojmu, že komunikuje přímo s aplikací, mezi ním a aplikací se však nachází rozhraní, které je opět řízeno operačním systémem. Kromě periferních zařízení pro komunikaci s uživatelem, aplikace navíc komunikuje prostřednictvím svých rozhraní přímo s operačním systémem pro využívání dalších interních služeb, které mohou být však skryty před uživatelem – nemusí o nich mít tušení. Jako příklad interních služeb můžeme uvést čtení systémového času, synchronizace pomocí časovačů, čtení nebo zápis do souborů na interním pevném disku, apod. Na úrovni návrhu aplikace, propojení s externími a interními rozhraními může být definováno jako součást specifikace aplikace nebo její komponenty. Jaký je formát použitých dat, nebo jak probíhá komunikace s vnějším světem, to je rovněž předmětem návrhu aplikace. Pro komponenty aplikace není podstatné, jestli si právě vyměňují data dvě interní komponenty mezi sebou, dvě aplikace mezi sebou, aplikace s operačním systémem, nebo jestli data přicházejí nebo odcházejí z/do vnějšího prostředí ve kterém se nachází uživatel. Obrázek **Obr. 6.1** popisuje principy na jakých je založeno propojení aplikace s operačním systémem a s vnějším světem z kterého je řízena uživatelem. Obrázek je rozdělen na několik částí. Levá a spodní část představuje uživatele ve vnějším světě, horní a pravá část představuje operační systém. Uprostřed se nachází aplikace složená z komponent, která vlastní několik vstupů a několik výstupů, které jsou sami napojeny na různá rozhraní. Některé

vstupy a výstupy jsou určeny pro vnitřní komunikaci s operačním systémem a jiné vstupy a výstupy slouží pro komunikaci s vnějším světem. Ve skutečnosti aplikace komunikuje pouze s operačním systémem. Část vstupů a výstupů je však svou povahou určena k propojení s vnějším světem a toto propojení je tedy operačním systémem zprostředkováno. O jaké vstupy a výstupy se právě jedná je vždy uvedeno v konkrétní specifikaci dané aplikace, nebo její komponenty.

Přestože sám operační systém je v této práci uvažován jako aplikace schopna evolučního vývoje a do všech detailů popsána svými specifikacemi, musí existovat jistá neměnná vnitřní konstrukce, která je pevně daná. Lze uvažovat o tom, že tato konstrukce by mohla být realizována podobně, jako svého času sloužil BIOS (Basic Input Output System) na základní desce počítače. Jednalo se v té době o software, který zprostředkovával propojení hardware počítače se softwarem a poskytoval sadu základních služeb, která musela být pro všechny počítače stejná, aby veškerý software správně fungoval na všech systémech. Hardwarová konfigurace těchto systémů mohla být jiná, ale když měly nad sebou standardní sadu služeb, nebyl to pro software žádný problém, aby se i přes různé odlišnosti s počítačovými komponentami bezchybně spojil. Základní softwarová konstrukce pro propojení operačního systému s počítačovými komponentami, rozhraními a periferiemi by byla naprogramována konvenčními metodami a do procesu evoluce by nebyla zahrnuta. Jisté základní principy a mechanizmy musí být vždy nastaveny jako neměnné. Na těchto základech lze potom dále budovat složitější struktury, které již neměnné být nemusí a ty svou složitostí mohou dále překonávat původní pevně dané podloží.



Obr. 6.1 Uživatel ovládá aplikaci skrze vstupy a výstupy zprostředkované přes Operační systém

7. Zdrojové specifikace

Název zdrojová specifikace byl zvolen pro svou podobnost se slovním spojením zdrojový kód. Tato podobnost má však hlubší význam. U zdrojového kódu jde o přepis algoritmu člověkem do pro počítač srozumitelné formy, obvykle s použitím nějakého programovacího jazyka (obvykle procedurálního typu). Ze zdrojového kódu se vygeneruje aplikace, která je pak používána uživatelem a dle potřeby spouštěna v operačním systému. Tento proces je většinou jednosměrný, ze zdrojového kódu se generuje aplikace, ale z již vygenerované aplikace obvykle nelze získat zpět zdrojový kód.

V operačním systému nové generace se pro popis algoritmů předpokládá užití zdrojových specifikací místo zdrojového kódu. Hlavní rozdíl spočívá v tom, že specifikace má charakter popisný podle předložených případů, kde se vlastnosti algoritmu obecněji popisují očekávaným chováním. Tedy jinak, než jak jsou definovány ve zdrojovém kódu konkrétními programovými kroky. Zdrojový kód tedy obsahuje již realizovaný algoritmus ve formě programových kroků. Je přesně dána posloupnost těchto kroků, aby bylo dosaženo správného chování popisovaného algoritmu. Tyto kroky musí být popsány velice podrobně programátorem. Tento proces lze velmi obtížně zautomatizovat a účast člověka je zatím nutná. Naopak u zdrojové specifikace se pro popis algoritmu nepředpokládá definice pomocí programových kroků a popis je více strukturální než procedurální. Správná posloupnost programových kroků je v tomto případě nazývána implementací, která se vztahuje ke své zdrojové specifikaci a je hledána automatizovaným procesem genetického programování. K jedné zdrojové specifikaci lze během evolučního procesu genetického programování nalézt více variant svým chováním podobných implementací, podobně jako jeden algoritmus lze realizovat různými variantami zdrojového kódu. Která varianta je nejvhodnější, která nejlépe dokáže řešit daný problém, nemusí být předem známo, a proto je lepší uchovávat souběžně více variant, které se vztahují k jedné zdrojové specifikaci. Hlavní výhoda popisu algoritmů pomocí zdrojových specifikací je ta, že problém nemusí být popsán vyčerpávajícím způsobem. Vytvoří se dostatečně velká sada případů popisující daný problém a některé aspekty, které nejsou přímo uvedeny, mohou samy z již zadaných informací vyplynout. Pro všechny případy, které jsou uvedeny ve specifikaci, je výsledné chování algoritmu zřejmé – to je také cílem implementace, aby se chovala podle zadaných případů. Když se však později na vstupy nalezené implementace předloží kombinace dat, která se ve specifikaci nenachází, implementace navrhne řešení vycházející z naučeného chování u známých případů – simuluje tak extrapolaci výsledku u neznámého zadání. Pro účely extrapolace daného problému se proto hodí mít k dispozici co nejširší základnu variant implementací, kdy všechny implementace mohou hlasovat a výsledkem bude zvolen ten výstup, který je nejčetnější. Díky zákonům pravděpodobnosti, při dostatečném množství případů uvedených ve specifikaci popisující nějaký problém a při dostatečně velkém počtu variant implementací, se budeme blížit ke správným řešením i pro případy extrapolace řešení u neznámého zadání. Toto není možné u procedurálního popisu algoritmů. Algoritmus

se bude chovat pouze jedním způsobem a to vždy stejně, lhostejno kolik máme variant zdrojového kódu.

U každého z typů zdrojových specifikací představíme jejich vlastnosti, a čím se liší od ostatních typů zdrojových specifikací. V jednotlivých specifikacích se uvádí informace o různých datových spojích mezi komponentami, nebo mezi aplikací a svým okolím. Datové spoje jsou realizovány kanály dat, které mohou mít libovolnou bitovou šířku a tak reprezentují různé typy dat. Nejjednodušším typem spoje je spoj logický, který nabývá pouze hodnoty 0 nebo 1. Pro realizaci spojů například z výstupu jedné komponenty do druhé, je důležité, aby byla stejná šířka dat na straně výstupů a vstupů – informace by se neměla ztrácet, ale ani vznikat mimo komponenty.

7.1 Zdrojová specifikace popisující aplikaci

Základní parametry aplikace jsou popsány na úrovni zdrojové specifikace pro aplikaci a mezi ně patří například (vše je uvedeno jen jako možné, ale ne jediné řešení):

- Seznam všech vstupů a výstupů s okolním prostředím.
  Příklad položek seznamu kde O – znamená output, I – znamená input, X – znamená externí: IXkeyboard, IXcom1, IXlpt1, IXmicrofon, OXscreen, OXprinter, OXspeaker
- Seznam všech vstupů a výstupů pro komunikaci s operačním systémem.
  Příklad položek seznamu kde O – znamená output, I – znamení input, S – znamená operační systém: OSopen_file, OSwrite_to_file, OSset_time, ISread_time, IStimer1, ISreset_application
- Seznam všech komponent, ze kterých se aplikace skládá.
  Příklad položek seznamu: Ksecti_cisla, Kzaokrouhleni, Kzjisti_delku_seznamu
- Seznam všech spojů mezi komponentami ve formě orientovaného grafu, kdy z jednoho uzlu grafu nebo do jednoho uzlu grafu může vycházet/vstupovat několik spojů m:n, kde m je počet vstupů do uzlu a n je počet výstupů z uzlu. Každý spoj má jako parametr svou datovou šířku v počtu bitů, která musí souhlasit jak na straně odesílatele informace, tak i na straně příjemce informace.
  Příklad jedné položky seznamu: Kkomp1Ocislo1/Kkomp2Icislo1, kde Kkomp1 je jedna komponenta jejíž výstup „Ocislo1“ je směrován do komponenty Kkomp2, do jejího vstupu „Icislo1“.

7.2 Zdrojová specifikace popisující komponentu

Základní parametry komponenty jsou popsány na úrovni zdrojové specifikace pro komponentu a mezi ně patří například (vše je uvedeno jen jako možné, ale ne jediné řešení):

- Seznam všech vstupů do komponenty.
  Příklad položek seznamu kde číslo za „:“ značí šířku datového spoje: Icislo1:8, Icislo2:8, Ikladne:1, Iword:16
- Seznam všech výstupů z komponenty.
  Příklad položek seznamu kde číslo za „:“ značí šířku datového spoje: Odelka:8, Oznak:8, Opotvrzeni:1, Osouradnicex:16
- Seznam všech interních spojů, které jsou použité pro propojení dat ze dvou specifikací jen uvnitř komponenty.
  Příklad položek seznamu pro vnitřní vstupy kde číslo za „:“ značí šířku datového spoje: Xinterni_spoj1:4, Xprenos_dat:8
  Příklad položek seznamu pro vnitřní výstupy kde číslo za „:“ značí šířku datového spoje: Yinterni3:8, Yzaokrouhleno:12
- Seznam všech zdrojových elementárních specifikací popisující komponentu.
  Příklad položek seznamu, kde položky uvnitř závorek určují které vstupy a které výstupy se u specifikace použijí. Doporučuje se v seznamu mezi závorkami na začátek uvést vstupy a na konec výstupy: Esoucet(Icislo1, Icislo2, Ocislo3, Ykladny_vysledek), Epreved_znak(Iznak1, Xznak2, Oznak, Ypotvrzeno)

7.3 Zdrojová elementární specifikace

Zdrojová elementární specifikace se nachází na nejnižší úrovni popisu celé aplikace. Tato slouží párovacímu systému k hledání implementací a variant implementací, které se mají blížit svým chování k chování popsanému v elementární zdrojové specifikaci. Tato specifikace je popsána svými vstupy, svými výstupy, bitovou šířkou použitých vstupních nebo výstupních dat a především se zde nachází tabulka popisující chování hledané implementace. Jsou dvě možnosti, jak může být chování hledané implementace popsáno:

- Popis vyčerpávajícím způsobem: Jsou popsány všechny možné kombinace hodnot všech vstupů a výstupů. Nemůže se stát, že by se nalezená implementace později odchylovala svým chováním od předepsaných stavů ve všech jednotlivých případech, které mohou nastat. Nevýhodou je to, že se musí vytvořit rozsáhlá množina případů, aby všechny možné stavy byly podchyceny. Poté stačí nalézt pouze jednu variantu implementace, která splní všechny uvedené případy vstupních a výstupních dat.
- Popis neúplnou množinou stavů: U složitého chování, u kterého neznáme způsob, jak algoritmus generující data funguje, ale máme dostatečné

množství dat pro jeho popis, můžeme zadat alespoň tato data, která jsou v tu chvíli dostupná. Chování bude namodelováno i bez znalosti jeho podstaty. V tomto případě je lepší nalézt pomocí párovacího systému větší množství variant implementací, než pouze jednu variantu, aby bylo možné volit domnělé výsledky většinovou volbou. Díky zákonům pravděpodobnosti je velká šance, že při použití více variant implementací pro volbu aktuálních výstupů, bude zvolený výsledek správný. To jestli je popis úplný, nebo neúplný, to by mělo být uvedeno již v nastavení specifikace, aby se párovací systém mohl rozhodovat, jestli hledat více variant, nebo zda to není pro daný případ potřeba.

Podle počtu vstupů, výstupů a jejich datových šířek, párovací systém se sám pokusí rozdělit vstupní a výstupní data na menší datové šířky a tím vytvářet paralelní implementace, které se na venek mohou tvářit jako jedna implementace, ale uvnitř může dojít k rozštěpení na menší části použité pro hledání genetickým programováním odděleně. Důvodem může být přílišná náročnost pro hledání implementace a hrozí, že hledání by bylo neúspěšné, pokud by byly použity pro hledání implementace více vstupů a výstupů o dlouhých bitových šířkách najednou. Párovací systém může vytvořit několik variant vnitřního štěpení informací a pokusit se sám o hledání implementací s různými nastaveními. Během hledání by se mělo dojít na to, které varianty vedou rychleji k hledanému výsledku a které naopak trvají neúměrně dlouho a tyto by byly ukončeny.

Výhodou tohoto způsobu popisu známého i neznámého chování algoritmů je to, že systém může sám během provozu ověřovat, které případy kdy nastávají a provádět u nich statistiku, která se může hodit pro další analýzy. Součástí této statistiky může být i to, že u některých předem neznámých stavů, které nebyly ve specifikaci nadefinovány, ale vyskytují se během provozu, je vždy snaha nalézt správné řešení i pro tyto případy. Pokud evidentně dochází při výběru k chybnému chování, uživatel může i později provést korekci přidáním konkrétních dodatečných dat do specifikace, podle kterých lze nalézt nové varianty implementací, které by už později měly vykazovat správné chování.

Výhodou je také to, že kdykoliv později kdyby bylo potřeba dříve nadefinované chování změnit, stačí pouze změnit existující data, aktualizovat je a provést opětovné hledání implementací jen na tyto změněné části. Celá aplikace je díky tomu dynamickým systémem, který je schopen kdykoliv poupravit své chování podle aktuálních potřeb a požadavků uživatele.

8. Cílové implementace, interpret a kontrola shody

8.1 Reprezentace implementace binárním vektorem

Pro potřeby automatického vývoje algoritmů - implementací s použitím genetického programování, je vhodné navrhnout obecný zápis algoritmů - implementací genetickým jazykem a softwarové prostředí, které je schopno tímto genetickým jazykem zapsané algoritmy provádět - interpret. Cílem návrhu je algoritmus zapsaný genetickým jazykem, který by měl mít tyto vlastnosti [15]:

- Jednoduchost zápisu (při nesplnění požadavku se prudce zvyšuje výpočetní zátěž),
- robustnost (při nesplnění požadavku hrozí zablokování celého algoritmu při jediné chybě v algoritmu),
- optimální variabilita zápisu (nízká variabilita, nebo příliš vysoká variabilita vede k problémům při dosažení cíle – nedostatek nebo velký nárůst kombinací),
- upřednostnění paralelního zpracování před sekvenčním nebo procedurálním (Několik paralelních současně běžících větví algoritmu může dojít k výsledku různými cestami a přitom nehrozí zablokování celého algoritmu chybou jediné větve – viz robustnost.),
- realizovaný algoritmus by měl být deterministický, aby bylo možné vždy dojít ke stejnému výsledku při stejných vstupních datech,
- zapsaný algoritmus by měl být pokud možno do jisté míry čitelný i pro člověka, aby byl schopen zpětně pochopit a analyzovat vnitřní chování algoritmu nebo jej následně ručně opravovat či dooptimalizovat (nepřehledné a složité zápisy algoritmu neumožňují další ruční zdokonalování, úpravy nebo jen pouhou analýzu vnitřního chování),
- celý systém (zapsaná implementace + její interpret) by měl být schopen realizovat libovolný myslitelný (deterministický) algoritmus včetně virtuální emulace sama sebe, např. měl by být schopen emulovat libovolný známý počítačový mikroprocesor a program na něm běžící v jazyce tohoto mikroprocesoru (v mezích technické realizovatelnosti dle časové a kapacitní náročnosti – teoretická proveditelnost).

Ke splnění výše uvedených předpokladů se velmi blíží návrh, který je popsán v následující kapitole. Protože je tento návrh založen na periodickém zpracování informací, které jsou atomizovány na jednotlivé bity (jsou reprezentovány řadou buněk) a následně jsou dále zpracovávány pomocí logických funkcí, je tento systém nazýván „Celulární procesor logických funkcí“.

8.2 Celulární procesor logických funkcí

Celulární procesor logických funkcí je používán pro výpočet vhodnosti jedinců během evolučního procesu (po nalezení hledané implementace slouží také k provádění zakódovaného algoritmu) a jeho základní stavební jednotkou je jedna buňka $B_{n,k}$, která v součinnosti s ostatními buňkami tvoří jednorozměrný buněčný automat s absolutním adresovým prostorem v rozsahu < 0, n_{max} > [16]. Ve skutečnosti se nejedná přímo o klasický buněčný automat, protože není splněna podmínka homogenity ve všech ohledech, a protože jednotlivé buňky mohou obsahovat jinou přenosovou funkci a mohou mít i různé spoje na okolní buňky. Ostatní vlastnosti již dostatečně odpovídají vlastnostem buněčného automatu.

Během procesu evoluce u genetického programování, každý člen populace (každý člen Generace) je současně i kandidátem na řešení algoritmu a jedná se o binární vektor složený z buněk, kde každá z nich obsahuje binární informaci o svém aktuálním výstupu a jako celek určují chování hledaného algoritmu (implementace) tak, že se generují výstupy z předložených vstupů během opakování určitého počtu cyklů.

V již realizovaném programu, který provádí párování specifikací s implementacemi je každá z buněk $B_{n,k}$ kódována 32 bity a představuje binární informaci (její výstup) [20]:

- 1 bit: aktuální platná hodnota vypočítaná pro aktuální krok "k" vyjádřená jako
  $y_{n,k} \in \{(0)_2, (1)_2\}$, (8.2.1)
- 1 bit: předchozí platná hodnota výstupu buňky z předchozího kroku "k-1" jako
  $y_{n,k-1} \in \{(0)_2, (1)_2\}$, (8.2.2)
- 11 bitů: relativní adresa ukazující na první okolní buňku
  $a_n = < -(n_{max}+1)/2 , +(n_{max}+1)/2 - 1 >$, (8.2.3)
- 11 bitů: relativní adresa ukazující na druhou okolní buňku
  $b_n = < -(n_{max}+1)/2 , +(n_{max}+1)/2 - 1 >$, (8.2.4)
- 8 bitů: logická funkce F_n kódovaná 8mi-bitovou tabulkou (jeden bajt)
  $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}] = < (0)_{10} , (255)_{10} >$. (8.2.5)

Během každého iteračního kroku „k" všechny nové hodnoty výstupů každé buňky $B_{n,k}$ jsou vypočítávány z předchozí hodnoty výstupu první okolní buňky $B_{n+an,k-1}$ (adresu buňky A vypočítáme z n + a_n), z předchozí hodnoty výstupu druhé okolní buňky $B_{n+bn,k-1}$ (adresu buňky B vypočítáme z n + b_n) a z předchozí hodnoty výstupu buňky $B_{n,k-1}$ samotné. Všechny tyto tři binární hodnoty tvoří adresu v tabulce (0 … 7) která určuje novou výstupní hodnotu ($f_{n,0}$ … $f_{n,7}$) buňky $B_{n,k}$ z 8mi-bitové tabulky F_n [19].

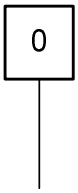
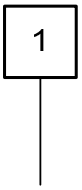
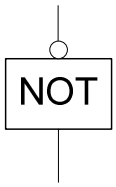
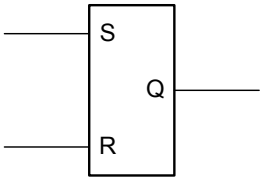
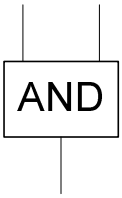
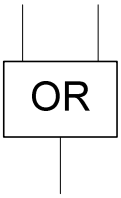
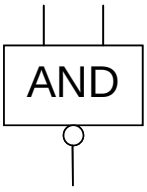
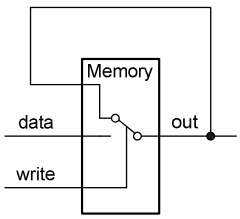
Tab. 8.2.1 8mi-bitová tabulka pro převod vstupních hodnot na výstupní hodnotu y_n

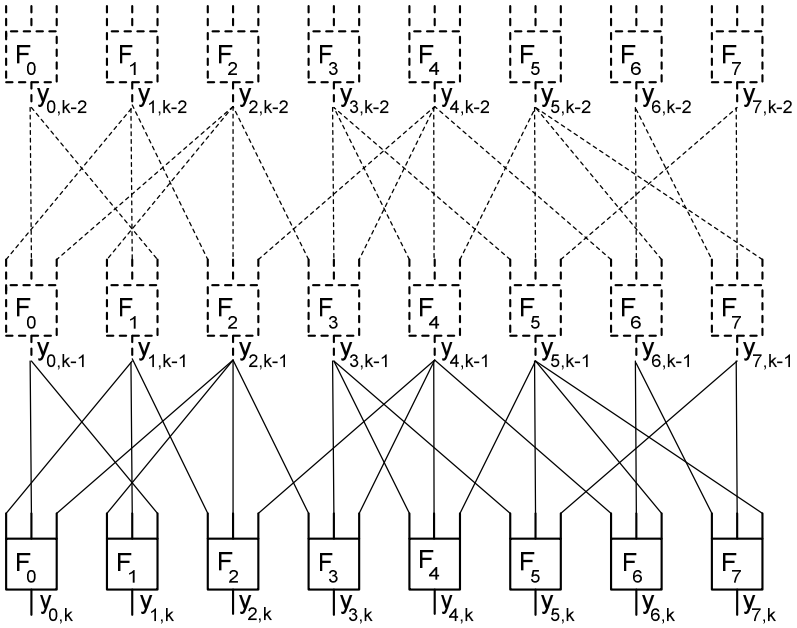
index	Inputs			Output
	$y_{n,k-1}$	$y_{n+bn,k-1}$	$y_{n+an,k-1}$	$y_{n,k}$
0	0	0	0	$f_{n,0}$
1	0	0	1	$f_{n,1}$
2	0	1	0	$f_{n,2}$
3	0	1	1	$f_{n,3}$
4	1	0	0	$f_{n,4}$
5	1	0	1	$f_{n,5}$
6	1	1	0	$f_{n,6}$
7	1	1	1	$f_{n,7}$

Funkce F_n popisuje pomocí svých osmi bitů logickou operaci, která se provádí se vstupy $y_{n+an,k-1}$, $y_{n+bn,k-1}$ a $y_{n,k-1}$.

Příklady několika takových funkcí a logických operací, které se k nim vztahují, jsou uvedeny v tabulce **Tab. 8.2.2** V této tabulce $y_{a,k-1} = y_{n+an,k-1}$ a $y_{b,k-1} = y_{n+bn,k-1}$. Vyčerpávající seznam všech F_n funkcí s číselnou hodnotou 0 – 255 a jejich přidružených logických operací je uveden v příloze A.

Tab. 8.2.2 Příklady 8mi-bitových F_n funkcí a logických operací které reprezentují.

 $(0)_{10}$ $y_{n,k} = 0$	 $(255)_{10}$ $y_{n,k} = 1$	 $(85)_{10}$ nebo $(51)_{10}$ $y_{n,k} = \text{NOT}(y_{a,k-1})$ $y_{n,k} = \text{NOT}(y_{b,k-1})$	 $(178)_{10}$ $S = y_{a,k-1}$, $R = y_{b,k-1}$ $y_{n,k} = Q$
 $(136)_{10}$ $y_{n,k} =$ $\text{AND}(y_{a,k-1} , y_{b,k-1})$	 $(238)_{10}$ $y_{n,k} =$ $\text{OR}(y_{a,k-1} , y_{b,k-1})$	 $(119)_{10}$ $y_{n,k} =$ $\text{NAND}(y_{a,k-1} , y_{b,k-1})$	 $(184)_{10}$ $data = y_{a,k-1}$, $write = y_{b,k-1}$, $y_{n,k} = out$



Obr. 8.2.1 Znázornění struktury nalezené implementace

Jakmile je nalezena hledaná implementace, která odpovídá zadané specifikaci, nastavení všech buněk v implementaci lze znázornit určitou strukturou. Struktura implementace je tvořena díky nastavení relativních adres a_n a b_n každé buňky a jejich funkcí F_n. Příklad jak by mohla vypadat struktura implementace je znázorněn na obrázku **Obr. 8.2.1**

Tab. 8.2.3 Přehled formátu jak jsou ukládány buňky vektoru v paměti programové realizace párovacího systému GenAlg (viz. kapitola Párovací systém)

B: Adresa - nižší část			**Tabulka výstupů nižší část**				**Předchozí y**	**B: Adresa – vyšší část**							
B2	B1	B0	F3	F2	F1	F0	y_	B10	B9	B8	B7	B6	B5	B4	B3

A: Adresa - nižší část			**Tabulka výstupů vyšší část**				**Aktuální y**	**A: Adresa – vyšší část**							
A2	A1	A0	F7	F6	F5	F4	y	A10	A9	A8	A7	A6	A5	A4	A3

Formát dat představený tabulkou **Tab. 8.2.3** se u momentálně realizovaného párovacího programu opakuje pro všechny buňky vektoru na adresách buněk 0 až n_{max} [20].

8.3 Matematický popis celulárního procesoru logických funkcí

Stavová matice Y_k celulárního procesoru logických funkcí obsahující aktuální binární hodnoty všech buněk v kroku „k“ [15]:

$$Y_k = \begin{bmatrix} y_{0,k} & y_{1,k} & y_{2,k} & \cdots & y_{n\max,k} \end{bmatrix}, \qquad y_{n,k} \in \{(0)_2, (1)_2\} \tag{8.3.1}$$

Matice F obsahující logické funkce všech buněk vyjádřené pravdivostními tabulkami v matici F:

$$F = \begin{bmatrix} F_0 & F_1 & F_2 & \cdots & F_{n\max} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} f_{0,0} \\ f_{0,1} \\ f_{0,2} \\ f_{0,3} \\ f_{0,4} \\ f_{0,5} \\ f_{0,6} \\ f_{0,7} \end{bmatrix} & \begin{bmatrix} f_{1,0} \\ f_{1,1} \\ f_{1,2} \\ f_{1,3} \\ f_{1,4} \\ f_{1,5} \\ f_{1,6} \\ f_{1,7} \end{bmatrix} & \begin{bmatrix} f_{2,0} \\ f_{2,1} \\ f_{2,2} \\ f_{2,3} \\ f_{2,4} \\ f_{2,5} \\ f_{2,6} \\ f_{2,7} \end{bmatrix} & \cdots & \begin{bmatrix} f_{n\max,0} \\ f_{n\max,1} \\ f_{n\max,2} \\ f_{n\max,3} \\ f_{n\max,4} \\ f_{n\max,5} \\ f_{n\max,6} \\ f_{n\max,7} \end{bmatrix} \end{bmatrix}, \quad f_{n,i} \in \{(0)_2, (1)_2\} \tag{8.3.2}$$

Čtvercová matice V obsahující vazby na okolní buňky zakódované v koeficientech uvnitř matice, sloužící pro sběr výstupů z okolních buněk:

$$V = \begin{bmatrix} 2^2 & v_{1,0} & v_{2,0} & \cdots & v_{n\max,0} \\ v_{0,1} & 2^2 & v_{2,1} & \cdots & v_{n\max,1} \\ v_{0,2} & v_{1,2} & 2^2 & \cdots & v_{n\max,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{0,n\max} & v_{1,n\max} & v_{2,n\max} & \cdots & 2^2 \end{bmatrix}, \quad \text{pro } v_{n,j} \text{ platí:} \tag{8.3.3}$$

- Buňka na pozici n přijímá výstup z jiné buňky na pozici n_a $\Rightarrow v_{n,n_a} = 2^0$, (8.3.4)
- buňka na pozici n přijímá výstup z jiné buňky na pozici n_b $\Rightarrow v_{n,n_b} = 2^1$, (8.3.5)
- buňka nemá žádnou další vazbu (maximum jsou 2 vazby, minimum je 0 vazeb) $\Rightarrow v_{n,j} = 0$. (8.3.6)

Matice P_k obsahující vypočtené indexy pro výběr nové aktuální binární hodnoty všech buněk z matice logických funkcí F pravdivostních tabulek:

$$P_k = \begin{bmatrix} i_{0,k} & i_{1,k} & i_{2,k} & \cdots & i_{n\max,k} \end{bmatrix}, \quad i_{n,k} \in \{(0)_{10}, (1)_{10}, (2)_{10}, (3)_{10}, (4)_{10}, (5)_{10}, (6)_{10}, (7)_{10}\} \tag{8.3.7}$$

Transformační matice $T_{n,k}(i)$ obsahující výpočet koeficientů pro převod indexů $i_{n,k}$ z desítkové soustavy (0,1,2, … ,7) na konkrétní vybranou hodnotu z pravdivostní tabulky:

$$T_{n,k}(i) = \begin{bmatrix} \frac{(i-1)(i-2)(i-3)(i-4)(i-5)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-2)(i-3)(i-4)(i-5)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-3)(i-4)(i-5)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-2)(i-4)(i-5)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-2)(i-3)(i-5)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-2)(i-3)(i-4)(i-6)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-2)(i-3)(i-4)(i-5)(i-7)}{i!\cdot(7-i)!}(-1)^{i+1} \\ \frac{i(i-1)(i-2)(i-3)(i-4)(i-5)(i-6)}{i!\cdot(7-i)!}(-1)^{i+1} \end{bmatrix}, \quad \text{nebo } T_{n,k}(i_{n,k}) = \begin{bmatrix} 0^{i_{n,k}} \\ 0^{|i_{n,k}-1|} \\ 0^{|i_{n,k}-2|} \\ 0^{|i_{n,k}-3|} \\ 0^{|i_{n,k}-4|} \\ 0^{|i_{n,k}-5|} \\ 0^{|i_{n,k}-6|} \\ 0^{|i_{n,k}-7|} \end{bmatrix} \tag{8.3.8}$$

Všechny výsledné matice $T_{n,k}$ lze sdružit transponovaně do celkové matice

$$T_k = \begin{bmatrix} T_{0,k}{}^T & T_{1,k}{}^T & T_{2,k}{}^T & \cdots & T_{n\max,k}{}^T \end{bmatrix} \tag{8.3.9}$$

k pozdějšímu použití pro skalární součin matic.

Průběh výpočtu přechodu z předchozího stavu celulárního logického procesoru Y_k do nového stavu Y_{k+1}:

- Pro výpočet všech indexů (které obsahuje matice P_k) výběru nového výstupu Y_{k+1} z pravdivostních tabulek P_k v kroku „k“ se provede maticová operace násobení dvou matic: $P_k = Y_k V$, (8.3.10)
- pro každý ze vzniklých indexů $i_{n,k}$ matice P_k se vypočítá transformační matice $T_{n,k}(i_{n,k})$ a sdruží se jako transponované prvky $T_{n,k}{}^T$ v matici T_k, - viz (8.3.8) a (8.3.9).
- Dle vypočítaných indexů $i_{n,k}$ matice převedených na transformační matice v matici T_k s pomocí skalárního součinu matic $T_{n,k}$ s pravdivostními tabulkami logických funkcí F_n vyjádřených v matici F se provede výběr nových hodnot $y_{n,k+1}$:

$$Y_{k+1} : y_{n,k+1} = T_{n,k}{}^T \cdot F_n \tag{8.3.11}$$

8.4 Návrh implementace bez Párovacího systému

Stejně tak jako se hledají implementace splňující popis nějakého chování ve specifikaci pomocí Párovacího systému metodami genetického programování, je u některých specifikací možné implementaci navrhnout i ručně. Není to původním záměrem, aby se takto navrhovala implementace bez použití Párovacího systému, ale spíše jsou zde tyto příklady uvedeny pro snadnější pochopení jak implementace fungují a aby bylo zřejmé, že je lze občas navrhnout i ručně.

a) Prvním jednoduchým příkladem je návrh 3-bitového binárního čítače.
V tomto příkladu se použijí 3 buňky celulárního procesoru logických funkcí, které jsou propojeny jedna za druhou. Implementace nemá žádné vstupy, má pouze 3 výstupy, kterými jsou bity čítače y_0, y_1 a y_2.
Buňky s čísly 0, 1 a 2 jsou zde použity pro implementaci; ostatní buňky nejsou použity.
Na obrázku **Obr. 8.4.1** je zobrazeno nastavení implementace realizované těmito třemi buňkami [$B_0[a_0, b_0, F_0]$, $B_1[a_1, b_1, F_1]$, $B_2[a_2, b_2, F_2]$]
= [$B_0[0, 0, 15]$, $B_1[-1, -1, 120]$, $B_2[-1, -2, 120]$]:

- Logická funkce $F_0 = 15$ odpovídá $y_{0,k} = NOT(y_{0+0,k-1})$,
- Logická funkce $F_1 = 120$ odpovídá $y_{1,k} = XOR(AND(y_{1-1,k-1}, y_{1-1,k-1}), y_{1,k-1})$,
- Logická funkce $F_2 = 120$ odpovídá $y_{2,k} = XOR(AND(y_{2-1,k-1}, y_{2-2,k-1}), y_{2,k-1})$.

address	y0	ra	rb	b0	b1	b2	b3	b4	b5	b6	b7
0	0	0	0	1	1	1	1	0	0	0	0
1	0	-1	-1	0	0	0	1	1	1	1	0
2	0	-1	-2	0	0	0	1	1	1	1	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0

Obr. 8.4.1 Implementace 3-bitového binárního čítače nalezená ručně.

Následující obrázek **Obr. 8.4.2** znázorňuje několik vypočítaných kroků při provádění ručně navržené implementace 3-bitového čítače interpretem.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	STEPNR	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	5	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	6	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	7	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	9	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	10	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	11	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	12	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	14	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Obr. 8.4.2 Kroky 0 – 15 interpretu implementace binárního čítače.

b) Druhým příkladem je návrh 8-bitového posuvného registru.
Vstupem do posuvného registru je třetí bit binárního čítače z předchozího příkladu. V tomto příkladu se použije 11 buněk celulárního procesoru logických funkcí, které jsou propojeny jedna za druhou. Buňka č.3 není použita, pouze opticky odděluje čítač a posuvný registr.
Implementace nemá žádné vstupy, má pouze výstupy z čítače a z posuvného registru kterými jsou bity čítače y_0, y_1 a y_2 a bity posuvného registru y_4, y_5, y_6, y_7, y_8, y_9, y_{10} a y_{11}.

Na obrázku **Obr. 8.4.3** je zobrazeno nastavení implementace realizované těmito buňkami [$B_0[a_0, b_0, F_0]$, $B_1[a_1, b_1, F_1]$, $B_2[a_2, b_2, F_2]$, $B_4[a_4, b_4, F_4]$, $B_5[a_5, b_5, F_5]$, $B_6[a_6, b_6, F_6]$, $B_7[a_7, b_7, F_7]$, … $B_{11}[a_{11}, b_{11}, F_{11}]$]
= [$B_0[0, 0, 15]$, $B_1[-1, -1, 120]$, $B_2[-1, -2, 120]$, $B_4[-2, -2, 232]$, $B_5[-1, -1, 232]$, $B_6[-1, -1, 232]$, $B_7[-1, -1, 232]$, … $B_{11}[-1, -1, 232]$]:

- Logická funkce $F_0 = 15$ odpovídá $y_{0,k} = \text{NOT}(y_{0+0,k-1})$,
- Logická funkce $F_1 = 120$ odpovídá $y_{1,k} = \text{XOR}(\text{AND}(y_{1-1,k-1}, y_{1-1,k-1}), y_{1,k-1})$,
- Logická funkce $F_2 = 120$ odpovídá $y_{2,k} = \text{XOR}(\text{AND}(y_{2-1,k-1}, y_{2-2,k-1}), y_{2,k-1})$.
- Logická funkce F_4 … $F_{11} = 232$: $y_{4,k}$ … $y_{11,k}$ = přepínač řízený $y_{n,k-1}$:

  IF($y_{n,k-1} = 0$) : $y_n = \text{AND}(y_{n+an,k-1}, y_{n+bn,k-1})$; ELSE $y_n = \text{OR}(y_{n+an,k-1}, y_{n+bn,k-1})$.

address	y0	ra	rb	b0	b1	b2	b3	b4	b5	b6	b7
0	0	0	0	1	1	1	1	0	0	0	0
1	0	-1	-1	0	0	0	1	1	1	1	0
2	0	-1	-2	0	0	0	1	1	1	1	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	-2	-2	0	0	0	1	0	1	1	1
5	0	-1	-1	0	0	0	1	0	1	1	1
6	0	-1	-1	0	0	0	1	0	1	1	1
7	0	-1	-1	0	0	0	1	0	1	1	1
8	0	-1	-1	0	0	0	1	0	1	1	1
9	0	-1	-1	0	0	0	1	0	1	1	1
10	0	-1	-1	0	0	0	1	0	1	1	1
11	0	-1	-1	0	0	0	1	0	1	1	1
12	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0

Obr. 8.4.3 Implementace 3-bitového čítače a 8-bitového shift registru.

Následující obrázek **Obr. 8.4.4** znázorňuje několik vypočítaných kroků při provádění ručně navržené implementace 3-bitového čítače a 8-bitového shift registru interpretem.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
STEPNR	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
6	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
7	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
9	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
10	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
11	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0
12	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0
13	1	0	1	0	1	0	0	0	0	1	1	1	0	0	0	0
14	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0
15	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	0

Obr. 8.4.4 Kroky 0 – 15 interpretu implementace čítače + shift registru.

c) Třetím a posledním příkladem je návrh 8-bitového posuvného registru, který má zabudovanou zastavovací funkci.
V případě, že nejvyšší bit tohoto posuvného registru bude nastaven na 1, dojde k zastavení posuvu a registr už potom ukazuje stále stejné hodnoty jako v minulém cyklu. Vstupem do posuvného registru je 3. bit binárního čítače z předchozích příkladů. V tomto příkladu se použije také 11 buněk celulárního procesoru logických funkcí, které jsou propojeny jedna za druhou. Buňka č. 3 není použita, pouze opticky odděluje čítač a posuvný registr.
Implementace nemá žádné vstupy, má pouze výstupy z čítače a z posuvného registru kterými jsou bity čítače y_0, y_1 a y_2 a bity posuvného registru y_4, y_5, y_6, y_7, y_8, y_9, y_{10} a y_{11}.

Na obrázku **Obr. 8.4.5** je zobrazeno nastavení implementace realizované těmito buňkami
[$B_0[a_0, b_0, F_0]$, $B_1[a_1, b_1, F_1]$, $B_2[a_2, b_2, F_2]$, $B_4[a_4, b_4, F_4]$, $B_5[a_5, b_5, F_5]$, $B_6[a_6, b_6, F_6]$, $B_7[a_7, b_7, F_7]$, … $B_{11}[a_{11}, b_{11}, F_{11}]$]
= [$B_0[0, 0, 15]$, $B_1[-1, -1, 120]$, $B_2[-1, -2, 120]$, $B_4[-2, 7, 226]$, $B_5[-1, 6, 226]$, $B_6[-1, 5, 226]$, $B_7[-1, 4, 226]$, … $B_{11}[-1, 0, 226]$]:

- Logická funkce $F_0 = 15$ odpovídá $y_{0,k} = NOT(y_{0+0,k-1})$,
- Logická funkce $F_1 = 120$ odpovídá $y_{1,k} = XOR(AND(y_{1-1,k-1}, y_{1-1,k-1}), y_{1,k-1})$,
- Logická funkce $F_2 = 120$ odpovídá $y_{2,k} = XOR(AND(y_{2-1,k-1}, y_{2-2,k-1}), y_{2,k-1})$.
- Logická funkce F_4 … $F_{11} = 226$: $y_{4,k}$ … $y_{11,k}$ = Q výstup z D flip-flopu
  kde WRITE = $NOT(y_{n+bn,k-1})$; DATA = $y_{n+an,k-1}$.

address	y0	ra	rb	b0	b1	b2	b3	b4	b5	b6	b7
0	0	0	0	1	1	1	1	0	0	0	0
1	0	-1	-1	0	0	0	1	1	1	1	0
2	0	-1	-2	0	0	0	1	1	1	1	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	-2	7	0	1	0	0	0	1	1	1
5	0	-1	6	0	1	0	0	0	1	1	1
6	0	-1	5	0	1	0	0	0	1	1	1
7	0	-1	4	0	1	0	0	0	1	1	1
8	0	-1	3	0	1	0	0	0	1	1	1
9	0	-1	2	0	1	0	0	0	1	1	1
10	0	-1	1	0	1	0	0	0	1	1	1
11	0	-1	0	0	1	0	0	0	1	1	1
12	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0

Obr. 8.4.5 Implementace 3-bitového čítače a 8-bitového shift registru se zastavovací funkcí.

Následující obrázek **Obr. 8.4.6** znázorňuje několik vypočítaných kroků při provádění ručně navržené implementace 3-bitového čítače a 8-bitového shift registru se zastavovací funkcí.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	STEPNR	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	5	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
	6	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
	7	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
	9	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
	10	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	11	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0
	12	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0
	13	1	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0
	14	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0
	15	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0

Obr. 8.4.6 Kroky 0 – 15 interpretu implementace čítače + shift registru se zastavovací funkcí.

8.5 Kontrola shody implementace se specifikací

Kontrolou shody se v kontextu implementace myslí mechanismus porovnávání při hledání nejvyšší možné shody u nalezených implementací porovnáním všech jejich výstupů při všech ve zdrojových elementárních specifikacích popsaných případech jednotlivých vstupů s výstupy a vstupy uvedenými v tabulkách elementárních zdrojových implementací. Tuto shodu lze vyjádřit číselně, jako počet jednotlivých bitů kdy se výstupy shodují s předlohou. Každá shoda je oceněna přičtením čísla 1 a každá neshoda přičtením čísla 0. Výsledkem je celkový počet shod, který by se během evolučních cyklů měl postupem času dopracovat až na nejvyšší možnou hodnotu. Podle toho jak rychle se během procesu genetického programování shoda blíží cílové maximální hodnotě, pokusí se párovací systém odhadnout, jaká strategie má být právě uplatněna pro další hledání. Podle potřeby by párovací systém měl měnit parametry hledání, jako například bitovou šířku výstupní informace - která se postupně může zvyšovat, když je hledání úspěšné, nebo snižovat pokud trvá moc dlouho. Dále lze měnit pravděpodobnost mutací hodnot popisující implementace binárním vektorem a pravděpodobnostní charakteristiku pro selekci jedinců uvnitř generace. Podle rychlosti hledání shody lze také měnit varianty vnitřního dělení implementací a automatické štěpení vstupních a výstupních dat na kratší bitové šířky pro vícenásobné vyhledávání menších částí jedné implementace, které jsou později opět po nalezení propojeny.

9. Teoretické základy genetického programování

9.1 Historické souvislosti evolučních výpočetních technik

V následujících odstavcích jsou shrnuty historické události, které významným způsobem přispěly k rozvoji evolučních výpočetních technik a genetického programování z nich vycházející:

- 1859

  Charles Darwin poprvé publikoval knihu O vzniku druhů přirozeným výběrem čili zachováním vhodných odrůd v boji o život.

- 1948

  A.M. Turing se v eseji “Inteligentní stroje” poprvé zmínil že“... budoucí výzkum umělé inteligence bude velmi pravděpodobně spojen s prohledáváním...” dále uvedl “...genetické a evoluční hledání bude zaměřeno na nalezení kombinace genů, kde kritériem bude kvalita této kombinace..”

- 1953

  N.A.Barricelli (Princeton):První reálné experimenty s evolučními principy na počítačích, první simulovaná evoluce.

- 1955-1959

  Rechenberg, Schwefel a Bienart vyvynuli na Berlínské univerzitě optimalizační metodu pro návrh převodovek, později zobecněním a precizací Evoluční Strategie se zabýval H.-P. Schwefel v celé řadě prací (např. Evolution and Optimum Seeking - 1995).

- 1960

  Lawrence Fogel: Evoluční Programování. Cílem bylo evolučním postupem odvodit chování konečného automatu ve smyslu schopnosti určit změny prostředí, v němž se automat nachází. Prostředí bylo popsáno jako posloupnost symbolů z konečné abecedy a evoluční algoritmus měl na výstupu automatu předpovědět další očekávaný symbol této posloupnosti. 1992 (Evolving Artificial Intelligence), 1994 (IEEE Transactions on Neural Networks, special issue on Evolutionary Computation) zobecněno synem Davidem Fogelem na numerické optimalizační problémy.

- 1975

  John Holland (Adaptation in Natural and Artificial Systems): Americký teoretický biolog vydává knihu která položila základ genetickým

algoritmům, kde se snaží algoritmicky odpovědět na otázku "...Proč se navzájem liší jedinci patřící k témuž biologickému druhu?..."

- 1989

  David Goldberg (Genetic Algorithms in Search, Optimalization and Machine Learning): Genetické algoritmy systematicky studovány z pohledu technického, snažící se chápat genetické algoritmy jako techniku obecně aplikovatelnou na širokou třídu úloh.

- 1992, 1994

  John Koza (Genetic Programming, Genetic Programming 2): Zakladatel genetického programování.

- 1996

  Zbigniew Michalewicz (Genetic Algorithms + Data Structures = Evolution Programs): Jedním z původců výrazných modifikací klasických genetických algoritmů tak jak je představil J. Holland a rozpracoval D. Goldberg.

9.2 Základní rozdělení optimalizačních metod

U genetického programování a genetických algoritmů se jedná o řešení optimalizačních úloh, kde se provádí prohledávání stavového prostoru, které vede k takovému nastavení parametrů určité funkce, nebo algoritmu, že se jejich výsledek, nebo chování blíží co nejvíce k žádanému optimu, nebo přímo optima dosáhne. V našem případě dosažení optima odpovídá nalezení implementace shodující se se svou zdrojovou specifikací ve všech jejích bodech.

9.3 Evoluční výpočetní techniky

Nadřazená třída, do které genetické programování přísluší, se nazývá „Evoluční výpočetní techniky" (EVT), které se dále dělí na [14] (strana 39):

- **Evoluční strategie -** optimalizační úlohy s mutacemi a křížením,
- **Evoluční programování -** konečné automaty, žádné křížení, 1 potomek,
- **Genetické algoritmy** - kódování chromozomy, posloupnost symbolů,
- **Genetické programování -** automatizované generování programů.

Obvyklé metody prohledávání stavového prostoru jsou založeny na těchto strategiích [11] (strana 118):

- **Dostatečně malý stavový prostor** - lze jej prohledat úplně a beze zbytku a obvykle po uplynutí určité doby a při vynaložení přiměřeného úsilí bezpečně nalezneme žádané optimum „hrubou silou".

- **Rozsáhlé stavové prostory** - řešení nelze najít „hrubou silou“ v rozumném čase a s přiměřeně vynaloženým úsilím. Je nutno prohledávat heuristickými či znalostně řízenými metodami umělé inteligence.

Nasazení evolučních výpočetních technik se využívá, když velikost stavového prostoru hledaného řešení se nachází mezi těmito dvěma extrémy a jsou to zpravidla stochastické algoritmy, jejichž prohledávací schopnost je posílena modelováním genetické dědičnosti a Darwinovským zápasem o přežití [11] (strana 118). Při užití evolučních výpočetních technik je tedy zřejmá inspirace přírodním výběrem, kde jednotlivé druhy během procesu přirozené evoluce bojují o své přežití a o účast v dalším soutěžení s ostatními konkurenčními druhy. Podobnost s přírodní evolucí je tedy zřejmá v tom, že v tomto případě jednotlivá nalezená řešení daného problému také podstupují selekci dle určitých, předem stanovených kritérií. Při porovnání s jinými nalezenými řešení, která splňují tato kritéria více než jiná, nebo která dosáhnou větší shody s hledaným optimem se ta nevhodná vyřazují z dalšího procesu výběru a tak lze po uplynutí jistého počtu evolučních cyklů dospět k hledanému optimu, nebo se mu alespoň s dostatečnou požadovanou přesností přiblížit. Obecný postup hledání optima u EVT [11] (strana 118):

- Tradiční optimalizační metody vycházejí ze vhodného (nebo náhodně zvoleného) počátečního odhadu řešení, které iterativně vylepšují.
- EVT nepracují s jedním kandidátem řešení daného problému, nýbrž s jejich množinami, populacemi jedinců – kandidátů na optimální řešení.
- Evoluce probíhá v diskrétních krocích a vzniká posloupnost populací, zpravidla nazývaná generacemi.
- Každý člen populace (každý jedinec) je reprezentován stejnou datovou strukturou, ale s různým nastavením parametrů a jeho schopnost přiblížit se optimu je ohodnocena kriteriální funkcí - vypočítá se jeho kvalita („fitness“).
- Provede se statistické vyhodnocení kvality všech jedinců, na jehož základě dojde k selekci, obvykle pomocí pravděpodobnostního mechanismu, přičemž u jedinců s vyšší kvalitou je vyšší pravděpodobnost že postoupí do dalšího kola evoluce.
- Výsledek statistického vyhodnocení kvality bývá rovněž použit k rozhodování, zda je nutné dále v evoluci pokračovat, jestli výsledná kvalita jedinců je dostačující a zda hledané optimum bylo dosaženo.
- Na jedince, kteří prošli procesem selekce (ostatní byli odstraněni z populace), jsou uplatňovány operátory změny parametrů. Tyto se poté realizují s určitou pravděpodobností a v různém rozsahu.
- Slučováním dvou, nebo více různých jedinců jsou generováni jedinci noví, kdy kritériem pro jejich kombinování bývá jejich kvalita nebo určité žádané vlastnosti. Tito nový jedinci obvykle doplní dříve vzniklou mezeru v populaci.

Nejrozšířenější jsou tyto vývojové strategie u EVT [11] (strana 120):

- Generační: Úplná náhrada jedné populace populací následující (analogie životního cyklu jednoletých rostlin).

- Postupné: Změny podstupuje jen malá část populace a rodiče koexistují se svými potomky (déle žijící organismy nebo víceleté rostliny).

9.4 Genetické algoritmy - základní informace

Základní terminologie (původem z biologie) [13] (strana 20 – 21):

- Individuum – *fenotyp*,
- Reprezentace individua - *genotyp* (skupina chromozomů),
- Zjedodušení pro GA: Pouze jeden *chromozom* (= celý genotyp),
- *Chromozom* se dělí na jednotlivé lineárně uspořádané geny,
- Genetické algoritmy mají *chromozom* kódován jako posloupnost symbolů,
- Různé stavy jednoho *genu* – *alely*: jsou to binární stavy, decimální stavy, nebo hexadecimální stavy,
- Různé vlastnosti nebo parametry (např. barva, typ zakřivení, délka, apod.),
- Nejjednodušší a nejstarší typ kódování *chromozomu* u GA je binární kódování = sekvence bitů.

V jednoduchém příkladě na **Obr. 9.4.1**, *účelová funkce* je reprezentována počtem bitů 1 (žádané optimum je jedinec s co nejvyšším počtem bitů 1) [13] (strana 21).

Individuum č.	Chromozom	Ohodnocení
1	(1, 0, 1, 0, 1, 1, 0, 0)	4
2	(0, 1, 1, 1, 1, 0, 1, 1)	6
3	(0, 0, 0, 1, 0, 0, 0, 1)	2
4	(1, 1, 0, 0, 1, 1, 0, 0)	4

Obr. 9.4.1 Ohodnocení dle účelové funkce

Vhodnost (*fitness*) [14] (strana 174):

- Obvykle je to přetransformovaná hodnota účelové funkce.
- Používá se pro výběr rodičů. Obvykle je to funkce v níž se hledá maximum:

$$pro \forall ind1, ind2 \in R,\ f(ind1) \leq f(ind2) \Rightarrow F(ind1) \geq F(ind2), \qquad (9.4.1)$$

kde *ind* je jedinec z populace (individuum), *f* účelová funkce, *F* funkce počítající vhodnost.

- Transformace vhodnosti do normalizovaného tvaru:

$$F(ind) = \frac{F_{max} - F_{min}}{f_{min} - f_{max}} f(ind) + \frac{f_{min} F_{min} - f_{max} F_{max}}{f_{min} - f_{max}}, \qquad (9.4.2)$$

kde f_{max} je maximální hodnota účelové funkce, f_{min} je minimální hodnota účelové funkce, F_{max} je maximální hodnota vhodnosti, F_{min} je minimální hodnota vhodnosti, *f(ind)* je hodnota účelové funkce aktivního jedince, *F(ind)* je vhodnost aktivního jedince.

- Vhodnost je v intervalu [0,1], musíme tedy dosadit za F_{max} hodnotu 1 a za F_{min} malé kladné číslo blízké nule (kvůli dělení).
- Po dosazení hodnot do vztahu 9.4.2 získáme:

$$F(ind) = \frac{1}{f_{\min} - f_{\max}}\left[(1-\varepsilon) f(ind) + f_{\min}\varepsilon - f_{\max}\right], \qquad (9.4.3)$$

  kde f_{max} je maximální hodnota účelové funkce, f_{min} je minimální hodnota účelové funkce, *f(ind)* je hodnota účelové funkce aktivního jedince, *F(ind)* je vhodnost aktivního jedince, ε je malé kladné číslo.
- Rovnice přerozděluje hodnoty účelových funkcí lineárně do intervalu [0,1], tedy největší hodnotě účelové funcke (maximu) se přiřadí hodnota 1 a nejmenší hodnotě (minimu) 0, respektivě malé kladné číslo.

Další procesy na kterých je genetický algoritmus založen [13] (strana 21 – 25):

- **Přirozený výběr:** Nejrozšířenější implementací přirozeného výběru je takzvané „Ruletové kolo“. Jedincům s vyšším ohodnocením je na ruletovém kole přiřazena větší výseč a existuje tedy vyšší pravděpodobnost, že při pomyslném roztočení rulety se kulička zastaví na výseči s kvalitnějším individuem. Potom pravděpodobnost, s jakou bude i-tý jedinec vybrán, odpovídá velikosti jeho kruhové výseče.
- **Křížení:** Bod křížení musí být určen náhodným přirozeným číslem z množiny {1, .., d-1}, kde d je délka chromozomu.
- **Mutace:** Operátor mutace s relativně malou pravděpodobností mění hodnotu jednotlivých genů.
- **Nová populace:** Při použití *Generační strategie* pro vytváření nové populace stará populace *P(0)* ztratí jakýkoliv význam, vymře a je nahrazena populací novou *P(1).*

9.5 Genetické algoritmy - shrnutí

Genetický algoritmus je založen na těchto principech:

- Algoritmus pracuje s populací individuí, jež je obvykle inicializována zcela náhodně.
- Každé individuum reprezentuje skrze vhodně zvolené kódování jedno potenciální řešení daného problému.
- Každému individuu je přiřazeno ohodnocení (*fitness*).
- Individua jsou vybírána k další reprodukci náhodně tak, že pravděpodobnost výběru každého individua je úměrná jeho ohodnocení.
- Aplikací různých sexuálních (*křížení*) a asexuálních (*mutace*) genetických operátorů jsou produkována nová individua, tedy nová potenciální řešení daného problému.

9.6 Genetické programování

Základní popis [13] (strana 123):

- Genetické programování lze považovat za rozšíření genetických algoritmů.
- *Chromozomy* zde nejsou řetězce pevné délky, ale hiearchicky strukturované počítačové programy.
- Tyto programy po spuštění mohou představovat potenciální řešení problému.
- Výsledkem evolučního procesu je tedy program, který řeší (nebo přibližně řeší) konkrétní problém.
- Výsledný algoritmus (program) je zapsán speciálním, pro účely genetického programování navrženým jazykem, který je koncipován tak aby bylo možno s jeho stavebními bloky (programovými kroky) jednoduše manipulovat a provádět automatické operace řízené evolučním procesem.

Typy problémů, které je vhodné řešit s pomocí genetických algoritmů jsou přehledně shrnuty v tabulce **Tab. 9.6.1** [12] (strana 129).

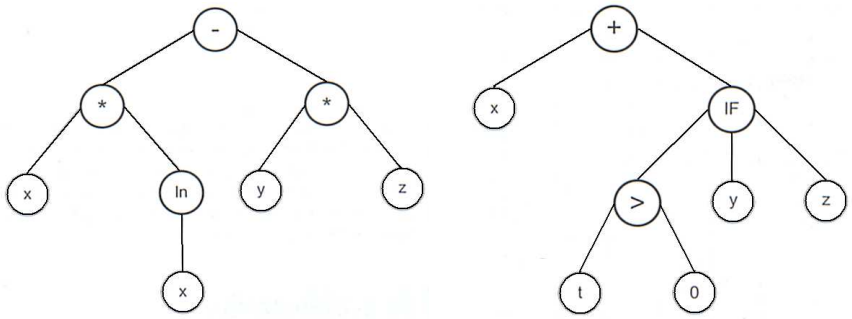
Tab. 9.6.1 Přehled úloh u kterých je vhodné pro řešení použít genetické programování

	Úloha	**Hledaný algoritmus**	**Vstup**	**Výstup**
1.	indukce posloupnosti	analytický předpis	Index	element posloupnosti
2.	symbolická regrese množiny dat	matematický výraz	nezávislé proměnné	závislé proměnné
3.	optimální řízení	řídicí strategie	stavové proměnné	akční veličiny
4.	identifikace a predikce	matematický model systému	nezávislé proměnné	závislé proměnné
5.	klasifikace	rozhodovací strom	hodnoty atributů	přiřazení do třídy
6.	učení se cílenému individuálnímu chování	program popisující chování	vstupy ze senzorů	akce organismu
7.	odvození kolektivního chování	program popisující chování jedince	informace o vztahu jedince ke zbytku kolektivu	akce jedince v kolektivu

Požadavky na použitý jazyk pro genetické programování [13] (strana 124):

- Hierarchická struktura (syntaktické stromy).
- Syntaktická odolnost vůči mutacím a křížení.
- Strom se skládá z neterminálů (funkcí) a terminálů (proměnných a konstant).
- Počet hran vycházejících z každého uzlu, který odpovídá nějaké funkci f_i z množiny F, je určen aritou této funkce (počtem argumentů).

- Množina funkcí F a množina terminálů T musí být definovány tak, že budou splňovat požadavky uzavřenosti a postačitelnosti.
- Uzavřenost: Funkce nesmí způsobit chybu programu, když se objeví nedovolená operace (např. dělení nulou), mezní stavy musí být dostatečně ošetřeny.
- Postačitelnost: Dobré porozumění problému je nutné k použití takové množiny funkcí a terminálů, které ve své kombinaci a struktuře mohou popsat hledané cílové řešení (nelehký požadavek), jinak není možné nalézt cílové řešení.
- Některá řešení vyžadují přítomnost cyklů, zde je nutno je i použít.
- Někdy je vhodné obohatit množinu terminálů o náhodnou konstantu, která se "vypočítá" v okamžiku kdy je použita.
- Jazyk by měl být co nejjednodušší, mít co nejméně prvků množin F a T (rychlost).



Obr. 9.6.1 Dva příklady syntaktických stromů [13] (strana 124, 125)

Generování počáteční populace [13] (strana 127, 128):

- Náhodná selekce funkcí a terminálů z množin F a T, které jsou propojovány do náhodných syntaktických stromů. Začíná operátor, následují jeho operandy.
- Dvě metody omezení velikosti syntaktického stromu: **úplná** a **růstová**
- **Úplná metoda:** Všechny větve mají mít maximální předepsanou délku (hloubku). Terminály se generují až po dosažení této délky.
- **Růstová metoda:** Vybírají se náhodně terminály nebo funkce, ale jakmile je vybrán terminál, větev se považuje již za ukončenou.
- V praxi se doporučuje kombinovat obě metody půl na půl. Polovinu populace generovat úplnou metodou a polovinu populace růstovou metodou. => Rozmanitost
- Je třeba stanovit způsob hodnocení kvality jedinců. Blíží se program hledanému řešení a pokud ano, tak o kolik lépe nebo o kolik hůře než jiné varianty v populaci?

Během průběhu genetického programování se užívají tyto genetické operátory [13] (strana 131):

- **Mutace uzlová:** Nahrazuje neterminální uzel neterminálem se stejným počtem argumentů, nebo terminální uzel jiným terminálem.
- **Mutace vyzvedávající:** Nahradí celý syntaktický strom některým z jeho podstromů.
- **Mutace smršťující:** Nahrazuje náhodně zvolený podstrom jediným terminálem.
- **Permutace:** Prohazování operandů, změna pořadí.
- **Editace:** Zjednodušování stromů, využití známých pravidel, které redukují nadbytečné struktury.
- **Zapouzdření:** Modularita, tvorba podprogramů. Užitečné části stromu jsou zapouzdřeny a je na ně odkazováno i na jiných místech stromu. Současně jsou tyto části chráněny před změnami.
- **Decimace:** Způsob jak zmenšit příliš rozsáhlé populace, které zpomalují evoluční cyklus. Zvýšený tlak na vyřazování jedinců, spouštěný určitou podmínkou (podle velikosti populace, apod.).

10. Aplikace GenAlg

10.1 Aplikace pro realizaci párovacího systému

Párovací systém je ve své podstatě založen na vyhledávání implementací pomocí genetického programování porovnáváním této implementace se svou zdrojovou specifikací tak dlouho, dokud se neshodují. V rámci této práce byl vyvinut program „GenAlg“ jehož úkolem je provádět stejnou operaci jakou by měl v operačním systému nové generace vykonávat párovací systém.

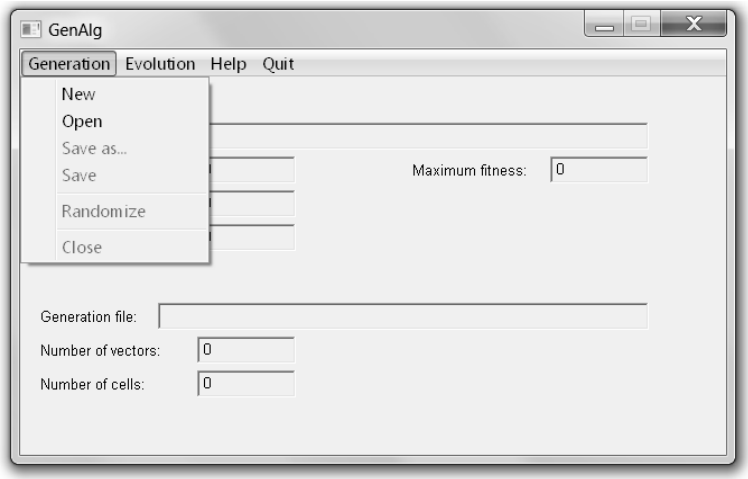
Genetické programování je založeno na mnohonásobném opakování cyklů programového kódu, takže jeho výsledná výkonnost a mnohdy i použitelnost závisí na tom, jak úsporně je kód naprogramován. Čím méně programových instrukcí bude v hlavním cyklu prováděno, tím lépe. Při běhu aplikace se ve velké míře operuje hlavně s elementárními jednotkami informace – jednotlivými bity a spolu s požadavkem na nejvyšší možnou dosažitelnou rychlost a úspornost prováděného programu tedy platí, že nejvhodnějším programovacím jazykem je v tomto případě právě assemblerový kód, který se v praxi používá hlavně na časově a výkonově kritické procesy (což je tento případ). Pro tyto účely byl zvolen vývojový nástroj MASM32 SDK V10 pro CPU x86 v grafickém prostředí Win32API, což zaručuje dobrou kompatibilitu s rodinou 32bitových operačních systémů Microsoft Windows [18]. Současná verze programu GenAlg je však odladěna pro chod v operačním systému Windows 7 (pravděpodobně i pro Windows Vista). Je známo, že ve starším operačním systému Windows XP neběží program spolehlivě. Nekompatibilita s Windows XP byla zjištěna až později a není v současné době smysluplné provést odladění i pro Windows XP. Důvodem je, že samotný běh programu „GenAlg“ a párování implementací se specifikacemi je časově a výkonově náročné. Většina výkonově dostatečných počítačů již novější Windows 7 obsahuje a na starších počítačích by program stejně kvůli pomalému hledání nemělo smysl spouštět. Pro spouštění programu „GenAlg“ by měl být použit počítač, který má procesor výkonově odpovídající alespoň procesoru Intel Core i5. Doporučuje se však i výkonnější procesor, jako například nejnovější procesor Intel Core i7 s architekturou Ivy Bridge (leden 2013). Tento procesor byl rovněž použit pro vývoj programu a pro následné získávání zkušeností při práci na příkladech evolučním procesem nalezených implementací.

10.2 Vzhled a ovládání aplikace

V této kapitole se seznámíme se základním popisem aplikace GenAlg s ohledem na její vzhled a způsob jakým se aplikace používá. Aplikace se spouští spustitelným souborem GenAlg.exe a pro její běh se doporučuje použít operační systém Windows 7, neboť zde byla aplikace i odlaďována a nelze zaručit, že bude bez problémů fungovat i jinde.

Po spuštění aplikace jsou v hlavním menu dostupné tyto položky:

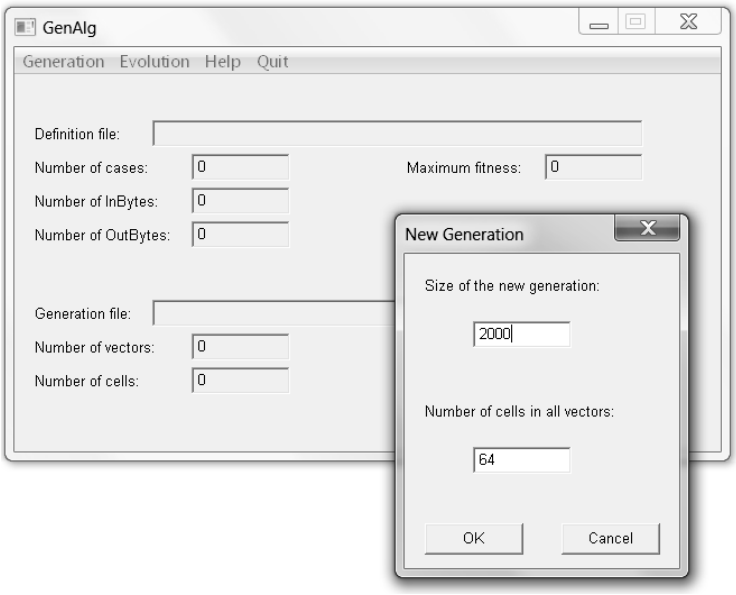
- **Generation** : obsahuje operace které se provádějí s generacemi všech 8mi paralelních evolucí
  - **New** : vytvoř novou sadu 8mi generací
  - **Open** : otevři soubor *.gen s uloženými generacemi
  - **Save as...** : ulož generace do souboru *.gen dle výběru
  - **Save** : ulož generace do aktuálního souboru *.gen
  - **Randomize** : naplň generace náhodným obsahem
  - **Close** : zavři generace a vymaž je z paměti
- **Evolution** : obsahuje operace prováděné s evolucemi
  - **Load Definition** : načti definici/specifikaci hledané funkce *.fdf
  - **Show evolution** : otevře, nebo zobrazí okno s aktuální evolucí
- **Help** : otevře okno se stručným popisem programu
- **Quit** : ukončení aplikace, zavřou se všechna okna



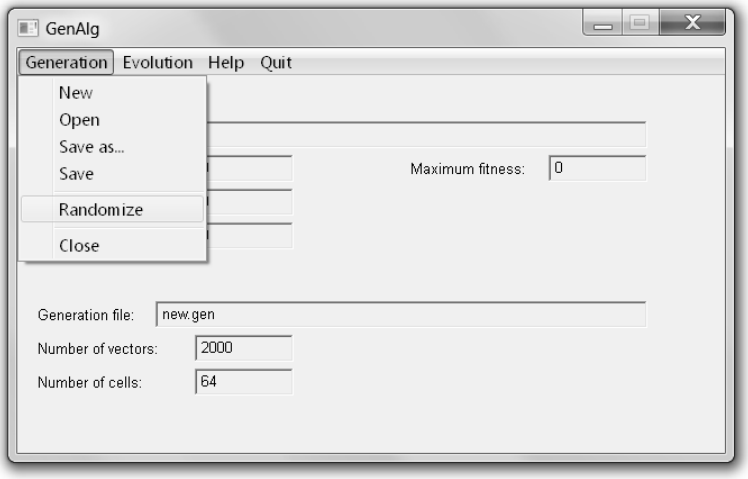
Obr. 10.2.1 Vzhled aplikace po spuštění a menu pro operace s generacemi

Na obrázcích **Obr. 10.2.1 – 10.2.7** jsou zachyceny situace během ovládání aplikace. Po spuštění můžeme vytvořit novou sadu 8mi generací položkou „New“ z nabídky „Generation“, nebo otevřít generace již existující v souborech typu *.gen

pomocí položky „Open“. Po nastavení parametrů generací výběrem počtu implementací (vektorů) a počtu buněk v každé implementaci (přiměřených výsledků dosahováno s nastavením 2000 vektorů a 64 buněk) se doporučuje použít funkci „Randomize“, která naplní všechny buňky všech implementací u všech 8mi generací náhodným obsahem, ze kterého se během evoluce bude vycházet.

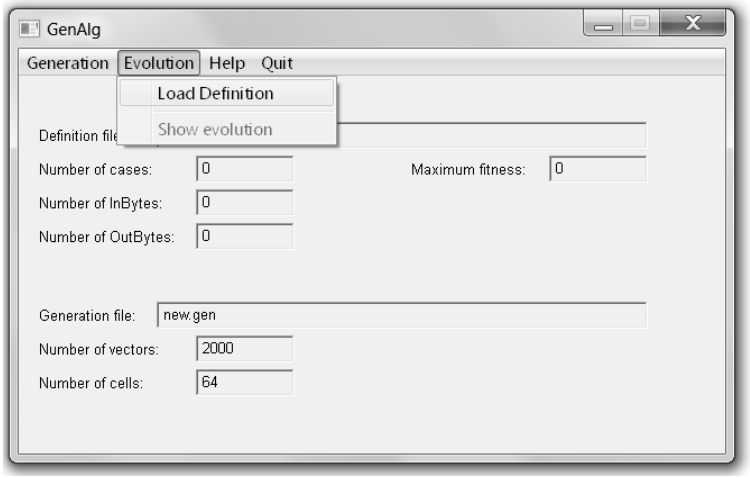


Obr. 10.2.2 Vytvoření nové generace po výběru „New“ z menu „Generation“

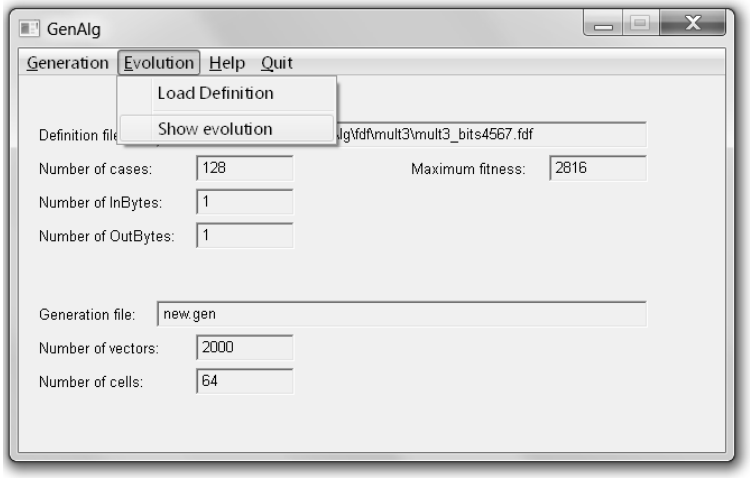


Obr. 10.2.3 Nová generace byla vytvořena s vyobrazenými parametry

Po přípravě generací nahrajeme do aplikace *.fdf soubor, který musíme mít předem připravený a který obsahuje definici (specifikaci) funkce (implementace), použité k hledání algoritmu evolučním procesem. Později nalezená implementace musí tuto definici splňovat. Po otevření definice se vypočítá nejvyšší možná shoda kterou lze během evolučního procesu dosáhnout (Maximum fitness) a naplní se i údaje o počtu vstupních a výstupních bajtů a počtu všech případů v definici (Number of cases).



Obr. 10.2.4 Ukázka nabídky z menu „Evolution“ před nahráním specifikace

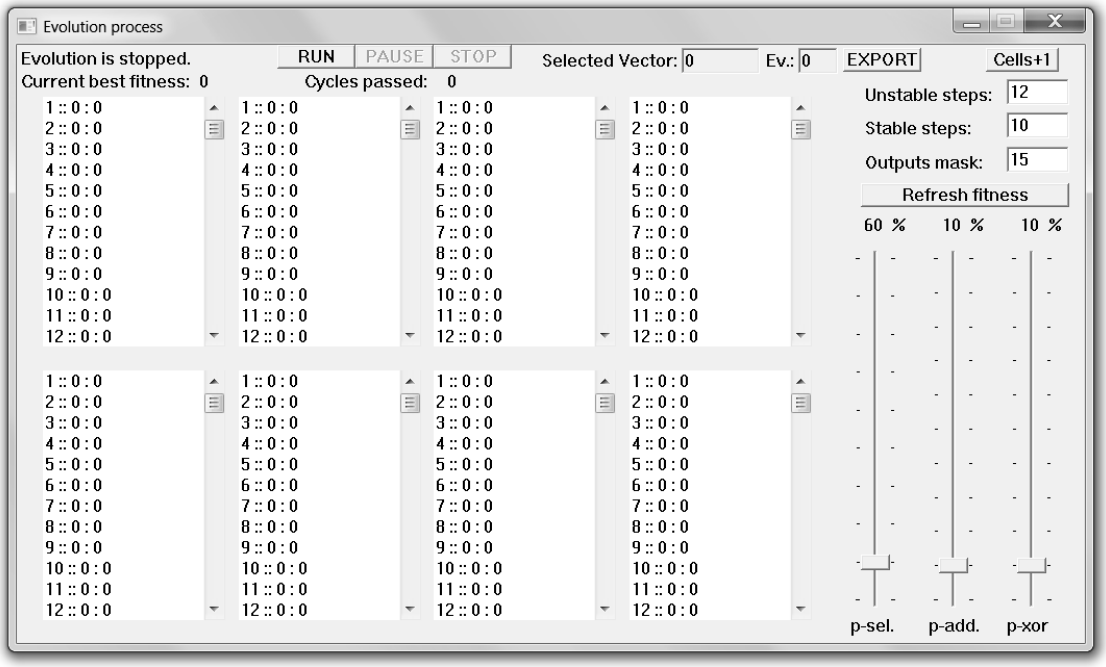


Obr. 10.2.5 Ukázka nabídky z menu „Evolution“ po nahrání specifikace

Nyní je již možné otevřít okno evolučního procesu a po jeho otevření se zobrazí přehled všech 8mi generací a u každé z nich je uveden seznam všech jejich aktuálních implementací. U téhož okna jsou dostupné další ovládací prvky s kterými je možné spouštět (RUN), pozastavit (PAUSE), nebo zcela zastavit (STOP) evoluční děj.

Pokud je evoluce zastavena, vhodnou manipulací s posuvnými ovládacími prvky vpravo dole lze měnit parametry pro následně spouštěnou evoluci:

- p-sel. : Lze nastavit pravděpodobnost přežití jedinců během evolučních cyklů. Touto hodnotou je násobena selekční charakteristika.
- p-add. : Nastavuje pravděpodobnost výskytu aditivní mutace v každé buňce během reprodukce. Jde o náhodné přičtení jednoho bitu.
- p-xor. : Nastavuje pravděpodobnost výskytu xorové mutace v každé buňce během reprodukce. Jde o náhodný xor jedním bitem.



Obr. 10.2.6 Ukázka okna evoluce potom co byly nastaveny všechny parametry

Tlačítkem „Cells+1“ vpravo nahoře lze zvýšit počet buněk ve všech implementacích, ale jen pokud je evoluční proces momentálně zastaven tlačítkem „STOP“. Po použití tlačítka „Cells+1“ bude na náhodnou pozici uvnitř všech implementací vložena buňka s náhodným obsahem. Doporučuje se toto tlačítko používat s rozvahou, protože v současné verzi aplikace je tento proces nevratný a tuto přidanou buňku již nelze později odstranit, ani není možné jiným způsobem počet buněk snižovat. Rovněž se nedoporučuje přidávat v jednom okamžiku více než jednu buňku, protože při každém použití dochází k velkým změnám v aktuálním rozložení buněk všech implementací a v obsahu původně v seznamu blízkých a velmi podobných implementací. Po dalším spuštění evolučního procesu je vhodné nechat evoluci chvíli běžet, než přikročíme k přidání další buňky, aby se mezitím stihlo rozložení vůči sobě podobných implementací opět ustálit. Do jisté míry vůči sobě podobné implementace jsou důležité proto, aby byla možná jejich reprodukce, jinak dochází k degradaci a vznikají produkty s horší kvalitou než byly jejich rodiče. Tato situace může přechodně po přidání buňky nastat, ale obvykle se generace brzy opět z tohoto „šoku“ vzpamatuje. Přidání buňky lze také provést v případě pokud máme pocit, že evoluce delší dobu stagnuje a že se dále nevyvíjí. Docílíme tak postrčení k lepším výsledkům i za cenu přechodného zhoršení, které ale většinou brzy vymizí.

Další parametry které výrazně ovlivňují průběh běžící evoluce jsou počty kroků předpokládaného přechodného děje (Unstable steps) a ustáleného stavu (Stable steps). Tyto kroky se vztahují k ohodnocování kvality jednotlivých implementací, kdy se testuje každá z definic uvedených v souboru *.fdf a vypočítává se počet shod aktuálních výstupů s očekávanými výstupy. Během přechodného děje se nepočítají shody (až na poslední krok této fáze), ale až během kroků ustáleného stavu. Nejnižší počet kroků který lze nastavit u přechodného děje je číslo 2, kdy se potom ohodnocují vlastně všechny kroky, kromě prvního (a druhého po něm) kroku, u

kterého jsou výstupy všech buněk nastaveny na nulu a který je výchozím krokem po resetu všech buněk.

Další parametr který lze nastavit, je maska binárních výstupů (Output mask), která uvádí informaci o tom, jaké binární výstupy se mají zahrnout pro aktuální výpočet shody. Pro aktivaci změny libovolného parametru je nutné použít tlačítko „Refresh fitness“, které slouží pro výpočet nového čísla nejvyšší možné shody (Maximum fitness) a které automaticky upraví podle nového nastavení i parametry v dříve nahrané definici *.fdf, ale pouze v definici nahrané v paměti (se souborem na disku se nemanipuluje). Po změně parametrů lze opět spustit evoluční proces tlačítkem „RUN“ (nebo „PAUSE“ pokud byl proces jen pozastaven), který už používá nové nastavení.

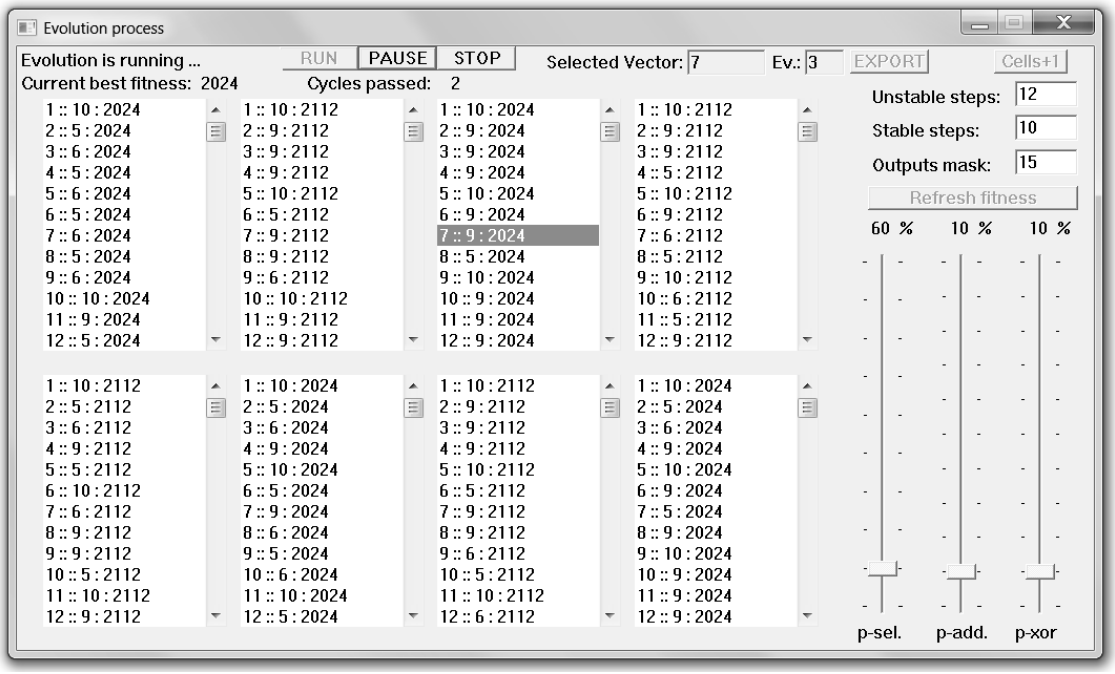
Zde následují doporučené hodnoty pro nastavení masky binárních výstupů:

- 1 : používá se pouze první binární výstup „y_0“ (poslední buňka v implementaci)
- 3 : binární výstupy „y_0 a y_1“ (poslední 2 buňky v implementaci)
- 7 : binární výstupy „y_0, y_1 a y_2“ (poslední 3 buňky v implementaci)
- 15 : binární výstupy „y_0, y_1, y_2 a y_3“ (poslední 4 buňky v implementaci)
- 31 : binární výstupy „y_0, y_1, y_2, y_3 a y_4“ (posledních 5 buňek v implementaci)
- 63, 127, 255, 511, 1023, 2047, 4095, ... 65535 : Obdobným způsobem lze přidávat další výstupy, musí být ale splněna podmínka že s nimi původní definice počítala - nelze přidat více výstupů než původně v definici existovalo. Výstupy musí jít také po sobě, aplikace nepočítá s vynecháním výstupu uvnitř mezi několika jinými.

Pokud je evoluce zastavena, nebo pozastavena, je možné dvojitým kliknutím vybírat libovolné implementace z libovolné z 8mi generací což se projeví změnou čísel u „Selected Vector“ a „Ev.“. Následně tlačítkem „Export“ vygenerujeme soubor out.txt, který by měl být k nalezení ve stejném adresáři, odkud jsme naposledy prováděli nahrání souborů *.gen nebo souboru *.fdf. Pozor, pokud se nacházíte na nějakém datovém médii, které není určeno pro zápis (např. CD-ROM), soubor out.txt vygenerovat nelze. Soubor *.fdf se musí pak zkopírovat například na pevný disk a odtud musí být znovu otevřen aby bylo možné do stejného adresáře generovat soubory out.txt.

Soubor out.txt obsahuje veškeré informace o vybrané implementaci a je možné jej otevřít v libovolném textovém editoru. Na začátku souboru je vypsán obsah všech buněk celé implementace, potom následují všechny definice a jejich související prováděné kroky včetně přechodných a těch ustálených z kterých se vypočítává vhodnost (fitness). Lze zkoumat výstupy všech buněk ve všech krocích a po pravé straně nalezneme vyhodnocení shody pomocí znaku „.“ - shoda a znaku „X“ - neshoda. Pozor, statistika shod a neshod je vyobrazena zrcadlově vůči výstupním buňkám v implementaci, takže první výstupní buňka y_0 (v implementaci ta poslední uplně napravo) je vyhodnocena hned sousedním znakem. U výstupů postupujeme zprava doleva a u jejich statistiky zleva doprava. Na úplném konci souboru je zapsán

aktuální součet všech shod a hned vedle něj za znakem „/“ je uveden nejvyšší možný dosažitelný součet všech shod. Pokud se obě čísla shodují, cílová implementace byla nalezena a evoluční proces je možné ukončit.



Obr. 10.2.7 Ukázka okna evoluce která byla spuštěna a momentálně běží

Výpisy všech implementací u jednotlivých generací obsahují informaci o jejich pořadovém čísle, dále za „::“ následuje číselný kód 10 (rodič1), 9 (rodič2 nebo nepoužit pro reprodukci), 5 (potomek1), 6 (potomek2) nebo 0 (vyřazen z evoluce) upřesňující jejich poslední status který získali při selekci, nebo při reprodukci. Nakonec za „:“ je uvedeno jejich vypočítané číslo shody, které lze brát v potaz jen u jedinců s kódem 10 nebo 9 protože u ostatních toto číslo už neplatí a musí se vypočítat nové (Výpočet probíhá na začátku cyklu, po reprodukci – kde se právě nacházíme, se už znovu nepočítá).

Na závěr popisu je dobré se zmínit o informaci u „Current best fitness“ a „Cycles passed“. První informace uvádí aktuálně nejvyšší dosaženou shodu, ale pozor, toto číslo je vzato pouze z první generace, v jiné z ostatních 7mi generací může být v tu chvíli nalezena ještě vyšší shoda. „Cycles passed“ uvádí kolik evolučních cyklů uběhlo od posledního stištěného tlačítka „RUN“ a smaže se na nulu kdykoliv se zmáčkne tlačítko „STOP“ a potom nový „RUN“. Pozor toto číslo se vztahuje opět pouze k první evoluci, neboť všechny paralelně běžící evoluce běží asynchronně a některé z nich mohou být rychlejší, jiné pomalejší – podle aktuální náročnosti a vytíženosti procesu. Při používání tlačítek pro zastavení, nebo pozastavení běhu evoluce je nutné chvíli počkat než se zastaví všech 8 paralelně běžicích evolucí, které běži zcela nezávisle a můžeme je tedy zastihnout v libovolné z jejich fází. Dokud se všechny evoluční procesy nezastaví, nelze provádět žádnou další akci, což je poznat podle toho, že jsou tlačítka nedostupná pro jakoukoliv manipulaci.

Po dosažení počtu cyklů rovno 207, nebo 353 a jejich dalších násobků (čísla jsou v aplikaci nastavena napevno a jejich velikost není blíže zdůvodněna, byla jen zvolena tak aby uběhl „rozumný“ počet cyklů) se automaticky proces evoluce pozastaví na krátkou chvíli a provede promíchání jedinců mezi paralelními generacemi a nakonec vypíše sám aktuální informace o všech implementacích. Je zde potom k vidění jak byly generace mezi sebou promíchány (podle uvedených aktuálních informací o implementacích). Dále se obnoví proces evoluce, která pokračuje dál podobně jako před chvílí, ale již s promíchanými jedinci napříč generacemi.

Vzhledem k tomu, že při složitějších problémech evoluční proces může v současné verzi aplikace běžet velice dlouho (řádově i dny), nemá zatím smysl aby aplikace kontrolovala sama dosažení nejvyšší shody implementace se specifikací. Proto zastavení je v současné verzi řešeno pouze tak, že evoluci zastaví sám uživatel. Uživatel má mnoho příležitostí občas nahlédnout na aktuální stav, případně aplikaci pozastavit a ověřit si jaké shody již bylo dosaženo u jednotlivých vybraných implementací, včetně kontroly shody souborem out.txt. Poté může uživatel libovolně měnit parametry, kdyby evoluce vykazovala známky stagnace, nebo jinak zasahovat do evolučního procesu. Pro jednoduché problémy nejsou zásahy uživatele příliš potřeba, pro řešení složitějších problémů je však lepší uživatelův aktivní přístup. Ten může v průběhu podle potřeby měnit parametry evoluce, nebo například přidávat buňky kdyby se zdálo že jich bylo na začátku zvoleno málo. V opačném případě by aplikace musela sama sledovat statistiku procesu a autonomně si měnit sama sobě parametry podle toho jestli evoluce pokračuje vhodným tempem, nebo stagnuje. Toto by mohlo být však předmětem nějaké pokročilejší verze této aplikace, neboť toto není zcela jednoduchý problém a jeho řešení zde nebylo zahrnuto.

10.3 Popis datových formátů

Tato kapitola se zaměřuje na popis datových formátů se kterými pracuje program GenAlg. Během provozu programu se používají pouze tyto dva datové formáty (typy souborů):

- **Soubor zdrojové elementární specifikace *.fdf (Function Definition File):** Soubor slouží pro hledání cílové implementace. Soubor je binárního typu a obsahuje popis hledané funkce formou sady požadovaných vstupů (binární masky + hodnoty na vstupech) a k nim očekávaných výstupů (binární masky + hodnoty na výstupech). Je popsáno i dynamické chování během výpočtu vhodnosti nalezené implementace: Kdy se má obsah buněk resetovat (zpravidla před nesledovanými kroky přechodného děje), kolik iteračních kroků se má provést, kde nejsou sledovány výstupy (není vypočítávána vhodnost) a kolik se jich má provést kde se výstupy už sledují (provádí se součet všech shod specifikace s implementací).
- **Soubor generací *.gen (Generation file):** V kterékoliv fázi hledání cílové implementace lze po zastavení evolučního procesu uložit formou generací všechny v tu chvíli nalezené implementace do jednoho souboru. Tento soubor lze později opět kdykoliv nahrát zpět do programu a pracovat s jeho nalezenými implementacemi. Soubor je binárního typu a kromě hlaviček, které popisují všechny parametry každé z generací, soubor obsahuje i obsah všech 8mi paralelních generací z 8mi souběžně běžících evolučních procesů. Každá z nalezených implementací uvnitř generací je binárním vektorem, kdy každý má sám svou vlastní hlavičku a tělo a jsou v souboru umístěny jeden za druhým.

Tab. 10.3.1 Formát binárního souboru pro popis hledaných algoritmů *.fdf

pozice bajtů v souboru	část souboru	počet bajtů	položka struktury	standardní hodnota
1 .. 4	hlavička *.fdf souboru	4	velikost hlavičky	75
5 .. 8		4	ukazatel na data v paměti (nedůležité)	0
9 .. 35		27	signatura souboru pro kontrolu integrity	"jgoewiuie89486 3u32hjgfpbvch"
36, 37		2	počet všech definic= sledované kroky + nesledované kroky	0 .. 65535
38, 39		2	počet vstupních bajtů	1 .. 65535
40, 41		2	počet výstupních bajtů (v současné době se doporučují maximálně dva)	1 .. 2
42 .. 71		30	externí dll soubor pro rozšíření programu (není používán, musí být defaultní jméno)	"def_fit_calc.dll",0, 0,0,0,0,0,0,0,0,0,0,0,0
72 .. 75		4	vypočítaný počet shod (dobrovolné, není nutno nastavovat)	0 .. 2^32-1
76	první definice páru vstup/výstup pro nesledované kroky s resetem všech buněk na začátku	1	sledované (stabilní) kroky * 2 + 1 (+1: reset všech buněk, +0: žádný reset na začátku) počet kroků zde vždy 1 => 1 * 2 + 1 = 3	vždy 3
77		1	maska prvních 8mi vstupů	1 .. 255
78		1	8 binárních vstupů x0 .. x7	0 .. 255
(79)		(1)	(volitelná maska dalších 8mi vstupů)	(1 .. 255)
(80)		(1)	(volitelně dalších 8 binárních vstupů x8 .. x15)	(0 .. 255)
79, 80		2	počet nesledovaných (nestabilních) kroků, musí být větší než nula	2 .. 65535
81, 82		2	rezervováno, vždy 0	0
83		1	maska prvních 8mi výstupů	1 .. 255
84		1	8 binárních výstupů y0 .. y7	0 .. 255
(85)		(1)	(volitelná maska dalších 8mi výstupů)	(1 .. 255)
(86)		(1)	(volitelně dalších 8 binárních výstupů y8 .. y15)	(0 .. 255)
85	první definice páru vstup/výstup pro sledované kroky bez resetu	1	sledované (stabilní) kroky * 2 + 0 (není reset všech buněk) počet kroků musí být větší než nula	sudá čísla > 0
86		1	maska prvních 8mi vstupů (stejná jako předchozí pro nesledované kroky)	1 .. 255
87		1	8 binárních vstupů x0 .. x7 (stejné jako předchozí pro nesledované kroky)	0 .. 255
(88)		(1)	(volitelná maska dalších 8mi vstupů)	(1 .. 255)
(89)		(1)	(volitelně dalších 8 binárních vstupů x8 .. x15)	(0 .. 255)
88, 89		2	počet nesledovaných (nestabilních) kroků, zde rovno jedné	1
90, 91		2	rezervováno, vždy 0	0
92		1	maska prvních 8mi výstupů	0 .. 255
93		1	8 binárních výstupů y0 .. y7	1 .. 255
(94)		(1)	(volitelná maska dalších 8mi výstupů)	(1 .. 255)
(95)		(1)	(volitelně dalších 8 binárních výstupů y8 .. y15)	(0 .. 255)
94 .. Nmax	další definice	n	Sekvence dalších definic složené z párů nesledovaných a sledovaných kroků. Každý pár má v první definici reset (signalizuje číslo 3).	

Tab. 10.3.2 Formát binárního souboru Generací nalezených implementací *.gen

pozice bajtů v souboru	část souboru	počet bajtů	položka struktury	standardní hodnota
1 .. 4	hlavička *.gen souboru	4	velikost hlavičky	43
5 .. 8		4	ukazatel na data v paměti (nedůležité, lze i 0)	?
9 .. 35		27	signatura souboru pro kontrolu integrity	"jfweoiiuwe7665873 dshfoowgdw"
36 .. 39		4	počet všech implementací v jedné generaci	1 .. 2^32 -1
40 .. 43		4	počet všech buněk v jedné implementaci	1 .. 2^32 -1
44 .. 47	hlavička první implementace	4	velikost hlavičky implementace (binárního vektoru)	42
48 .. 51		4	ukazatel na data v paměti (nedůležité, lze i 0)	?
52 .. 78		27	rezervováno	27* bajt 0
79		1	počet bajtů kolik zabírá jedna buňka	4
80 .. 83		4	počet všech buněk v této implementaci	1 .. 2^32 -1
84		1	poslední status který implementace nabyla	10 – rodič1 9 – rodič2 nebo nepoužit 5 – potomek1 6 – potomek2 0 – neprošel selekcí a nepoužit
85		1	bitová maska omezující ukazatele buněk A a B	0 .. 255
86 .. 89	1. fitness	4	poslední spočítaná shoda (fitness, vhodnost)	1 .. 2^32 -1
90	první buňka první implementace	1	rel. Adresa B: funkce Fn: minulá hodnota y B2, B1, B0; F3, F2, F1, F0; y_	0 .. 255
91		1	rel. Adresa B – vyšší část: B10, B9, B8, B7, B6, B5, B4, B3	0 .. 255
92		1	rel. Adresa A: funkce Fn: aktuální hodnota y A2, A1, A0; F7, F6, F5, F4; y	0 .. 255
93		1	rel. Adresa A – vyšší část: A10, A9, A8, A7, A6, A5, A4, A3	0 .. 255
(94)	další buňky první implementace	(1)	……	(0 .. 255)
(95)		(1)	……	(0 .. 255)
(96)		(1)	……	(0 .. 255)
(97)		(1)	……	(0 .. 255)
……				
(44 + 42 + 4 + n * 4) (n=počet buněk)	hlavička další implementace + fitness	46	……	……
(44 + 42 + 4 + n * 4 + 46) (n=počet buněk)	buňky další implementace	n * 4	……	……

Následuje 7 dalších generací se stejnou strukturou jakou měla první generace až sem, včetně všech hlaviček. Všech 8 generací zabírá 8 * (43 + m *(42 + 4 + n*4)) bajtů, kde m= počet implementací a n= počet buněk.

10.4 Příprava nové specifikace

V této kapitole se zaměříme na to, jakým způsobem lze připravovat *.fdf soubory obsahující elementární specifikace nutné pro hledání implementací. Pro co nejvyšší efektivitu celého evolučního procesu byl pro zápis všech případů párů vstup/výstup, popisující hledané chování implementace, zvolen binární formát. Aplikace GenAlg si tento soubor načte do paměti pro své účely hledání implementace a hned jej i v této paměti používá ve stejném formátu jako byl už uložen v souboru.

Dále popsaný způsob přípravy binárního souboru samozřejmě nemusí být použit, je to pouze jedna z možností, neboť jsou i jiné známé způsoby jak lze vytvářet binární soubory. Většinou se setkáváme se zápisem dat v textovém formátu k čemuž lze obvykle použít i například v operačním systému Windows známý program „zápisník“ (notepad). Pro vytváření binárních souborů (jako je v tomto případě soubor *.fdf), textový editor bohužel tak snadno použít nelze. Méně zkušení uživatelé většinou nejsou schopni vytvářet binární soubory s libovolným obsahem (oproti textovým souborům), právě kvůli nedostupnosti vhodných nástrojů.

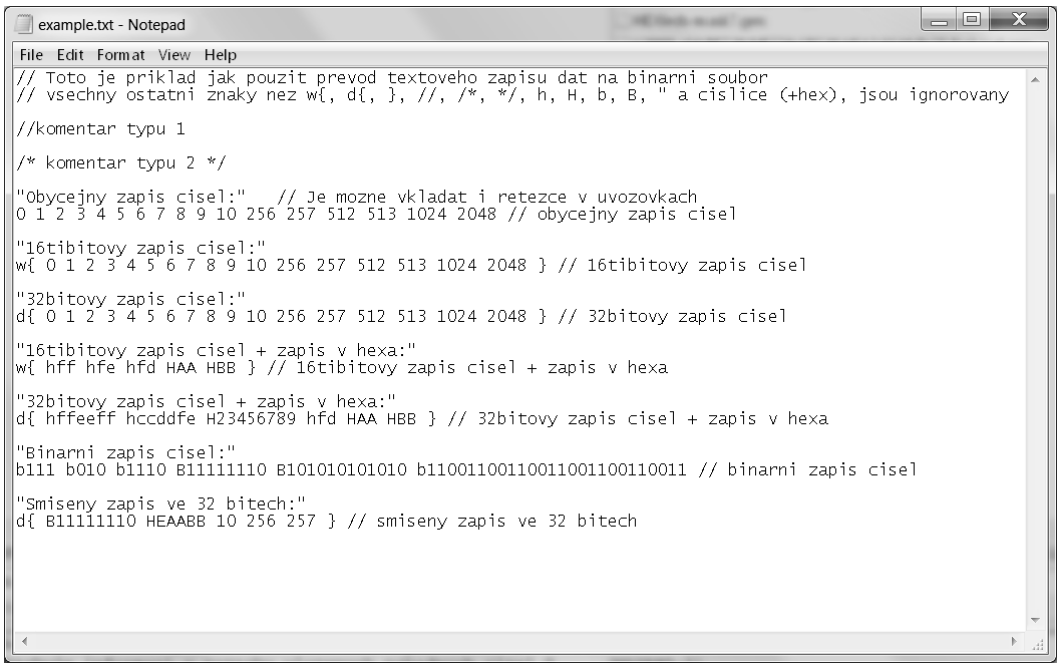
Pro účely snadné manipulace s obsahem binárních souborů byla vytvořena jednoduchá konzolová aplikace spustitelná v operačním systému Windows nazvaná CSV2BIN. Její princip spočívá v tom, že si vytvoříme textový popis očekávané struktury a obsahu jaký má být s pomocí této aplikace uložen do binárního souboru a ten pak bude aplikací vygenerován. Výhodou je hlavně to, že pro tento popis lze opět využít libovolný textový editor (například zápisník „notepad“).

Po vytvoření textového souboru s popisem obsahu binárního souboru, tento soubor použijeme jako vstup pro program CSV2BIN zadáním následujícího řetězce na příkazové řádce operačního systému: „csv2bin.exe popis.txt soubor.bin“

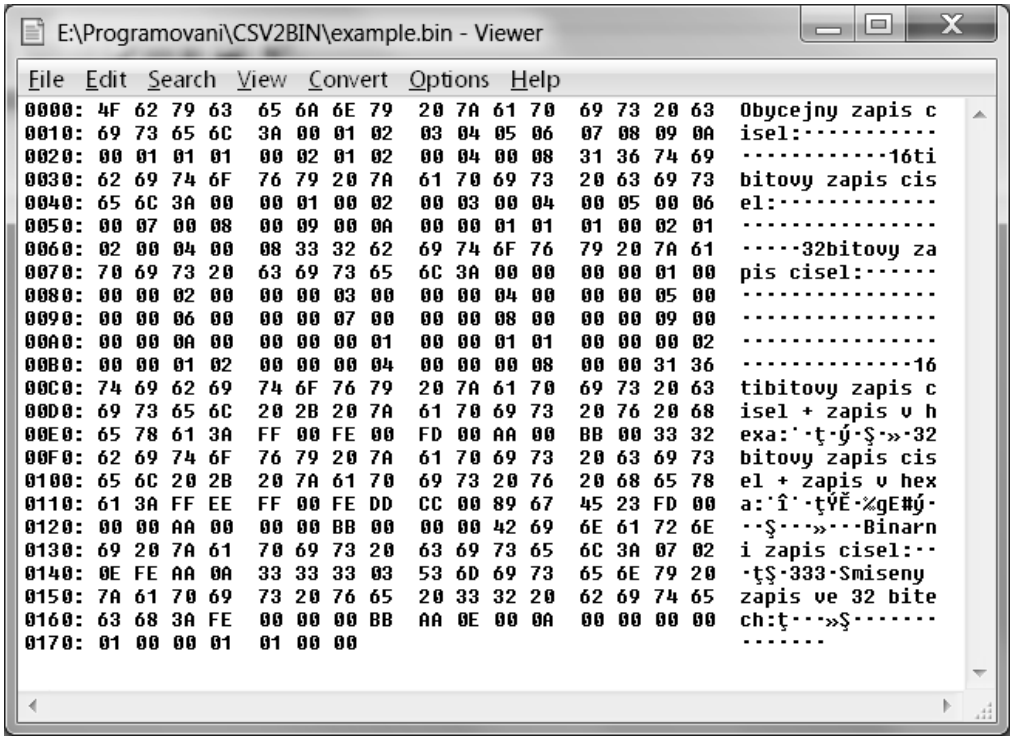
Zde popis.txt obsahuje náš textový soubor s popisem obsahu a soubor.bin bude následně vytvořen jako binární soubor přesně s tím obsahem, jaký byl předtím popsán v textovém editoru. Názvy souborů lze samozřejmě změnit na ty, které jsou pro nás v tu chvíli aktuální a pro binární soubor popisující specifikaci cílové implementace bude použita koncovka *.fdf místo *.bin. Takže například nástroj pro převod spustíme takto: „csv2bin.exe zdroj_specifikace_algoritmu.txt specifikace.fdf“

Na obrázku **Obr. 10.4.1** na následující straně je vyobrazeno, s pomocí jednoduchého příkladu, jaký zápis pro aplikaci csv2bin.exe se používá pro popis struktury obsahu binárního souboru s použitím textového editoru.

Další obrázek **Obr. 10.4.2** znázorňuje výsledný vygenerovaný binární soubor.



Obr. 10.4.1 Příklad popisu struktury binárního souboru obsahující všechny možné prvky které program CSV2BIN umí rozlišit a použít v example.txt



Obr. 10.4.2 Výsledný obsah binárního souboru example.bin vygenerovaného s pomocí programu CSV2BIN použitím example.txt

Jako praktický příklad převodu specifikace popsané v textovém editoru je na obrázku **Obr. 10.4.3** uvedena specifikace algoritmu pro zobrazení číselné informace s pomocí 7mi segmentů LED a na obrázku **Obr. 10.4.4** je výsledný vygenerovaný *.fdf binární soubor.

E:\Programovani\API\GenAlg\fdf\hexleds\LEDnumbers.txt - Viewer

File Edit Search View Convert Options Help

```
//hlavicka:
d{ 75 } // sizeof(DEFHEAD) : 75
d{ 0 } // ukazatel na zacatek dat
"jgoewiuie894863u32hjgfpbvch" // signature of the file - 27 bytes
w{ 20 } // Number of all definition cases
w{ 1 } // Contains number of input bytes
w{ 1 } // Contains number of output bytes
"def_fit_calc.dll" 0 "             "    // reserved (30 bytes)
d{ 770 } //Maximalni pocet bodu ktere lze ziskat za celou sadu definici
              //  11 * 7 * 10 = 770

// Zde zacinaji vlastni data

//modus, maska vstupu, input data, Nb of steps, reserved, maska vystupu, output data
3 h0F b00000000 w{ 12 0 } h7f b01110111 // case 1 : 0
20 h0F b00000000 w{ 1 0 } h7f b01110111 // case 1 : 0

3 h0F b00000001 w{ 12 0 } h7f b00010100 // case 2 : 1
20 h0F b00000001 w{ 1 0 } h7f b00010100 // case 2 : 1

3 h0F b00000010 w{ 12 0 } h7f b00111011 // case 3 : 2
20 h0F b00000010 w{ 1 0 } h7f b00111011 // case 3 : 2

3 h0F b00000011 w{ 12 0 } h7f b00111110 // case 4 : 3
20 h0F b00000011 w{ 1 0 } h7f b00111110 // case 4 : 3

3 h0F b00000100 w{ 12 0 } h7f b01011100 // case 5 : 4
20 h0F b00000100 w{ 1 0 } h7f b01011100 // case 5 : 4

3 h0F b00000101 w{ 12 0 } h7f b01101110 // case 6 : 5
20 h0F b00000101 w{ 1 0 } h7f b01101110 // case 6 : 5

3 h0F b00000110 w{ 12 0 } h7f b01101111 // case 7 : 6
20 h0F b00000110 w{ 1 0 } h7f b01101111 // case 7 : 6

3 h0F b00000111 w{ 12 0 } h7f b00110100 // case 8 : 7
20 h0F b00000111 w{ 1 0 } h7f b00110100 // case 8 : 7

3 h0F b00001000 w{ 12 0 } h7f b011111111 // case 9 : 8
20 h0F b00001000 w{ 1 0 } h7f b011111111 // case 9 : 8

3 h0F b00001001 w{ 12 0 } h7f b01111110 // case 10 : 9
20 h0F b00001001 w{ 1 0 } h7f b01111110 // case 10 : 9
```

Obr. 10.4.3 Příklad popisu struktury binárního souboru popisující jednotlivé definice párů vstup/výstup ve specifikaci zobrazení čísel LED

E:\Programovani\API\GenAlg\fdf\hexleds\LEDnumbers.fdf - Viewer

File Edit Search View Convert Options Help

```
0000: 4B 00 00 00  00 00 00 00  6A 67 6F 65  77 69 75 69  K·······jgoewiui
0010: 65 38 39 34  38 36 33 75  33 32 68 6A  67 66 70 62  e894863u32hjgfpb
0020: 76 63 68 14  00 01 00 01  00 64 65 66  5F 66 69 74  vch·····def_fit
0030: 5F 63 61 6C  63 2E 64 6C  6C 00 20 20  20 20 20 20  _calc.dll·
0040: 20 20 20 20  20 20 20 02  03 00 00 03  0F 00 0C 00          ·········
0050: 00 00 7F 77  14 0F 00 01  00 00 00 7F  77 03 0F 01  ··■w·····■w···
0060: 0C 00 00 00  7F 14 14 0F  01 01 00 00  00 7F 14 03  ····■·····■···
0070: 0F 02 0C 00  00 00 7F 3B  14 0F 02 01  00 00 00 7F  ·····■;·····■
0080: 3B 03 0F 03  0C 00 00 00  7F 3E 14 0F  03 01 00 00  ;·····■>·····
0090: 00 7F 3E 03  0F 04 0C 00  00 00 7F 5C  14 0F 04 01  ·■>·····■\····
00A0: 00 00 00 7F  5C 03 0F 05  0C 00 00 00  7F 6E 14 0F  ···■\·····■n··
00B0: 05 01 00 00  00 7F 6E 03  0F 06 0C 00  00 00 7F 6F  ·····■n·····■o
00C0: 14 0F 06 01  00 00 00 7F  6F 03 0F 07  0C 00 00 00  ·····■o·····
00D0: 7F 34 14 0F  07 01 00 00  00 7F 34 03  0F 08 0C 00  ■4·····■4·····
00E0: 00 00 7F FF  14 0F 08 01  00 00 00 7F  FF 03 0F 09  ··■·····■·····
00F0: 0C 00 00 00  7F 7E 14 0F  09 01 00 00  00 7F 7E     ····■~·····■~
```

Obr. 10.4.4 Výsledný obsah binárního souboru LEDnumbers.fdf vygenerovaného s pomocí programu CSV2BIN použitím textového popisu LEDnumbers.txt

10.5 Zkušenosti s provozem aplikace a doporučení pro její provoz

Během vývoje aplikace GenAlg, při jejím testování a během praktického využití pro přípravu vzorových příkladů bylo nashromážděno určité množství zkušeností, které jsou soustředěny v této kapitole.

V současné době aplikace není schopna samostatně analyzovat průběh evolučního procesu a na základě těchto statisticky získaných údajů neumí dále sama měnit svou strategii jakým způsobem evoluce má probíhat. Tato nedokonalost vede k tomu, že u složitějších problémů, evoluční proces po nějaké době uvízne v bodě, odkud se jen obtížně sám dostává. Pokud se mu to přesto podaří, tak až po neúměrně delším čase, než při vhodné a cílené změně parametrů podle aktuálního vývoje.

Ze začátku, během postupného vývoje, měla aplikace jen velmi omezené možnosti a ani nebylo možné měnit žádné parametry, které by mohly nějakým způsobem evoluční děj ovlivnit - vše bylo nastaveno pevně. Aplikace uměla řešit jen ty nejjednoduší typy problémů a evoluční děj probíhal velmi dlouho. Z toho vyplývá nejprimitivnější typ strategie která byla použita jako první, Strategie A:

- Nastaví se základní parametry které budou neměnné, jako jsou velikost generace a počet buněk pro každý vektor v generaci
- Nahraje se specifikace hledané funkce *.fdf, kde jsou přednastavené parametry: parametr pro délku přechodného děje - u nějž se nesledují výstupy a parametr délky ustáleného stavu - u kterého se počíta míra shody fitness.
- Počet vstupů a výstupů zůstává nezměněn a i maska vystupů je pevně dána.
- Pravděpodobnosti selekce a mutací jsou pevně dané a nemění se.
- Spustí se evoluční proces a nechá se bežet tak dlouho, dokud vypočítaná shoda u většiny jedinců nedosáhne nejvyšší možné hodnoty – poté uživatel celý proces zastaví. Do té doby než je proces zastaven, tak uživatel nijak do běhu nezasahuje, jen občas zkontroluje jestli míra shody již odpovídá jeho požadavkům.

Strategii A lze použít spíše na řešení jednodušších problémů s menším počtem definovaných stavů pro vstupy a výstupy ve specifikaci, s menším počtem používaných vstupů, výstupů a jen s kratšími binárními vektory. I přes jednoduchost strategie se lze dopracovat k hledanému řešení s přihlédnutím k tomu že jsou zadány pouze jednodušší problémy a uživatel má mnoho času aby nechal evoluci běžet dostatečně dlouho (mohou to být i dny, nebo i týden).

Od strategie A se odvozují další možné způsoby, jak průběh evoluce ovlivňovat a jak urychlit nalezení hledané implementace spolu s ošetřením stavů kdy se evoluce předčasně zachytí v lokálním extrému a nechce se dále vyvíjet. Doporučuje se:

- Strategie B: Evoluci provést po menších krocích, kdy se před spuštěním evoluce do masky výstupů nastaví číslo 1, které znamená, že se bude hledat řešení pouze pro jeden výstup. Není potřeba nic dalšího nastavovat, nemusí se ani měnit definice *.fdf, změna má dopad pouze pro aktuální

stav hledání. Je zajímavé, jak se hledání pouze jednoho výstupu výrazně urychlí a pokud byl problém při plném počtu výstupů řešení nalézt, většinou bývá potom řešení pro jeden výstup nalezeno velmi rychle. Po nalezení nejvyšší možné shody se přidá pro hledání další výstup s pomocí masky nastavené na číslo 3. Po každém nalezení nejvyšší shody se obdobně přidávají další výstupy pomocí masek 7, 15, 31, atd. až do nejvyšší možné masky 65535, která je použitelná pro 16 výstupů. Pro vyšší počet výstupů se už změna masky nedoporučuje používat a nelze tuto masku snadno měnit za provozu (kromě opětovného nahrávání nové *.fdf definice). Číslo masky odpovídá vzorci $m = 2^n - 1$, kde n odpovídá počtu výstupů. Jiné hodnoty se nedoporučují protože s nimi aplikace nepočítá. Pokud nastane podezření, že evoluce uvízla, je možné poslední výstup odebrat a nechat chvíli běžet evoluci bez něj a později ho znovu přidat. Tím se obsah vektorů změnil a je možné že už k uvíznutí nedojde. Po přidání posledního výstupu je možné z uvízlé evoluce uniknout i tak, že se nastaví maska, jako bychom očekávali o jeden výstup víc. Toto nelze provést, pokud máme 8 výstupů a chceme přidat devátý, nebo pokud máme 16 výstupů a chceme přidat 17tý protože by se musel nahrát jiný *.fdf soubor který by s tím počítal. Hledání probíhá tak, že se na virtuálním přidaném výstupu očekává po celou dobu číslo 0 a evoluce se pokouší takovou implementaci která to splňuje najít a dojde k promíchání obsahu hledaných vektorů. Tento postup má smysl pouze pro uniknutí z uvízlého stavu a je dobré přidaný výstup opět odebrat, pokud se nám zdá že evoluce z uvíznutí unikla.

- Strategie C: V této strategii se počítá s postupnou změnou parametrů pravděpodobností pro selekci, pro aditivní mutaci a pro xorovou mutaci. Začneme s nastavením vysokých hodnot u aditivní a xorové mutace (kolem 17%) a s nastavením nízkých hodnot u pravděpodobnosti selekce (kolem 57%). Dále necháme evoluční děj běžet tak dlouho, dokud se neustálí v nějakém rovnovážném stavu a kdy je zřejmé, že se už kvalita shody nezvyšuje. Obvykle to lze poznat i podle toho, že nápadně u všech paralelně běžících evolucí je dosažena stejná hodnota míry shody (fitness) a tato se již dále nemění. Při dosažení rovnovážného stavu, které mohou být každý dosažen i v rozmězí až několika hodin, se doporučuje snížit pravděpodobnost obou mutací o jedno procento níž a zvýšit pravděpodobnost selekce o jedno procento výš. Tímto způsobem dosáhneme vyšší shody (fitness) po dosažení dalšího rovnovážného stavu. Tento postup lze po různě dlouhých časových etapách opakovat až do dosažení pravděpodobnosti mutací kolem 5% a pravděpodobnosti selekce kolem 70% kdy ve většině případů již bývá dosaženo nejvyšší možné shody (fitness). Pokud máme podezření na uvíznutí evolučního procesu, je možné změnit i směr ve kterém jsme přidávali nebo ubírali procenta u selekce nebo u mutací. Nezaškodí občas snížit pravděpodobnost selekce, nebo zvýšit pravděpodobnosti mutací. Můžeme krátkodobě snížit míru shody (fitness), ale získáme tím únik z uvízlého evolučního procesu. Uvíznutí bývá často způsobeno přemnožením implementací jednoho typu, kdy všechny paralelní evoluce jsou naplněny jedinci, kteří si jsou vůči sobě příliš podobní. Genetická diverzifikace je příliš nízká a evoluce dále

nepostupuje. Při zjištění, že došlo k přílišnému přemnožení a útlumu, pomůže krátkodobé snížení pravděpodobnosti selekce a zvýšení pravděpodobnosti mutací i o několik procent. Tímto dojde k postupnému vymření jedinců s majoritním genetickým obsahem a můžeme později posunout evoluční proces k lepším výsledkům. Při postupném čekání na ustálené stavy během změn pravděpodobností, není vhodné nechat běžet evoluční proces příliš dlouho bez kontroly, protože po několika hodinách, kdy už pravděpodobně evoluce byla v rozumném ustáleném stavu, dochází k přemnožení a přílišnému výskytu sobě podobných jedinců. Tím se může velmi vyčerpat potenciál k dalšímu růstu shody. Pochopitelně by teoreticky mohlo být možné, nastavit rovnou parametry pro pravděpodobnost selekce na 70% a pravděpodobnosti mutací na 5%. Je ale jisté, že celý evoluční proces bude trvat velice dlouho a pravděpodobně snadno uvízne bez dosažení nejvyšší shody. Postupná změna parametrů a dosahování ustálených stavů má tu výhodu, že na začátku při vysokých hodnotách mutací a snížené pravděpodobnosti pro selekci se generuje velké množství nových jedinců s různým genetickým materiálem a rozsáhlé změny obsahu binárních vektorů jsou žádoucí, protože se prohledává velká část stavového prostoru v kratším čase. Později už rozsáhlé změny genetického materiálu nejsou tak žádoucí a jsou spíše na škodu. Postupujeme se stále menšími úpravami, které nejsou tak invazivní a tolik neohrožují již nalezené struktury. Docílí se toho, že postupně dospějeme do cílového stavu kdy nalezení jedinci dosáhnou nejvyšší možné shody.

- Strategie D: Tato strategie spočívá ve změně parametrů, které se týkají počtu kroků - definované jako přechodný děj kdy se nesledují výstupy a počtu kroků kdy se výstupy sledují - definované jako stálený stav kdy se vypočítává shoda. Pro většinu uvedených strategií by mělo stačit pevné nastavení těchto parametrů, které si zvolíme na začátku a dále je neměníme. Typicky používané hodnoty mohou být 10 kroků pro „Unstable steps“ a 5, 8, nebo 10 kroků pro „Stable steps“. Při dosažení ustáleného stavu, kdy se zřejmě evoluce dále nevyvíjí, je možné snižovat, nebo zvyšovat počet kroků přechodného děje, což může mít vliv na aktuálně dosaženou míru shody. Pokud snižíme počet kroků a nejvyšší dosažená shoda se nezmění, je zřejmé, že jsme měli nastavenu rezervu a kroky přechodného dějě byly naddimenzované a je možno je snížit. Pokud se ale dosažená míra shody sníží, je zřejmé, že se pohybujeme na hranici přechodného děje a začínáme do něj zasahovat. Podle momentální situace si můžeme vybrat jestli vrátíme hodnotu zpět jak byla předtím a délku přechodného děje akceptujeme, nebo se ponecháním snížené hodnoty pokusíme nalézt jiné implementace, které naopak budou založeny na kratším přechodném ději. Varianta s kratším přechodným dějem má tu výhodu, že by výsledná složitost nalezeného řešení implementace měla být o něco snížena a efektivní počet buněk využívaných řešením by měl být lepší. Zkracováním přechodného děje se snižuje tedy i počet pro řešení využívaných buněk a zjednodušené řešení umožňuje další růst během hledání shody. Při delších přechodných dějích se na výsledném chování obvykle podílí větší počet buněk. Z počátku tato varianta může pomoci k rychlejšímu nárustu shody, ale aktuální dílčí řešení vyčerpá více buněk

než by bylo nutno a tyto potom chybí pro další nárust shody. Je proto vhodné, pokud je to možné, délku přechodného děje snižovat aby se vyvinul tlak na zjednodušování aktuálních řešení. Samozřejmě pro oživení evolučního vývoje může být vhodné občas toleranci pro délku přechodného děje zvýšit, pokud to pomůže dosáhnout vyšší shody, s tím že počítáme s pozdějším snížením počtu kroků přechodného děje opět tak aby se vyvinul tlak na zjednodušení nalezených řešení. Je dobré vyzkoušet i extrémní případ, kdy se pro parametr „Unstable steps“ nastaví číslo 2 a „Stable steps“ se odpovídajícím způsobem zvýší například na 15 aby se dodržel celkový počet kroků jako byl předtím. Dosáhneme toho, že by se evoluční proces měl sám snažit o hledání řešení s co nejkratší délkou přechodného děje a díky tomu by měl využívat pro řešení i co nejmenší počet buněk. Tato varianta do určité míry funguje a lze dosáhnout celkem optimálních počátečních řešení. Tyto je ale vhodnější použít spíše jako výchozí stav pro další hledání například s vyšším počtem výstupů, kdy první výstupy byly hledány s krátkým přechodným dějem a u dalších přidaných výstupů se přechodné děje prodlužují. Je to vlastně rozšíření strategie B o změnu délky přechodného děje. Samozřejmě je možné i postupné zkracování přechodného děje použít i na nalezená řešení, která mají již nejvyšší možnou shodu dosaženu. Lze takto zjednodušovat aktuální nalezená řešení a snižovat jejich složitost.

- Strategie E: Tato strategie spočívá ve změně velikosti binárních vektorů – ve změně počtu použitých buněk. Strategie E se kombinuje se strategií B, kdy během evolučního hledání postupně přidáváme další výstupy, ale přitom zvyšujeme i počet buněk které jsou použity v binárních vektorech. Kdykoliv máme pocit, že evoluce uvízla v dílčím vývojovém bodě, je možné se pokusit prodloužit nalezená řešení o jednu další buňku. Není dobré ale tento postup opakovat příliš často a až když jsou vyčerpány jiné možnosti jak se dostat z uvízlé evoluce. Tento postup je výhodný v tom, že můžeme u evoluce začít s malým počtem výstupů a s malým počtem buněk, což má příznivý vliv na rychlost probíhající evoluce, a postupně počet buněk zvyšovat. Jsou zde však bohužel i určité během praxe zjištěné nevýhody. Nedoporučuje se najednou přidávat více než jednu buňku. Vložení buňky probíhá náhodně - na náhodnou pozici se vloží buňka s náhodným obsahem u všech existujících vektorů ve všech 8mi paralelních evolučních procesech. To má za následek, že přestože struktura všech nalezených implementací je zachována (protože se po vložení nové buňky všechny spoje všech buněk správně přepočítají), sníží se ale výrazně podobnost binárních vektorů mezi sebou – každý má náhodnou buňku vloženu jinam a u každého jsou tedy spoje mezi buňkami přepočítány odlišně. Vzniká problém během křížení: Přestože dosažená shoda obou rodičů je vysoká, jejich podobnost je narušena odlišnými posuny buněk při vkládání náhodné buňky a výslední potomci potom mívají horší vypočítanou shodu než jejich rodiče. Při neuváženém použití funkce přidávání buněk může dojít snadno k vymření v tuto chvíli nejlépe ohodnocených jedinců, protože nebudou schopni se spolehlivě reprodukovat. Zvýšení počtu buněk naopak může vyřešit ty situace, kdy máme příliš mnoho sobě podobných jedinců, pokud evoluce zjevně dále

nepostupuje a takto lze populaci sobě podobných jedinců uměle pročistit – po narušení schopnosti jejich reprodukce jich většina vymře kvůli tomu že se jim nepodaří párovat se s podobnými jedinci. Pokud před použitím funkce přidání buňky bylo k dispozici velké množství podobných jedinců, nemělo by se stát, že by vymřeli uplně všichni a došlo tím k degradaci nalezených řešení, spíše se jejich počet výrazně sníží. Je nutné mít na paměti ještě jednu nevýhodu strategie E: V současné verzi aplikace GenAlg, není možné již jednou přidanou buňku opět odebrat, lze buňky pouze přidávat. Pokud přidáme příliš mnoho buněk, můžeme dospět do bodu, kdy evoluce postupuje velice pomalu kvůli vyšší výpočetní náročnosti, ale i proto, že s přidáním každé další buňky rozšiřujeme stavový prostor hledaných řešení. Toto může v určité chvíli být přínosem, pokud původně malý počet buněk není schopen vystačit na reprezentaci hledaného řešení. Za jiných okolností se ale prohledávání stavového prostoru spíš zhoršuje, protože se stane po přidání přehnaného počtu buněk příliš rozsáhlým.

- Strategie F: Jako velice úspěšná strategie se ukázal být postup, kdy při přítomnosti většího počtu výstupů se problém rozdělí na více částí, které se budou řešit odděleně. Pokud máme například u nějakého řešeného problému 10 výstupů, lze řešení problému rozdělit na dvě části, u kterých každé řešení bude mít jen 5 výstupů. Je možné, že nalézt řešení pro všech 10 výstupů by bylo příliš obtížné, ale pro 5 výstupů už to tak težké není. Z jedné specifikace která by popisovala 10 výstupů se vytvoří dvě specifikace po 5ti výstupech a musí se evoluční hledání potom nechat běžet dvakrát – pro každou specifikaci zvlášť. Nevýhodou je, že potřebný počet buněk které popisují nalezená řešení bude dohromady jistě vyšší než kdyby se řešení pro všechny výstupy hledalo najednou. Pokud by to bylo potřeba, je samozřejme technicky možné nalezené vektory obou řešení sloučit do jednoho velkého, jde pouze o to jakým způsobem by se oba vektory propojily a jak by se kvůli tomu přepočítali jejich spoje. Aplikace GenAlg tuto funkci nenabízí, v případě potřeby je tato varianta ale možná i například v excelu pokud si vytvoříme na to vhodný algoritmus podle toho jak vektory chceme propojit.

- Strategie G: Tato strategie spočívá v libovolné manipulaci se soubory specifikace *.fdf. Nejmocnějších manipulací probíhajícího evolučního děje lze samozřejmě docílit změnou přímo ve specifikaci hledané implementace. Je možné mít připravenu sadu různých specifikací, které obsahují různá nastavení a popisy vstupů a výstupů jednotlivých případů. Po dosažení určitých dílčích cílů je možné specifikace libovolně měnit a určité useky evolučního děje je možné nechat běžet s různými specifikacemi. Lze tak například začít s menším počtem případů párů vstup/výstup ke kterým je možné přidávat časem další případy jakmile evoluce dosáhne určitých dílčích úspěchů. Tímto způsobem nebude hledání hned na začátku zahlceno a zbržděno přílišnou složitostí specifikace, ale lze postupně složitost stupňovat. Je možné také balancovat mezi stavy kdy by evoluce uvízla v nějakém bodě. Pokud začne evoluce vykazovat známky stagnace, je možné použít mírně upravenou specifikaci, která budě zaměřena na trochu jiné aspekty hledaného řešení a později se

lze opět vrátit k té původní specifikaci jakmile se nám podaří uniknout z vývojové stagnace.

Je dobré pro různé typy problémů používat různé strategie, není vhodné však použít během evolučního procesu pouze jednu strategii ale spíše správnou kombinaci několika výše uvedených typů a obměňovat je podle aktuální situace podle toho v jakém stavu se právě evoluční proces nachází. Některé strategie mají vyšší užitnou hodnotu během počátečních fází hledání, jiné více pomohou pokud jsme už například dosáhli téměř nejvyšší možné shody a hledáme už jen řešení pro několik posledních chybějících bodů shody.

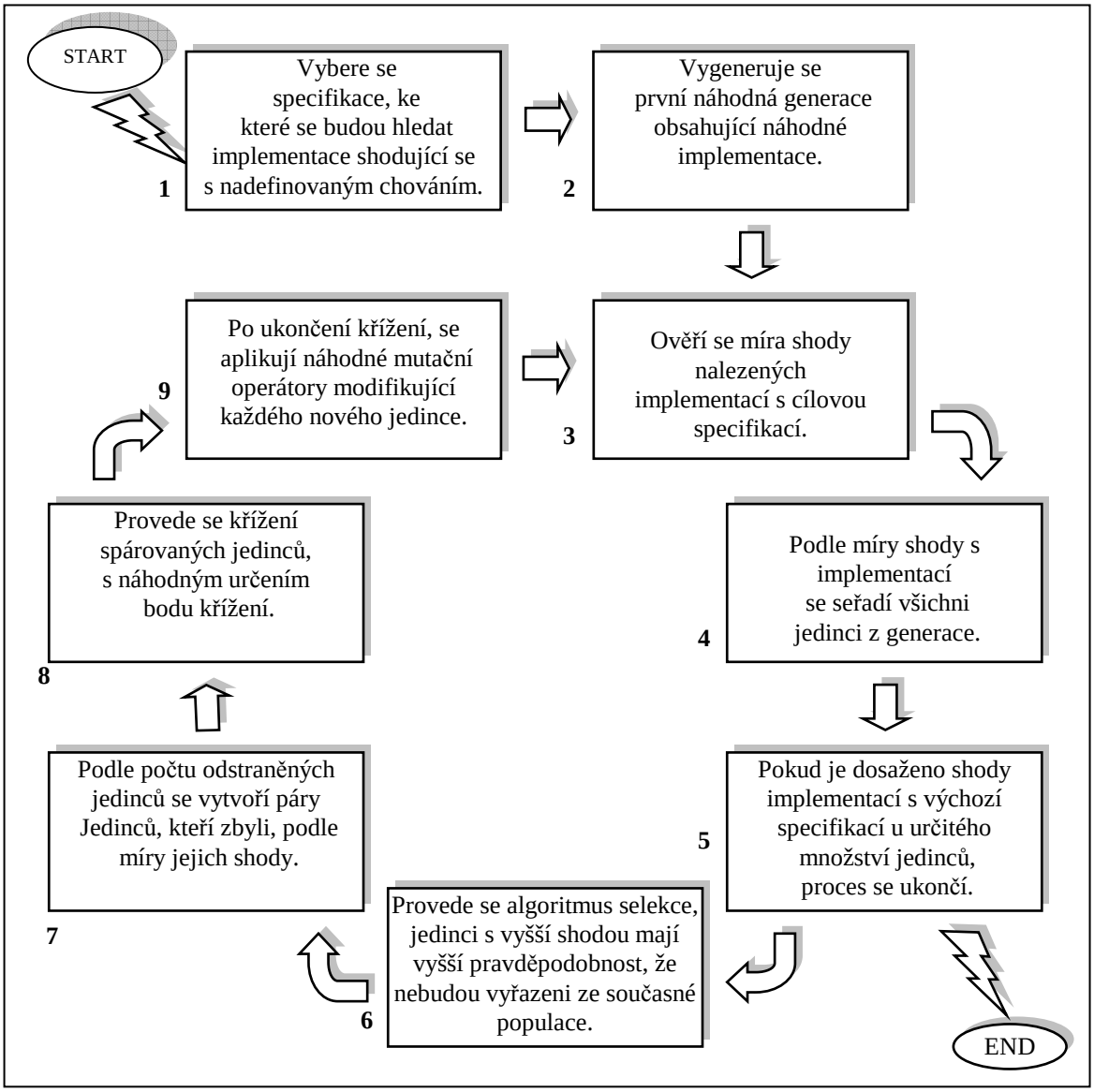
Z předchozího popisu všech uvedených strategií vyplývá, že hledání řešení s pomocí aplikace GenAlg nemusí být vždy tak jednoduché jak by se mohlo původně zdát. Samozřejmě nejlepší řešení by bylo všechny výše uvedené strategie naprogramovat do aplikace, aby se během evoluce prováděly sami. Potom bychom skutečně dosáhli stavu, kdy zadáme specifikaci řešeného problému a řešení bude zcela samo a autonomně nalezeno v co nejkratším čase a s co neefektivnější implementací a s rozumnou mírou složitosti. Bohužel by to pravděpodobně vyžadovalo další roky vývoje aplikace. Musel by se vyvinout účinný mechanizmus vyhodnocování statistiky běžící evoluce, která by sloužila pro detekci stagnujícího vývoje a následně by se podle určitých dosažených ukazatelů a vyhodnocených parametrů rozhodovalo jaká strategie bude v tu chvíli zvolena pro další vývoj. Toto není neřešitelné, ale vyžadovalo by to další a nemalou práci na vývoji aplikace. V tuto chvíli je tedy pouze vytyčen směr, kam by se mohl vývoj aplikace dál ubírat a byly vynalezeny a popsány účinné strategie, které by po úspěšném zapracování kvalitativně vylepšili užitnou hodnotu současné verze aplikace GenAlg.

10.6 Architektura aplikace, popis programových bloků

Aplikace pro genetické programování popisovaná v této práci, se skládá z těchto hlavních programových bloků a umožňují tak běh celého cyklu [18]:

- Definice parametrů genetického programování, parametry pro řízení a ukončení procesu evoluce.
- Náhodné generování počáteční populace jedinců (Generace). Populace obsahuje určitý počet binárních vektorů, kdy každý binární vektor odpovídá hledanému algoritmu s určitou vyčíslitelnou přesností a tento popisuje chování tohoto hledaného algoritmu, pokud se na něj aplikuje celulární procesor logických funkcí.
- Výpočet vhodnosti všech jedinců v rámci jedné populace (Generace). Použije se předpřipravená definice chování hledaného algoritmu, která popisuje určité hodnoty vstupů (používají se na to první buňky vektoru), aplikuje se určitý zadaný počet cyklů celulárního procesoru logických funkcí a výstupy (používají se poslední buňky vektoru) jsou pak porovnány s očekávanými hodnotami (dle definice hledaného algoritmu). Čím více shod je dosaženo, tím vyšší je kvalita jedince (součet všech shod s výstupy v definici hledaného algoritmu).
- Setřídění všech jedinců podle jejich vypočítané vhodnosti (kvality).
- Kontrola splnění ukončovacích parametrů (Jestli byl nalezen hledaný algoritmus nebo ne se určuje podle bodového hodnocení vhodnosti).
- Simulace přirozeného výběru jedinců uvnitř populace; jedinci s nejlepší vypočítanou vhodností mají nejvyšší pravděpodobnost k přežití. Aplikuje se pravděpodobnostní model řízený pravděpodobnostní funkcí.
- Označení jedinců, kteří neprošli selekcí. (V dalších krocích budou odstraněni z populace a místo nich se vygenerují jedinci noví).
- Generování párů ze seznamu jedinců, kteří prošli selekcí a kteří nebyli odstraněni a určení počtu jejich potomků. Počet potomků je opět řízen kvalitou jedinců – méně kvalitní jedinci mají nárok jen na méně potomků.
- Náhodný výběr operátorů pro křížení, proces křížení – vzniká nový jedinec, jehož genetický kód odpovídá kombinaci genetického kódu jeho rodičů. V tomto případě vznikají jedinci dva, kdy po aplikování bodu křížení se využijí obě kombinace způsobu slepení rozdělených (často nestejně velkých) polovin kódů: J1:=(J1A+J2B) a J2:=(J2A+J1B), kdy (J1A+J1B) a (J2A+J2B) jsou původní jedinci a A a B jsou jejich poloviny genetického kódu (Bod křížení nemusí být právě uprostřed).
- Jsou aplikovány náhodné operátory mutace, čímž se mění jakákoliv část genetického kódu nově vzniklých jedinců.
- Celý cyklus se opakuje od bodu, kde se vypočítává kvalita jedinců, dokud není splněno nalezení vhodného jedince, který by měl požadovanou kvalitu (Např. 100% shody definice s nalezeným algoritmem).
- Vyvinutá aplikace nechává běžet 8 paralelně běžících evolucí [21], které jsou zpočátku vůči sobě navzájem izolovány a po uběhnutí určitého počtu evolučních cyklů se nalezení jedinci mezi těmito evolucemi periodicky promíchávají. Tímto

se dosahuje lepší genetické variability a rychleji se daří dosáhnout požadovaného cíle, kromě toho, že i počet všech jedinců, se kterými se v jeden okamžik pracuje, je osminásobný. Tato vlastnost aplikace především vyniká při užití moderních vícejádrových procesorů, kdy každá evoluce může využívat pro svůj běh jedno jádro procesoru (podle počtu jader) a opět se tím získává další nárůst výkonu a kratší hledání cílového algoritmu.

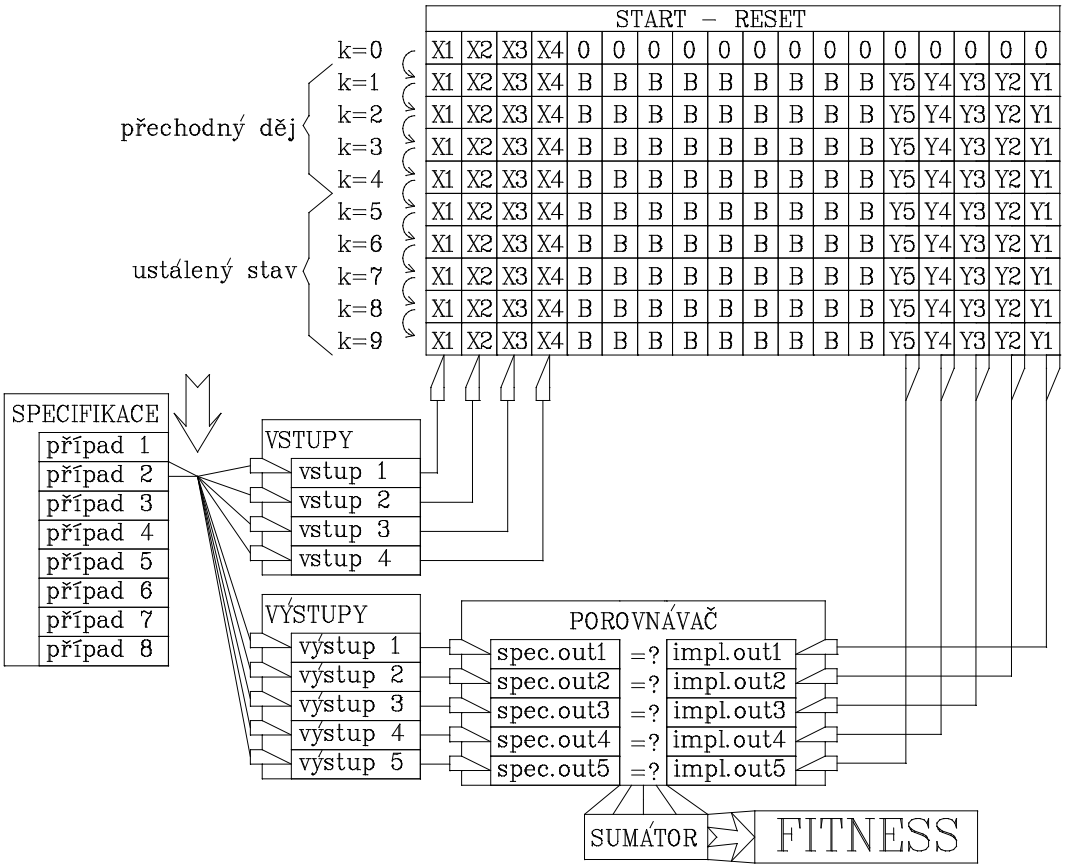


Obr. 10.6.1 Schematické znázornění celého procesu hledání implementací dle zadané specifikace.

10.7 Algoritmus ověření shody implementace se specifikací

Velmi důležitou částí celého evolučního procesu je ověřování míry shody výstupů z nalezených implementací s jejími očekávanými výstupy, které jsou uvedeny v přidružené specifikaci. Informace získané v této fázi významným způsobem ovlivňují celý další proces a způsob, jakým je s jednotlivými implementacemi zacházeno. Míra shody je oceňována počtem bodů, které odpovídají počtu binárních výstupů shodných

s výstupy uvedenými ve specifikaci, pro každý jednotlivý případ a v každém iteračním kroku ve kterém je sledování shody aktivní. Platí, že čím vyššího bodového hodnocení shody (fitness) je dosaženo, tím více se aktuální chování právě oceňované implementace shoduje. Pokud je dosaženo nejvyšší možné hodnoty, lze říci, že cílová implementace byla nalezena a ze strany uživatele je možné celý evoluční proces ukončit.



Obr. 10.7.1 Schematické znázornění procesu výpočtu míry shody - fitness.

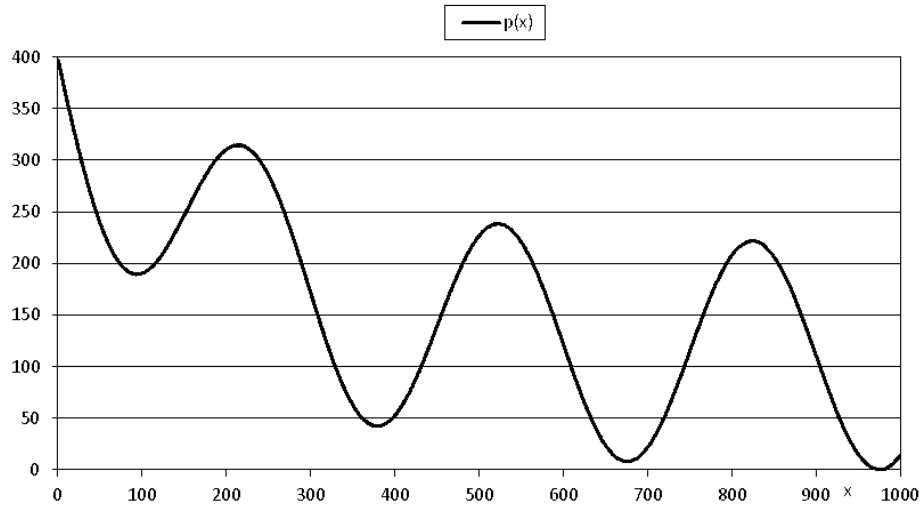
Na obrázku **10.7.1** je zobrazen příklad procesu výpočtu shody (fitness), kde je pro porovnání použita specifikace popisující 8 různých stavů (párů vstup – výstup). Na první 4 buňky celulárního procesoru logických funkcí jsou přivedeny vstupy ze specifikace X1 – X4 a provede se reset čímž se všechny ostaní buňky (kromě těch vstupních) nastaví na výstupní hodnotu 0. Dále v příkladu na obrázku následuje 9 cyklů celulárního procesoru (k = 1 .. k = 9), kdy se provádí v buňkách práve uložená implementace generující výstupy na výstupních buňkách Y1 – Y5. Pro výpočet míry shody se používají pouze ty výstupy, které přísluší ke krokům označené jako “ustálený stav“ (k = 5 .. k = 9) a tyto se během každého z těchto cyklů porovnávají s očekávanými výstupy které jsou uvedeny ve specifikaci. Za každou takovou shodu se přičte jeden bod, v případě neshody se nepřičítá nic. Po dokončení jednoho případu specifikace, se přejde k dalšímu případu a do pracovních vstupů a výstupů specifikace se načtou nové hodnoty které budou použity pro dalších 9 iteračních cyklů. Na začátku iterací každého případu ze specifikace se provádí reset aby výstupy všech buněk začínaly v přesně definovaném stavu – v tomto případě budou během k = 0 na výstupech všech buněk samé nuly. Vyobrazený sumátor sečte počet všech shod během všech kroků ustálených stavů a během všech uvedených 8mi případů ve specifikaci a výsledná hodnota se uloží jako

„fitness“ (míra shody) prověřované implementace. Všechny hodnoty na obrázku jsou uvedeny jen jako příklad a různé specifikace mohou mít různý počet popsaných případů, je možné mít i různý počet vstupů, výstupů, buněk, nebo iteračních cyklů. Rovněž přechodné a ustálené stavy je možné definovat různým způsobem a podle povahy právě řešeného problému.

10.8 Algoritmus selekce jedinců podle jejich kvality

Poté co všechny implementace z celé právě aktuální generace byly použity pro výpočet jejich shody se specifikací a každá získala svou hodnotu míry shody „fitness“, tyto všechny implementace jsou v paměti seřazeny od nejvyšší míry shody až po tu nejnižší míru shody. Následuje proces výběru (selekce), který je inspirován přirozeným výběrem probíhajícím v přírodě, kdy pro svůj život lépe vybavení jedinci mají vyšší pravděpodobnost přežití a pravděpodonost předání genetické výbavy svým potomkům, nežli hůře vybavení jedinci (nebo tací kteří neměli dostatek štěstí pro přežití nebo rozmnožení) kteří jsou uloveni, nebo zahynou kvůli nemoci a nepodaří se jim rozmnožit se a předat dále svou genetickou výbavu. Přesně tento mechanismus je zde simulován. Implementace s vyšší mírou shody mají vyšší předpoklady k reprodukci oproti těm které jsou po seřazení v paměti od nich vzdáleny směrem k nižším hodnotám jejich shody.

Pro simulaci výběru je použita funkce znázorňující pravděpodobnost přežití jednotlivých jedinců uvnitř generace, kde pravděpodobnost, tradičně vyjadřována číselně procenty od 0% do 100%, je kvůli použití celočíselných výpočtů v celém program GenAlg převedena na čísla od 0 do 400 čímž se docílí velikosti nejmenší jednotky pravděpodobnosti = 0,25%. Celá funkce je normalizována na počet jedinců roven počtu 1000.



Obr. 10.8.1 Funkce použitá pro selekci, odvozená ze vzorce p(x) =400 - 0,73*(400*(1-(EXP(-x/200)))+150*SIN(2*PI*(1000-x+50)/300)) normalizována na 1000 jedinců a p_{max} = 400

Zobecněný průběh pravděpodobnosti normalizovaný na 1000 jedinců uvedený na obrázku **Obr. 10.8.1** se přepočítává na aktuální (vyšší) počet jedinců tak, že vzniklé mezihodnoty mají stejnou hodnotu jako jejich sousedé, nebo pokud je konečný počet nižší než 1000, některé hodnoty jsou vynechány. V programu GenAlg zvolená výběrová

funkce obsahuje 3 lokální maxima: Na hodnotách 215, 523 a 825. V okolí těchto extrémů výběrová funkce poskytuje jedincům v generaci vyšší pravděpodobnost, že nebudou odstraněni, že se mohou rozmnožit a že postoupí do dalšího evolučního cyklu. Toto vylepšení zlepšuje genetickou diverzifikaci a dává šanci i jedincům s nižším ohodnocením účastnit se reprodukce. Kromě přepočítání kvůli momentálnímu počtu jedinců na ose x, se upravuje i pravděpodobnost na ose y tak, že se vynásobí právě nastavenou hodnotou v programu GenAlg, která je přidružena k posuvníku označeném jako „p-sel.“. Díky tomu se docílí toho, že v různých fázích evolučního vývoje se zvyšuje, nebo snižuje úmrtnost jedinců a reguluje se tak počet odstraněných nevyhovujících implementací během každého evolučního cyklu. V programu GenAlg byla zvolena strategie udržování konstatního počtu jedinců v každé generaci a právě počet odstraněných jedinců určuje, kolik nových jedinců (potomků), bude momentálně vygenerováno. Tuto míru lze průběžně měnit pomocí nastavení pravděpodobnosti přežití posuvníkem „p-sel.“. Samotný výběr je simulován tak, že pro každého jedince, který má přiřazenou pravděpodobnost přežití podle výběrové funkce, se z generátoru náhodných čísel získá náhodné číslo s hodnotou 0 – 399 a toto číslo je odečteno od momentální pravděpodobnosti přežití. Pokud po odečtení vznikne záporné číslo, jedinec je označen značkou že je určen k odstranění z generace (nepřežil) a pokud vznikne nula, nebo kladné číslo, tak jedinec zůstane v generaci, je označen značkou přežití (přežil) a může se účastnit další reprodukce pokud k tomu dostane příležitost. Přežití jedinců automaticky neznamená, že jedinci budou použiti pro reprodukci. Pravděpodobnost počtu potomků se řídí podobnými pravidly jako při pravděpodobnosti přežití. „Zdatnější“ jedinci získávají právo na vyšší počet potomků než ti „méně zdatní“. Počet potomků je tedy rovněž regulován podle umístění v seznamu jedinců mírou jejich kvality.

10.9 Algoritmus generování párů jedinců a počty potomků

Během procesu selekce byl z generace odstraněn určitý počet jedinců. Protože aplikace využívá strategii udržování konstatní velikosti generace, tak stejný počet jedinců kolik jich bylo právě odstraněno, bude i opět vygenerováno. Přednostně to budou potomci těch jedinců, kteří se pohybují na horním konci žebříčku nejvyšší dosažené shody. Algoritmus je navržen tak, že dochází k párování nejúspěšnějšího jedince s jeho sousedy a to tak, že počet párů, které bude tento jedinec tvořit a současně i počet jeho potomků je dán určitým číslem - jako příklad si vezmeme číslo 10. Takže s 10ti dalšími jedinci kteří jsou v žebříčku pod ním utvoří páry. Každý z těchto párů vygeneruje 2 nové jedince – potomky, tedy jich je celkem 20. V žebříčku se posuneme o dva jedince (ten který byl na druhém místě od shora je přeskočen) a hned ten následující jedinec utvoří s ostatními, kteří jsou v žebříčku pod ním, páry stejným způsobem jako u toho prvního. Rozdíl je ale v tom, že počet párů bude o jeden méně, tedy 9 a celkový počet nových potomků bude dvojnásobek který je roven 18. Tímto způsobem postupujeme v žebříčku směrem dolů (s přeskakováním nejbližšího souseda při každém cyklu kdy se mění počet párů) tak dlouho, dokud se počet párů nesníží až na číslo 1 odpovídající dvěma novým potomkům. Když nyní sečteme všechny vzniklé jedince (potomky) dojdeme k tomuto součtu: 20 + 18 + 16 + 14 + 12 + 10 + 8 + 6 + 4 + 2 = 10 * (10 + 1) = 110 nových jedinců. Pomocí vzorce n = k * (k + 1) lze spočítat celkový počet vygenerovaných nových jedinců n, pokud je předem určen počet párů k se kterými se začne u nejúspěšnějšího jedince v žebříčku.

Celý problém se musí ale otočit, protože v našem případě vycházíme z celkového počtu jedinců, kteří byly v momentálním evolučním cyklu odstraněni a vzniklá mezera musí být zaplněna novými jedinci – jejich počet je znám. Distribuce párů bude stejná, ale s tím rozdílem, že předem nevíme kolik párů a kolik potomků má být vygenerováno u nejúspěšnějšího jedince, naopak známe celkový součet potomků kteří budou mezi všechny páry rozděleny podle výše popsaného algoritmu. Počet párů získáme vyřešením kvadratické rovnice $k * (k + 1) – n = k^2 + k – n = 0$.

Pro určení nejvyššího počtu párů (a následného počtu potomků) u nejúspěšnějšího jedince potřebujeme kladné celočíselné hodnoty, takže výsledek by byl při řešení kvadratické rovnice zaokrouhlován směrem nahoru nebo dolu. Abychom se během evolučního procesu vyhnuli výpočtům, které by mohly zpomalovat celý algoritmus, byla navržena tabulka (uložená do paměti), která má u pořadové hodnoty - odpovídající celkovému počtu nových jedinců, uveden počet párů se kterými se začne u nejvyššího jedince. V tabulce je zahrnuto i zaokrouhlení na nejbližší celé číslo, aby se vždy pracovalo s celými hodnotami. Nový jedinci jsou generováni podle algoritmu párování tak dlouho, dokud se nezaplní celkový počet n. Ukázka tabulky pro určení počtu párů bez nutnosti řešení kvadratické rovnice je uvedena v tabulce **Tab. 10.9.1** Hodnoty v tabulce které odpovídají výsledku *k* bez zaokrouhlení jsou označeny šedou barvou, kdežto k_1 a k_2 představují dvě různé strategie pro zaokrouhlování. U aplikace GenAlg se v současné době používá strategie zaokrouhlování uvedená v tabulce jako k_2.

Tab. 10.9.1 Výchozí počet párů *k* algoritmu určený z celkového počtu potomků *n*

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathbf{k_1}$	1	1	2	2	2	2	3	3	3	3	3	3	4	4
$\mathbf{k_2}$	1	1	1	1	1	2	2	2	2	2	2	3	3	3

n	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$\mathbf{k_1}$	4	4	4	4	4	4	5	5	5	5	5	5	5	5
$\mathbf{k_2}$	3	3	3	3	3	4	4	4	4	4	4	4	4	4

n	29	30	31	32	33	34	35	36	37	38	39	40	41	42
$\mathbf{k_1}$	5	5	6	6	6	6	6	6	6	6	6	6	6	6
$\mathbf{k_2}$	4	5	5	5	5	5	5	5	5	5	5	5	5	6

n	43	44	45	46	47	48	49	50	51	52	53	54	55	56
$\mathbf{k_1}$	7	7	7	7	7	7	7	7	7	7	7	7	7	7
$\mathbf{k_2}$	6	6	6	6	6	6	6	6	6	6	6	6	6	7

n	57	58	59	60	61	62	63	64	65	66	67	68	69	...
$\mathbf{k_1}$	8	8	8	8	8	8	8	8	8	8	8	8	8	...
$\mathbf{k_2}$	7	7	7	7	7	7	7	7	7	7	7	7	7	...

10.10 Algoritmus křížení párů jedinců

Jakmile jsou v generaci vybráni jedinci kteří se mají účastnit společného křížení, tito jsou označeni jako rodič1 (parent1) a rodič2 (parent2). Cílem křížení je získat ze dvou výchozích implementací dvě nové implementace potomek1 (child1) a potomek2 (child2), které jsou genetickou kombinací těch dvou původních. Kombinují se tak jejich geneticky zakódované vlastnosti, které jsou rozhodující pro chování hledaných

implementací. Aby bylo možné zkřížit dva jedince, musí se v jejich buněčné reprezentaci určit, kde se bude nacházet bod křížení. Pro získání bodu křížení je třeba vygenerovat náhodné číslo v rozsahu, který je roven počtu buněk vektorů křížených implementací – 1. Je to proto, že bod křížení musí být zvolen tak, aby obě části z rozdělených binárních vektorů měly délku rovnu alespoň 1.

Pokud rodič1 je rozdělen náhodně na dvě části R1a a R1b a rodič2 je rozdělen na stejně velké části jako u R1a a R1b: R2a a R2b, potom potomek1 = R1a + R2b a potomek2 = R2a + R1b. Celková buněčná délka potomků bude stejná jako byla buněčná délka jejich rodičů. Na každého z vygenerovaných potomků jsou dále aplikovány operátory mutace, které jsou posány v následující kapitole.

10.11 Algoritmus aplikování mutačních operátorů na nové jedince

Nezbytnou součástí generování nových jedinců křížením je následné aplikování mutačních operátorů. V aplikaci GenAlg se používají dva typy mutačních operátorů, které se aplikují s určitou pravděpodobností pro každý bit binárního vektoru nového jedince:

- Aditivní mutace: Ke každé buňce jedince reprezentované 32bitovým číslem se s určitou pravděpodobností přičítá jeden bit s hodnotou jedna na náhodné pozici 0 – 31.
- Xorová mutace: U každé buňky jedince reprezentované 32bitovým číslem se s určitou pravděpodobností provede xor jednoho bitu s hodnotou jedna na náhodné pozici 0 – 31.

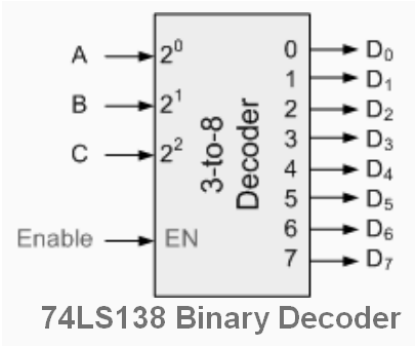
U aplikace GenAlg se pravděpodobnost obou typů mutací nastavuje nezávisle na sobě pomocí nastavitelných posuvníků „p-add.“ a „p-xor“ v rozsahu 0 – 100%. V případě, že máme v úmyslu měnit hodnotu těchto pravděpodobností, dokud evoluční proces běží, není možné pravděpodobnosti mutací měnit. Změnu lze provést až teprve po zastavení běhu evoluce tlačítkem “STOP”.

11. Příklady specifikací a nalezených implementací

11.1 Dekodér pro převod 3-bitové informace na 8-bitovou

Jedním z prvních algoritmů, který byl použit během vývoje aplikace GenAlg byla nadefinována specifikace pro dekodér pro převod 3-bitové informace na 8-bitovou. Na obrázku **Obr. 11.1.1** je zobrazen podobný dekodér, jako který je použit v tomto příkladu a jenž se běžně vyskytuje v digitální technice. Příklad byl zjednodušen a signál „enable“ zde není realizován.

Počet buněk pro jednu implementaci byl nastaven na 64 a bylo použito 1000 binárních vektorů pro hledání cílových implementací.



Obr. 11.1.1 Dekodér pro převod 3-bitové informace na 8-bitovou používaný v digitální technice

Byla připravena specifikace definující všechny případy vstupů a očekávaných výstupů pro dekodér 74LS138. Bylo očekáváno 8 iteračních kroků pro přechodný děj celé soustavy a během těchto kroků nejsou výsledné výstupy použity pro výpočet shody se zadáním ve specifikaci (až po jejich odeznění). 10 následujících iteračních kroků bylo nastaveno jako kritérium pro stanovení stabilních výstupů, u kterých se již shoda se specifikací vypočítává. Celkový počet shod výstupů s výstupy uvedenými ve specifikaci odpovídá celkovému počtu 704 bodů. (Obrázek **Obr. 11.1.2**).

```
// Maximum fitness is equal to:
// 8outputs * 8cases + 8outputs * 10steps of stability * 8cases = 704points
//3:case not repeated but reset of all cells
//20:no reset, but stability of outputs demanded(case repeated 10 times)
//status of the case, in mask, inputs, Nb of steps, out mask, outputs
3 h07 b00000000 w{ 8 } hff b00000001 // case 1 : 0
20 h07 b00000000 w{ 1 } hff b00000001 // case 1 : 0

3 h07 b00000001 w{ 8 } hff b00000010 // case 2 : 1
20 h07 b00000001 w{ 1 } hff b00000010 // case 2 : 1

3 h07 b00000010 w{ 8 } hff b00000100 // case 3 : 2
20 h07 b00000010 w{ 1 } hff b00000100 // case 3 : 2

3 h07 b00000011 w{ 8 } hff b00001000 // case 4 : 3
20 h07 b00000011 w{ 1 } hff b00001000 // case 4 : 3

3 h07 b00000100 w{ 8 } hff b00010000 // case 5 : 4
20 h07 b00000100 w{ 1 } hff b00010000 // case 5 : 4

3 h07 b00000101 w{ 8 } hff b00100000 // case 6 : 5
20 h07 b00000101 w{ 1 } hff b00100000 // case 6 : 5

3 h07 b00000110 w{ 8 } hff b01000000 // case 7 : 6
20 h07 b00000110 w{ 1 } hff b01000000 // case 7 : 6

3 h07 b00000111 w{ 8 } hff b10000000 // case 8 : 7
20 h07 b00000111 w{ 1 } hff b10000000 // case 8 : 7
```

Obr. 11.1.2 Specifikace dekodéru pro převod 3-bitové informace na 8-bitovou (popis zjednodušen oproti obsahu *.fdf souboru)

Poté co proces hledání běžel přibližně 5 hodin a proběhlo asi 9233 evolučních kroků, bylo dosaženo nejvyšší možné shody – 704 bodů shody. Po exportování jedné vybrané implementace, tato je zobrazena na obrázku **Obr. 11.1.3**

```
// |B2,B1,B0|F3,F2,F1,F0|y_||B10,B9,B8,B7,B6,B5,B4,B3||A2,A1,A0|F7,F6,F5,F4|y||A10,A9,A8,A7,A6,A5,A4,A3|
1.half:                                   2.half:
b00000000 b00000001 b00111110 b11111111 | b10011000 b11111101 b11000110 b00000010
b00100000 b00000111 b11111110 b00000000 | b10100010 b00000011 b01011010 b00000100
b01000000 b00000011 b01011110 b00000001 | b01010000 b00000110 b10001100 b00000001
b01111110 b11111011 b01111010 b00000000 | b01100000 b00000010 b10000000 b00000111
b00011110 b00000110 b11101010 b00000111 | b01010110 b00000001 b00010110 b00000011
b01111000 b00000110 b11011000 b00000100 | b11001110 b00000001 b11110100 b00000110
b10011010 b00000101 b01011100 b00000110 | b10100110 b00000100 b01101100 b00000001
b10100110 b00000101 b10111000 b00000001 | b11010000 b11111110 b00111100 b11111100
b00111010 b00000011 b01100110 b11111101 | b00000000 b00000000 b00001100 b00000000
b00011110 b11111010 b00011010 b00000110 | b11011110 b00000001 b01000000 b00000001
b11101100 b00000111 b11100110 b00000110 | b10100010 b00000101 b11001110 b00000111
b00011010 b11111111 b10101000 b00000001 | b01001010 b11111001 b01111000 b00000111
b10111010 b00000000 b01010100 b00000100 | b11110100 b00000101 b00101100 b00000111
b01001000 b11111001 b11010100 b00000011 | b11000100 b11111001 b01000000 b00000101
b11100110 b00000001 b00111010 b00000110 | b11110100 b00000101 b11011000 b00000000
b00100100 b00000000 b10110100 b00000010 | b00100010 b11111001 b10001110 b00000111
b10000110 b00000000 b10111110 b00000111 | b01000110 b00000010 b10011000 b00000111
b00110010 b00000110 b10001000 b00000011 | b10010110 b11111100 b11111110 b00000011
b01100100 b00000001 b00011000 b00000101 | b01000000 b00000000 b10010100 b00000110
b00011110 b00000111 b01111000 b00000110 | b00101000 b00000111 b00111010 b00000011
b10101000 b00000000 b00111010 b00000000 | b10000010 b00000111 b10111000 b00000111
b11001010 b00000001 b01101000 b00000110 | b10010000 b11111011 b01111100 b00000001
b11011100 b00000100 b11101100 b00000110 | b01010100 b00000100 b11100000 b00000011
b11001110 b00000001 b10110110 b00000111 | b11110000 b11111000 b01100110 b00000111
b00001010 b00000011 b00011110 b00000101 | b10000100 b11111101 b01110100 b11111111
b11111110 b00000010 b00111100 b00000101 | b10010000 b11111011 b00110110 b00000101
b11000010 b00000111 b00001110 b00000101 | b01010110 b11111101 b10110110 b00000010
b00011010 b00000000 b11010100 b00000011 | b11010100 b00000010 b11111100 b00000010
b10011100 b00000101 b11011010 b00000010 | b00101000 b11111011 b01111010 b11111110
b11000010 b11111101 b01101000 b00000010 | b00001000 b00000010 b00011010 b00000100
b01010010 b11111001 b01111100 b00000100 | b01010000 b00000000 b00111010 b00000100
b01100100 b11111100 b11100110 b11111111 | b00110000 b11111110 b00011100 b11111101
```

Obr. 11.1.3 Obsah exportované implementace, která se shoduje se specifikací

Pro prověření správného chování výsledného binárního vektoru (obrázek **Obr. 11.1.3**) bylo vygenerováno několik případů řešení výstupů celulárního procesoru

logických funkcí ze zadaných vstupů. Tyto případy jsou zobrazeny na obrázcích
Obr. 11.1.4-6. První 3 buňky z levé strany slouží jako 3 vstupy 3-to-8 bitového dekodéru. (v opačném pořadí bitů) a posledních 8 bitů je určeno pro výstupy (v běžném pořadí bitů). Prvních 8 iteračních cyklů má nestabilní výstupní hodnoty během přechodového děje a kroky 9 až 21 mají již stabilní výstupy:

```
 1 00000000000000000000000000000000000000000000000000000000000000000000000000000000
 2 00011011110110101101010111110110010011100111000111001000001000000
 3 00010001011000101001000011101001100010010010010111101001010000000000
 4 00011011110100101000110110111111100001100111000111010000000000000
 5 00010000111010101101000111101001100011010010110101010010101000000
 6 00011101110101101010100010111111000011110111100111010000100000000
 7 00010100111011101111101011101001000011010010000001010000100100001
 8 00011101110100001000101010111111100011110111100011000000100101010
 9 00010100111010001101100011101001100011010010000001010000000000001
10 00011011110100101000100010111111000011110111100111000000000000001
11 00010000111010101101100011101001000011010010000101010000000000001
12 00011011110100001000100010111111100011110111100111000000000000001
13 00010000111010001101100011101001100011010010000101010000000000001
14 00011011110100101000100010111111000011110111100111000000000000001
15 00010000111010101101100011101001000011010010000101010000000000001
16 00011011110100001000100010111111100011110111100111000000000000001
17 00010000111010001101100011101001100011010010000101010000000000001
18 00011011110100101000100010111111000011110111100111000000000000001
19 00010000111010101101100011101001000011010010000101010000000000001
20 00011011110100001000100010111111100011110111100111000000000000001
21 00010000111010001101100011101001100011010010000101010000000000001
```

Obr. 11.1.4 Prověření shody s případem č.1 ve specifikaci

```
 1 00000000000000000000000000000000000000000000000000000000000000000000000000000000
 2 00111011110110101001010111010110010011100111000101001000000100000
 3 00110001011000101001000011001100100010010010111101000010000000000
 4 00111010110100101100110111011010100001110111001101010000000000000
 5 00110001111010101101100111001000101001010010111010101001010000000
 6 00111100110101101010100111011110001011110111100101010000101000000
 7 00110101111011101011101111001100001011010010010001010000100100000
 8 00111100110100101110101111011010101011110111100001000000100100000
 9 00110101111010101101100111001000101011010010000001010000000100000
10 00111010110101101100100111011110001011110111100101000000000100000
11 00110001111011101101111001100001011010010000101010000000100000
12 00111010110100101100101111011010101011110111100101000000000100000
13 00110001111010101101100111001000101011010010000101010000000100000
14 00111010110101101100100111011110001011110111100101000000000100000
15 00110001111011101101111001100001011010010000101010000000100000
16 00111010110100101100101111011010101011110111100101000000000100000
17 00110001111010101101100111001000101011010010000101010000000100000
18 00111010110101101100100111011110001011110111100101000000000100000
19 00110001111011101101111001100001011010010000101010000000100000
20 00111010110100101100101111011010101011110111100101000000000100000
21 00110001111010101101100111001000101011010010000101010000000100000
```

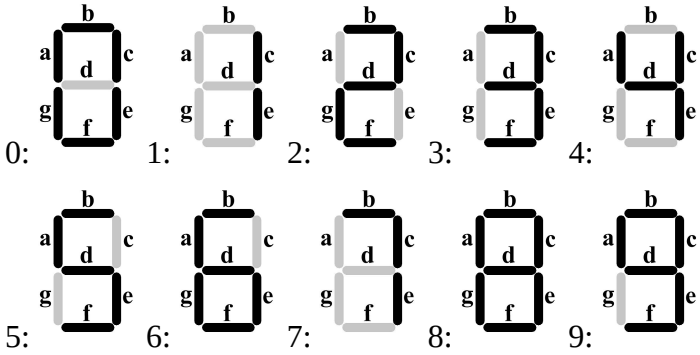
Obr. 11.1.5 Prověření shody s případem č. 5 ve specifikaci

```
1 00000000000000000000000000000000000000000000000000000000000000000000000000000000
2 01111011111110101001010111010100010011100111000101001000000100000
3 01100001010000101001000011001100100010010010111101000010000000000
4 01111010110000101100110111011000110001110111001101010000000000000
5 01100001010110101101100111001100101001110010110101010010010100000
6 01111100010001101001110111011000110110101101001010100000100000000
7 01110101110110101000101111001000001011110011010101010010011000000
8 01111100110000101101100111001000101011010110100001000000110000000
9 01110101110111101100100111001100101011110011110001010000010000000
10 01111010110001101001101111001100001011010110100001010000010000000
11 01100001110110101000101111001000001011110011110101010000001000000
12 01111010110000101101100111001000111011010110100101010000001000000
13 01100001010111101100100111001100101011110011110101010000001000000
14 01111010010001101001111111001100011011010110100101010000001000000
15 01100001110110101000101111001000001011110011110101010010010000000
16 01111010110000101101100111001000111011010110100101010000001000000
17 01100001010111101100100111001100101011110011110101010000001000000
18 01111010010001101001111111001100011011010110100101010000001000000
19 01100001110110101000101111001000001011110011110101010010010000000
20 01111010110000101101100111001000111011010110100101010000001000000
21 01100001010111101100100111001100101011110011110101010000001000000
```

Obr. 11.1.6 Prověření shody s případem č. 7 ve specifikaci

11.2 Dekodér číselné informace pro 7 segmentů

V tomto příkladu se hledá algoritmus (reprezentovaný binárním vektorem který je určen pro spouštění v celulárním procesoru logických funkcí), jenž popisuje chování logického obvodu dekodéru číselné informace zobrazované pomocí 7mi segmentů (čárkami LED zobrazovače, displeje) [23].



Obr. 11.2.1 Všechny kombinace segmentů dekadického LED dekóderu

Pro běh procesu evoluce během genetického programování byly zvoleny tyto parametry [23]:

- 8 paralelně běžících evolucí (na procesoru Intel CORE i5 – 4 jádra, 8 vláken),
- 2000 binárních vektorů v každé evoluci (celkem tedy 16000 jedinců),
- 64 buněk v každém vektoru,
- z definice hledaného algoritmu vyplývají 4 vstupní buňky pro kódování číslic 0 až 9 a 7 výstupních buněk pro jednotlivé LED segmenty,
- zvoleno 12 kroků celulárního procesoru logických funkcí pro ustálení hodnot na výstupních buňkách (pro odeznění přechodového děje),

- Vypočítaný počet shod, kterých musí být dosaženo, aby se chování algoritmu na 100% shodovalo s hledaným algoritmem, odpovídá číslu 770 bodů (= 11 cyklů pro ustálení hodnoty výstupů * 10 číslic které lze zobrazit * 7 LED segmentů).

Specifikace popisující dekodér číselné informace odpovídá tabulce **Tab. 11.2.1**, která vyplývá z kombinací segmentů, patrných z obrázku **Obr. 11.2.1** pro všechny zobrazované číslice **0** až **9**. Tyto jsou kódovány 4mi vstupy, namapovanými na začátek binárního vektoru na jeho první buňky. Jednotlivé segmenty **a** až **g** odpovídají 7mi výstupům, které jsou namapovány na konec binárního vektoru, na jeho poslední buňky.

Tab. 11.2.1 Přehled všech kombinací funkce číselného dekodéru

číslice 0 - 9	a	b	c	d	e	f	g
0	●	●	●		●	●	●
1			●		●		
2		●	●	●		●	●
3		●	●	●	●	●	
4	●		●	●	●		
5	●	●		●	●	●	
6	●	●		●	●	●	●
7		●	●		●		
8	●	●	●	●	●	●	●
9	●	●	●	●	●	●	

Po uplynutí přibližně 17400 evolučních cyklů byla nalezena cílová reprezentace hledaného algoritmu (byl vybrán pouze jeden vektor z mnoha jiných stejně kvalitních vektorů nalezených ve stejnou chvíli), kódována binárním vektorem spustitelným v celulárním procesoru logických funkcí a která splňuje 770 bodů shod s předlohou (čili 100%) a je znázorněna na obrázku **Obr. 11.2.2**

Na obrázku **Obr. 11.2.3** je znázorněno dynamické chování výsledného nalezeného algoritmu pro případ zobrazení čísla **5** na LED zobrazovači. Na začátku jsou všechny výstupy z buněk inicializovány na nulu a na první čtyři buňky je vystavena binární reprezentace čísla **5** (1010 – opačný zápis: v pořadí LSB první zleva; MSB poslední). Uběhne 12 cyklů (vybráno během definice hledaného algoritmu) nutných pro přechodný děj a poté jsou už výstupní hodnoty stabilní. LED segmenty **a** až **g** jsou mapovány na poslední buňky vektoru (1101110 : a b c d e f g). Všechny ostatní buňky nepatřící ani k vstupům a ani k výstupům, odpovídají vnitřním stavům algoritmu. Obdobně lze vygenerovat i ostatní kombinace pro zobrazení číslic **0**, **1**, **2**, **3**, **4**, **6**, **7**, **8** a **9**.

```
// |B2,B1,B0|F3,F2,F1,F0|y_||B10,B9,B8,B7,B6,B5,B4,B3||A2,A1,A0|F7,F6,F5,F4|y||A10,A9,A8,A7,A6,A5,A4,A3|
1  b00100000 b00000000 b01011110 b11111000 ' 33 b00011000 b00000101 b10010110 b00000101
2  b11000000 b00000000 b10011110 b11111111 ' 34 b00010000 b00000010 b01110110 b00000001
3  b01100000 b11111101 b10111110 b00000110 ' 35 b01000110 b00000011 b01101010 b00000001
4  b11000000 b00000111 b10011110 b00000100 ' 36 b11011000 b00000000 b01101110 b00000100
5  b00111000 b00000101 b10001000 b00000111 ' 37 b00000010 b00000110 b00100110 b00000111
6  b10110000 b00000111 b10001100 b00000110 ' 38 b10001010 b00000000 b11111110 b00000011
7  b10011110 b00000011 b10111000 b00000000 ' 39 b01111110 b00000000 b11101000 b00000001
8  b10000100 b00000011 b10110000 b00000001 ' 40 b10000000 b00000001 b01110010 b00000100
9  b01100100 b00000011 b10010000 b00000110 ' 41 b00000010 b00000111 b11101010 b00000100
10 b00110010 b00000111 b10011100 b00000110 ' 42 b00000000 b00000111 b11010110 b00000101
11 b01100000 b00000101 b00011100 b00000001 ' 43 b00100010 b11111001 b01000100 b00000101
12 b11110110 b00000101 b10101100 b00000101 ' 44 b01111100 b00000111 b01010100 b00000100
13 b00100100 b00000110 b00110010 b00000010 ' 45 b00110010 b11111001 b11110000 b00000010
14 b11000000 b00000010 b00010010 b00000111 ' 46 b10100010 b00000100 b10010010 b00000010
15 b01001110 b00000011 b00000000 b00000111 ' 47 b10001010 b00000101 b01101010 b00000100
16 b10101100 b00000101 b00010100 b00000001 ' 48 b10010110 b00000000 b01110110 b00000101
17 b01111000 b00000001 b10110110 b00000100 ' 49 b11111110 b00000101 b00110010 b00000010
18 b11110100 b11111101 b00101110 b00000000 ' 50 b11000110 b00000100 b11000000 b00000011
19 b11111100 b11111000 b00011110 b00000110 ' 51 b00100000 b11111001 b10001010 b00000101
20 b11110100 b00000101 b11110000 b00000010 ' 52 b11111110 b00000001 b10010000 b00000100
21 b00111100 b00000001 b11110010 b00000000 ' 53 b01001100 b00000001 b10111110 b00000010
22 b11101010 b00000011 b11010000 b00000101 ' 54 b01110010 b00000001 b01001000 b00000010
23 b00110000 b00000000 b10101000 b00000000 ' 55 b11111100 b00000010 b11010010 b00000111
24 b01101010 b00000111 b01110100 b00000100 ' 56 b10001000 b11111001 b11000110 b00000101
25 b11110100 b00000101 b00010000 b00000101 ' 57 b00010000 b00000100 b11000000 b00000011
26 b10011000 b00000010 b11011000 b00000010 ' 58 b00010000 b11111011 b10111110 b11111100
27 b01111100 b00000000 b00110010 b00000101 ' 59 b11110100 b11111000 b10011100 b00000000
28 b01011110 b00000010 b01001110 b00000111 ' 60 b00111010 b11111111 b01011010 b00000000
29 b10011010 b00000001 b11110110 b00000101 ' 61 b10000100 b11111110 b01111110 b00000110
30 b10001010 b00000100 b01111010 b00000100 ' 62 b00010010 b11111100 b10010110 b00000010
31 b01100010 b00000011 b00101010 b00000000 ' 63 b10011000 b11111101 b01100110 b00000000
32 b00010110 b11111010 b00101100 b00000010 ' 64 b01101000 b00000111 b00111100 b00000000
```

Obr. 11.2.2 Nalezený algoritmus shodující se na 100% s předloženou definicí - 64 buněk celulárního procesoru logických funkcí, popis všech pozic v záhlaví [23]

```
4inputs/ ................. algorithm ....................../7outputs
 1  1010 0010000100110011010110011011000101110101001111001010000 0010100
 2  1010 0010110100100110100001001000010001110001010000001001010 0000110
 3  1010 0000010000010011111011001101100000111100100101010000100 1 0100110
 4  1010 0010110000100110000100011001010001100000001010001001010 0100110
 5  1010 0000010100010011111011101100101000101000100101001010010 0101110
 6  1010 0010110000100110000100011001010001001001010000001000010 0101110
 7  1010 0000010100010011111011101100001000101000100101001010010 0101110
 8  1010 0010110000100110000100011011000001001001010010010000010 0101110
 9  1010 0000010110000001111011101100101000101000000010000101001 1101110
10  1010 0110110010110110000100011000010001001001010010010000010 1101110
11  1010 0010010110000001111011101101100000101000100101001010010 1101110
12  1010 0100110000110110000100011001010001001000010000001000010 1101110
                    ------ below are results with stable outputs------
13  1010 0110010100000001111011101100001000101000100101001010010 1101110
14  1010 0100110000110110000100011011000101001001010010010000010 1101110
15  1010 0110010110000001111011101100101000101000000010000101001 1101110
16  1010 0000110010110110000100011000010101001001010010010000010 1101110
17  1010 0110010110000001111011101100100000101000100101001010010 1101110
18  1010 0000110010110110000100011001010101001000001000000100001 1101110
19  1010 0110010110000001111011101101001000101000100101001010010 1101110
20  1010 0000110000110110000100011011000101001001010010010000010 1101110
21  1010 0010010110000001111011101100101000101000000010000101001 1101110
22  1010 0100110010110110000100011000010001001001010010010000010 1101110
```

Obr. 11.2.3 Nalezený algoritmus shodující se na 100% s předloženou definicí - přehled dynamického chování nalezeného algoritmu pro případ „číslo **5**“ [23]

Poznámka: Pro případ „**5**“ je zřetelně vidět, že k ustálení stavu výstupů došlo už dříve než po 12ti cyklech. To nemusí zdaleka platit i pro jiné případy.

11.3 Indukce číselné řady – násobení 3mi

Jako další příklad pro ověření správné funkčnosti párovacího systému byla zvolena indukce číselné řady [24]. Rozdílem oproti předchozím případům je to, že dle připravené specifikace se nejedná už jen o pouhý přenos, nebo konverzi informace z jednoho formátu do druhého, dochází zde ke složitějšímu procesu. Pokud například máme popsány ve specifikaci všechny možné případy, které mohou kdy nastat - formou tabulky všech možných vstupů a odpovídajících výstupů, není chování nalezené implementace tak překvapivé. Jakmile je taková implementace nalezena a její chování se shoduje ve všech popsaných bodech s chováním popsaným ve specifikaci, bude to jen prosté opakování nalezených vzorů. Mnohem zajímavější je však případ, kdy chování hledané implementace je ve specifikaci popsáno neúplným počtem případů a nalezená implementace se chová očekávaným a správným způsobem i pro ty případy, které původně ve specifikaci nebyly uvedeny. Tento mechanismus je znám jako princip indukce a skrývá se za ním již chování, které se vyznačuje určitou mírou inteligence. Toto chování nemá už nic společného s pouhým opakováním naučených vzorů.

Představme si systém, který by měl v sobě naprogramovánu velkou množinu různých početních operací. Systém by měl za úkol po předložení nějaké neznámé číselné řady odvodit předpis, kterým je řada vypočítána a následně dopočítat chybějící členy řady. Mohl by používat například strategii, že by zkoušel kombinovat různé početní operace ze své databáze, dokud by nenalezl správné řešení. Tímto způsobem by systém fungovat mohl. Potřeboval by k tomu mít ale naprogramovánu velkou databázi početních operací a mít schopnost je rychle spolu kombinovat a postupně ověřovat, jaké předpisy nejlépe na neznámou řadu pasují. Pravděpodobně i přesto by hledání trvalo velice dlouho a s nejistým výsledkem. Podobných číselných řad lze vytvořit nekonečné množství a dříve nebo později se vyskytne číselná řada, která obsahuje početní operace v databázi chybějící. U takové číselné řady by systém žádné řešení nenašel.

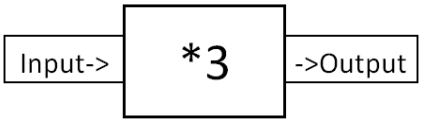
V našem případě v systému schopném indukce číselné řady, ale nemáme vůbec žádnou databázi početních operací a to se zdá být jako výhoda oproti předchozímu řešení. Databáze by stejně obsahovala vždy jen konečný počet operací oproti nekonečnému počtu různých číselných řad. Náš párovací systém pokoušející se o indukci číselné řady vlastně ani neví, že něco takového dělá a nekonečný počet různých případů řad zde nehraje žádnou roli. Vše funguje jen díky správné souhře několika na pouhé náhodě založených dějů při prohledávání celého stavového prostoru. Různé fáze procesu genetického programování jsou založeny na náhodě, přesto systém může díky selekčnímu tlaku dospět ke správnému řešení, odvodit předpis neznámé číselné řady a dopočítat její chybějící prvky. Identifikace řady, kde jsou prvky reprezentující násobení třemi, se může zdát být příkladem jednoduchým, náš párovací systém ale primárně žádné matematické výpočty neprovádí. Jednoduché i složitější případy řeší úplně stejným způsobem bez rozdílu a pokouší se jen nalézt procesem genetického programování tu implementaci, která by nejlépe odpovídala zadané specifikaci. Po spuštění celého procesu se tedy hledá nejpřesnější implementace odpovídající zadané specifikaci.

V našem případě je specifikace definována jako číselná řada vstupních hodnot od čísla 0 až po číslo 85. Všechny výstupní hodnoty uvedené v této specifikaci jsou vždy trojnásobkem číselné hodnoty na odpovídajícím vstupu. Nejvyšší hodnota na výstupu je

rovna 85 * 3 = 255 což na straně výstupů celulárního procesoru logických funkcí odpovídá 8mi výstupním bitům. Na straně vstupů pro vyjádření nejvyšší vstupní hodnoty rovné 85 postačí 7 bitů.

Aby zadání úlohy indukce číselné řady bylo úplné, 22 párů vstupních a výstupních hodnot je z celé řady hodnot vynecháno. Je tedy uvedeno jen 64 vstupních hodnot, ke kterým je přiřazeno 64 výstupních hodnot. Nalezená implementace má potom za úkol chybějící 22 prvky dopočítat. To odpovídá 25,5%tům hodnot z celé řady - více jak jedna čtvrtina hodnot tedy chybí.

Jako chybějící prvky specifikace byly zvoleny tyto páry hodnot: 10|30, 11|33, 16|48, 20|60, 22|66, 30|90, 31|93, 37|111, 38|114, 42|126, 47|141, 52|156, 57|171, 59|177, 61|183, 62|186, 68|204, 70|210, 72|216, 73|219, 79|237 a 81|243.



Obr. 11.3.1 Tento algoritmus není pro párovací systém znám a musí být nalezen, nebo spíše namodelován vnitřním chováním celulárního procesoru logických funkcí.

Tab. 11.3.1 Tabulka všech vstupních a výstupních hodnot včetně chybějících výstupů, které jsou zde označeny jako „?“

Input	Output
0	0
1	3
2	6
3	9
4	12
5	15
6	18
7	21
8	24
9	27
10	?
11	?
12	36
13	39
14	42
15	45
16	?
17	51
18	54
19	57
20	?
21	63

Input	Output
22	?
23	69
24	72
25	75
26	78
27	81
28	84
29	87
30	?
31	?
32	96
33	99
34	102
35	105
36	108
37	?
38	?
39	117
40	120
41	123
42	?
43	129

Input	Output
44	132
45	135
46	138
47	?
48	144
49	147
50	150
51	153
52	?
53	159
54	162
55	165
56	168
57	?
58	174
59	?
60	180
61	?
62	?
63	189
64	192
65	195

Input	Output
66	198
67	201
68	?
69	207
70	?
71	213
72	?
73	?
74	222
75	225
76	228
77	231
78	234
79	?
80	240
81	?
82	246
83	249
84	252
85	255

Pro běh procesu evoluce během genetického programování byly zvoleny tyto parametry:

- 8 paralelně běžících evolucí (na procesoru Intel CORE i7 – Ivy bridge, 8 vláken),
- 2000 binárních vektorů v každé evoluci (celkem tedy 16000 jedinců),

- 64 buněk v každém vektoru,
- z definice hledaného algoritmu vyplývá 7 vstupních buněk označených jako bit-x0 – bit-x6 pro kódování vstupních hodnot 0 - 85,
- Protože bylo obtížné nalézt řešení pro všechny výstupní bity y0-y7 popsané pouze jedním binárním vektorem, řešení bylo rozděleno na dvě části a poté řešeno odděleně ve dvou evolučních procesech:

  V prvním průběhu bylo hledáno řešení pro bity bit-y0 – bit-y4 výstupních hodnot 0 – 255 a tyto bity byly asociovány na 5 posledních buněk v hledaném binárním vektoru.

  Ve druhém průběhu bylo hledáno řešení pro bity bit-y4 – bit-y7 výstupních hodnot 0 – 255 a tyto bity byly asociovány na 4 poslední buňky v hledaném binárním vektoru.

  Z předchozích zkušeností vyplynulo, že čím více výstupních bitů bude z hledaného řešení použito pro hledání cílového binárního vektoru, tím je větší pravděpodobnost, že nalezené řešení bude spolehlivě fungovat. Při použití pouze tří bitů se po nalezení domnělého řešení zjistilo, že některá hledaná čísla (označená jako ?) byla špatně spočítána. Ničemu nevadí, když se potom ten bit, který se v řešení druhého evolučního procesu vyskytuje nadbytečně podruhé (bit-y4), nakonec nepoužije.
- Zvoleno 11 kroků celulárního procesoru logických funkcí pro ustálení hodnot na výstupních buňkách (pro odeznění přechodového děje), kdy výstupní hodnoty budou stabilní nejméně dalších 11 kroků.
- Vypočítaný počet shod, kterých musí být dosaženo, aby se chování algoritmu na 100% shodovalo s hledaným algoritmem, odpovídá pro první evoluční proces číslu 3520 bodů (= 11 cyklů pro ustálení hodnoty výstupů * 5 výstupních bitů * 64 vzorků číslic řady). Pro druhý evoluční proces odpovídá číslu 2816 bodů (= 11 cyklů pro ustálení hodnoty výstupů * 4 výstupní bity * 64 vzorků číslic řady).

Celý proces evoluce je možné ukončit ve chvíli, kdy počet bodů, ve kterých se momentálně vypočítané výstupy shodují s předem definovanými výstupy, dosáhne nejvyšší možné shody. V prvním průběhu evolučního procesu (myslí se tím celý dlouhý proces evoluce, ne pouze jeden cyklus) se podařilo nalézt shodu pro zadání výstupních bitů y0 – y4. Ve druhém průběhu se dosáhlo nejvyšší možné shody pro bity y5 – y7. Nejvyšší shoda pro bit y4 ve druhém průběhu dosažena nebyla, nebylo to ale nutné, tento výstup byl použit pouze pro podporu hledání řešení bitů y5 - y7. Jak se později ukázalo, nalezené vektory A i B (obrázek **Obr. 11.3.3** a **Obr. 11.3.4**) získané během prvního a druhého průběhu pro bity y0 – y7 postačily pro správné vyřešení všech případů pro prvky řady kde se nacházel „?“. Nyní je nejvyšší možná shoda pro vzorec y = 3*x splněna pro všechna celá čísla prvků řady na pozicích $x \in \langle 0; 85 \rangle$ včetně prvků původně v řadě chybějících.

Na obrázku **Obr. 11.3.2** jsou zobrazeny výstupy všech buněk u výsledných binárních vektorů A i B (prvního a druhého průběhu evolucí) pro kroky iterací 0 – 12 pokud je zvoleno zadání vstupů bitů x0 – x6 číslo 57. Z obrázku je zřejmé, že po 12ti krocích (v tomto případě už i dříve) se na výstupních bitech y0 – y7 při provádění algoritmu celulárního procesoru logických funkcí vypočítá číslo 171 které správně odpovídá rovnici 171 = 3 * 57. Obdobným způsobem bylo ověřeno, že binární vektory A i B lze použít ke správnému nalezení všech hodnot řady na pozicích $x \in \langle 0; 85 \rangle$.

STEP	bit-x0	bit-x1	bit-x2	bit-x3	bit-x4	bit-x5	bit-x6																								
y0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y1	1	0	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	1	1	1	1
y2	1	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0	0
y3	1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1
y4	1	0	0	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	0
y5	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	1	0	1
y6	1	0	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	1	0
y7	1	0	0	1	1	1	0	0	0	1	1	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	1	1	0	1
y8	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0
y9	1	0	0	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	1	0	1
y10	1	0	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0
y11	1	0	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	1	1	0	0	0	0	1	0	1	0	1	0	0	1
y12	1	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	0
Cell	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

STEP	bit-x0	bit-x1	bit-x2	bit-x3	bit-x4	bit-x5	bit-x6																								
y0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y1	1	0	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1
y2	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	0	1	1	0	0
y3	1	0	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	0	0	1	1	0	0	1	0	1	1	1	0
y4	1	0	0	1	1	1	0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	1	0	1	0	0	0	0	0	1	0	0
y5	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0	0	1	0	0
y6	1	0	0	1	1	1	0	1	0	1	1	1	0	0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0	0
y7	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1	0	1	0	0	1	1	0	0	1	0	0	1	0	0
y8	1	0	0	1	1	1	0	1	0	1	1	1	0	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	1	1	0	0
y9	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0
y10	1	0	0	1	1	1	0	1	0	1	1	1	0	0	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0	0
y11	1	0	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1	0	1	0	1	1	1	0	0	1	0	0	0	0	0
y12	1	0	0	1	1	1	0	1	0	1	1	1	0	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	1	1	0	0
	INPUT = 57: BIN 0111001																														

																											bit-y4	bit-y3	bit-y2	bit-y1	bit-y0	STEP
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	y0
1	1	1	0	1	1	1	0	0	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	1	0	0	0	0	1	0	1	y1
0	1	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0	0	0	0	0	1	1	y2
1	0	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0	1	0	1	0	1	1	y3
0	1	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	1	1	1	1	0	0	1	0	1	0	1	1	y4
1	0	1	0	1	0	1	0	0	1	0	1	1	0	1	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	1	y5
1	1	1	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1	1	1	1	1	0	0	0	1	0	1	0	1	1	y6
0	1	1	0	1	1	1	0	0	1	0	1	1	0	1	0	1	0	1	0	1	1	1	1	0	0	1	0	1	0	1	1	y7
1	0	1	0	0	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	1	0	1	0	1	1	1	y8
0	1	1	0	1	1	1	0	0	1	0	1	1	1	1	0	1	0	1	1	1	1	0	1	0	1	0	1	0	1	1	1	y9
1	0	1	0	0	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1	0	1	0	1	1	y10
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	0	1	0	1	1	1	1	1	0	1	0	1	0	1	1	y11
1	0	1	0	0	1	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	1	0	1	0	1	1	y12
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	Cell

																											bit-y7	bit-y6	bit-y5		STEP
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	y0
1	1	1	0	0	1	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	1	1	0	0	0	1	0	0	y1
1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	1	1	0	y2
1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0	y3
1	1	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1	1	0	0	0	0	0	1	0	y4
1	1	1	0	0	1	1	0	1	0	1	0	0	1	1	0	1	0	1	0	1	1	1	0	0	1	1	0	1	0	0	0	y5
1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	0	0	1	0	1	1	1	1	1	0	0	0	1	0	1	1	y6
1	1	1	0	0	1	1	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1	0	1	0	0	1	y7
1	1	1	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1	1	0	0	0	1	0	1	1	y8
1	1	1	0	0	1	1	0	1	0	0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1	0	1	0	1	1	y9
1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	0	1	0	1	1	y10
1	1	1	0	0	1	1	0	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	1	0	1	1	0	1	0	0	1	y11
1	1	1	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	0	1	0	1	1	y12

Obr. 11.3.2 Příklad 12ti krokové iterace, kdy na vstup je přivedeno číslo 57 a postupem iteračních kroků došly celulární procesory logických funkcí při použití binárních vektorů A i B na výstupech k hodnotě 171.

Inputs	address	ra	rb	0	1	2	3	4	5	6	7	FX	FX-HEX	logic function corresponding to Fx
bit-x0	0	40	4	0	0	0	0	1	1	1	1	240	F0	y_
bit-x1	1	63	23	0	0	0	0	1	1	1	1	240	F0	y_
bit-x2	2	-12	4	0	0	0	0	1	1	1	1	240	F0	y_
bit-x3	3	59	42	0	0	0	0	1	1	1	1	240	F0	y_
bit-x4	4	26	39	0	0	0	0	1	1	1	1	240	F0	y_
bit-x5	5	33	-38	0	0	0	0	1	1	1	1	240	F0	y_
bit-x6	6	59	53	0	0	0	0	1	1	1	1	240	F0	y_
	7	9	53	1	0	0	1	0	0	0	0	9	9	AND(EQUAL(ya_, yb_),NOT(y_))
	8	58	1	1	0	0	1	0	1	0	1	169	A9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=ya_
	9	21	18	0	1	0	1	0	0	1	1	202	CA	switch:Q_=0: y=ya_/Q_=1:y=yb_
	10	29	45	1	1	0	1	1	0	0	0	27	1B	RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
	11	22	-9	0	0	1	1	1	1	1	1	252	FC	OR(yb_, y_)
	12	22	50	1	0	0	1	1	0	1	1	217	D9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
	13	31	58	0	0	1	0	1	1	1	1	244	F4	OR(y_,NOT(yb_=>ya_))
	14	25	61	0	1	1	0	1	1	1	1	246	F6	OR(y_,XOR(ya_, yb_))
	15	9	57	0	0	0	0	1	1	0	1	176	B0	AND(y_,yb_=>ya_ (implikace))
	16	0	-10	0	1	0	0	0	0	1	0	66	42	EQUAL(ya_,yb_,NOT(y_))
	17	13	58	0	1	1	1	1	0	1	1	222	DE	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
	18	30	13	1	1	0	0	0	0	0	0	3	3	NOR(yb_, y_)
	19	57	48	0	1	0	1	1	0	1	1	218	DA	switch:Q_=0: y=ya_/Q_=1: y=OR(NOT(ya_),yb_)
	20	3	46	0	1	0	0	1	0	1	1	210	D2	XOR(NOT(ya_=>yb_),y_)
	21	32	-22	0	0	0	1	0	1	1	0	104	68	2 from 3 equal to 1
	22	-56	10	0	0	0	0	1	0	1	0	80	50	NOT(y_=>ya_)
	23	46	-51	1	0	0	0	1	0	1	1	209	D1	NOT(yb_=NOT(WRITE),ya_=data, y_=NOT(Q_))
	24	33	21	0	0	1	1	0	1	1	0	108	6C	NOT(RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
	25	53	-27	1	0	0	0	0	0	0	1	129	81	EQUAL(ya_,yb_,y_)
	26	39	0	1	0	0	1	1	1	1	0	121	79	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NAND(ya_,yb_)
	27	63	41	0	1	0	0	0	0	1	0	66	42	EQUAL(ya_,yb_,NOT(y_))
	28	17	44	1	0	0	1	0	1	1	0	105	69	XOR(EQUAL(ya_,yb_),y_)
	29	10	43	1	1	1	1	0	1	1	1	239	EF	OR(ya_,yb_,NOT(y_))
	30	48	58	1	1	0	0	0	1	1	0	99	63	switch:Q_=0: y=NOT(yb_)/Q_=1: y=XOR(ya_,yb_)
	31	60	45	1	1	1	0	0	0	0	0	7	7	AND(NAND(ya_, yb_),NOT(y_))
	32	62	15	1	1	1	1	1	0	0	0	31	1F	OR(NOR(ya_,yb_),NOT(y_))
	33	61	32	1	1	1	1	0	1	0	1	175	AF	y_=>ya_ (implikace)
	34	56	24	1	0	0	1	1	1	0	1	185	B9	RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_
	35	43	33	1	0	0	1	0	0	1	1	201	C9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=yb_
	36	0	31	1	0	1	1	1	0	1	0	93	5D	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=NOT(ya_)
	37	56	1	1	1	1	1	1	0	0	1	159	9F	OR(EQUAL(ya_,yb_),NOT(y_))
	38	29	-12	1	1	0	1	0	1	1	0	107	6B	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
	39	28	1	1	0	0	1	0	1	0	1	169	A9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=ya_
	40	36	15	0	0	1	0	0	0	0	0	4	4	NOR(ya_,NOT(yb_),y_)
	41	2	20	1	0	0	0	1	1	0	1	177	B1	RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=Q_
	42	45	27	1	1	0	1	0	0	0	1	139	8B	yb_=WRITE,ya_=data, y_=NOT(Q_)
	43	-28	9	1	0	1	0	1	1	0	0	53	35	switch:Q_=0: y=NOT(ya_)/Q_=1:y=NOT(yb_)
	44	14	3	1	1	0	1	1	1	1	1	251	FB	OR(ya_,NOT(yb_),y_)
	45	6	56	0	1	1	0	1	0	1	1	214	D6	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
	46	11	-5	1	1	1	1	0	0	1	0	79	4F	OR(NOT(yb_=>ya_),NOT(y_))
	47	-28	33	0	0	1	1	1	1	0	0	60	3C	XOR(yb_, y_)
	48	38	9	1	1	0	0	0	1	1	1	227	E3	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(ya_,yb_)
	49	47	36	0	0	0	0	0	1	1	1	224	E0	AND(y_,OR(ya_, yb_))
	50	2	58	1	0	0	1	1	1	0	1	185	B9	RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_
	51	50	22	1	1	1	1	1	0	1	0	95	5F	NAND(ya_, y_)
	52	19	-7	1	1	1	1	1	1	1	1	255	FF	CONST=1
	53	4	47	1	0	0	0	1	0	1	0	81	51	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOT(ya_)
	54	-23	54	1	1	0	0	0	0	1	1	195	C3	EQUAL(yb_, y_)
	55	34	45	1	0	0	0	0	1	1	0	97	61	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=XOR(ya_,yb_)
	56	38	28	1	0	0	0	0	1	1	1	225	E1	XOR(NOR(ya_,yb_),y_)
	57	51	53	0	1	1	0	1	0	0	0	22	16	only one from 3 equal to 1
Outputs	58	19	4	0	0	1	1	1	0	1	0	92	5C	NOT(RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=NOT(Q_))
bit-y4:	59	40	50	0	0	0	1	1	1	0	1	184	B8	yb_=WRITE,ya_=data, y_=Q_
bit-y3:	60	42	57	0	1	1	0	0	1	1	1	230	E6	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=OR(ya_,yb_)
bit-y2:	61	-27	-50	1	0	0	1	1	0	0	1	153	99	EQUAL(ya_, yb_)
bit-y1:	62	-36	2	1	0	0	1	1	0	0	1	153	99	EQUAL(ya_, yb_)
bit-y0:	63	59	1	1	0	1	1	0	0	1	1	205	CD	NOT(RS-flipflop, R-dominant, ya_:S, yb_:R,y_=Q_)

Obr. 11.3.3 Výsledný binární vektor A nalezený během prvního průběhu evoluce při hledání implementace pro výstupy y0 – y4 specifikace hledaného algoritmu

Inputs	address	ra	rb	0	1	2	3	4	5	6	7	FX	FX-HEX	logic function corresponding to Fx
bit-x0	0	4	6	0	0	0	0	1	1	1	1	240	F0	y_
bit-x1	1	48	-7	0	0	0	0	1	1	1	1	240	F0	y_
bit-x2	2	-29	53	0	0	0	0	1	1	1	1	240	F0	y_
bit-x3	3	41	48	0	0	0	0	1	1	1	1	240	F0	y_
bit-x4	4	43	19	0	0	0	0	1	1	1	1	240	F0	y_
bit-x5	5	-30	-11	0	0	0	0	1	1	1	1	240	F0	y_
bit-x6	6	24	14	0	0	0	0	1	1	1	1	240	F0	y_
	7	-3	-13	0	1	1	1	1	0	0	1	158	9E	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=EQUAL(ya_,yb_)
	8	-13	5	0	0	1	0	0	0	1	1	196	C4	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=yb_
	9	43	55	1	1	1	1	1	1	1	1	255	FF	CONST=1
	10	-21	16	0	1	0	1	1	0	1	0	90	5A	XOR(ya_, y_)
	11	42	39	0	1	1	1	0	0	0	0	14	E	AND(OR(ya_, yb_) ,NOT(y_))
	12	7	56	1	1	1	1	1	0	0	0	31	1F	OR(NOR(ya_,yb_),NOT(y_))
	13	6	30	0	0	0	1	0	0	0	1	136	88	AND(ya_, yb_)
	14	2	-10	1	1	1	1	0	0	1	1	207	CF	y_=>yb_ (implikace)
	15	-45	-16	1	0	0	1	1	0	0	1	153	99	EQUAL(ya_, yb_)
	16	40	54	1	1	0	0	0	1	0	1	163	A3	RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=NOT(Q_)
	17	19	57	0	1	1	1	1	1	1	0	126	7E	NOT EQUAL(ya_,yb_,y_)
	18	21	-15	1	1	0	1	0	1	1	1	235	EB	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=OR(ya_,yb_)
	19	3	37	0	0	0	1	0	0	1	0	72	48	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=AND(NOT(ya_),yb_)
	20	-48	12	0	0	1	1	1	0	0	0	28	1C	switch:Q_=0: y=yb_/Q_=1: y=NOR(ya_,yb_)
	21	58	58	0	1	0	1	0	1	0	1	170	AA	ya_
	22	26	34	0	0	1	1	0	1	1	0	108	6C	NOT(RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
	23	21	59	0	1	1	1	1	1	0	1	190	BE	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
	24	41	7	0	0	0	0	0	1	0	0	32	20	NOR(NOT(ya_),yb_,NOT(y_))
	25	41	25	0	1	0	0	0	1	1	0	98	62	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
	26	13	-38	1	1	1	1	0	0	0	0	15	F	NOT(y_)
	27	50	38	0	1	1	0	1	1	1	0	118	76	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NAND(ya_,yb_)
	28	33	30	0	1	1	0	0	1	0	0	38	26	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
	29	45	-14	1	0	1	0	1	1	1	0	117	75	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NAND(ya_,yb_)
	30	24	-48	1	0	1	0	0	0	0	0	5	5	NOR(ya_, y_)
	31	2	44	1	0	0	0	1	0	1	1	209	D1	NOT(yb_=NOT(WRITE),ya_=data, y_=NOT(Q_))
	32	42	38	1	1	0	0	1	1	1	0	115	73	switch:Q_=0: y=NOT(yb_)/Q_=1: y=NAND(ya_,yb_)
	33	-11	-18	1	1	1	1	1	1	0	1	191	BF	OR(ya_,NOT(yb_),NOT(yc_))
	34	-4	-31	0	0	1	0	1	1	1	0	116	74	NOT(yb_=WRITE,ya_=data, y_=NOT(Q_))
	35	13	-18	0	0	1	1	0	1	0	1	172	AC	switch:Q_=0: y=yb_/Q_=1: y=ya_
	36	-13	53	0	1	1	1	1	0	0	0	30	1E	XOR(OR(ya_, yb_),y_)
	37	16	38	1	1	0	1	1	0	0	0	27	1B	RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
	38	-46	7	1	1	1	1	1	0	0	1	159	9F	OR(EQUAL(ya_,yb_),NOT(y_))
	39	25	-38	0	0	0	1	1	1	1	1	248	F8	OR(y_, AND(ya_, yb_))
	40	60	15	0	0	0	1	1	1	1	1	248	F8	OR(y_, AND(ya_, yb_))
	41	-41	61	1	0	1	1	0	1	1	0	109	6D	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
	42	30	21	1	0	0	1	0	0	1	1	201	C9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=yb_
	43	-16	8	0	1	1	1	0	1	0	1	174	AE	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=ya_
	44	-18	47	0	1	0	1	0	1	0	1	170	AA	ya_
	45	50	28	0	1	1	0	1	0	1	0	86	56	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOT(ya_)
	46	52	-21	1	0	1	0	1	0	1	0	85	55	NOT(ya_)
	47	6	10	0	0	0	1	1	0	1	0	88	58	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=NOT(ya_)
	48	23	53	0	1	1	0	1	0	0	0	22	16	only one from 3 equal to 1
	49	35	24	0	1	0	0	0	1	0	0	34	22	NOT(ya_=>yb_)
	50	26	41	0	1	0	1	1	1	0	1	186	BA	RS-flipflop, S-dominant, ya_:S, yb_:R,y_=Q_
	51	13	13	0	1	1	0	0	0	1	0	70	46	NOT(RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_)
	52	2	43	1	1	0	0	0	1	1	0	99	63	switch:Q_=0: y=NOT(yb_)/Q_=1: y=XOR(ya_,yb_)
	53	46	-13	1	0	0	1	1	1	1	1	249	F9	OR(y_, EQUAL(ya_, yb_))
	54	21	34	0	1	1	0	1	1	0	0	54	36	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOT(yb_)
	55	38	12	1	1	0	1	0	0	0	0	11	B	AND(yb_=>ya_ ,NOT(y_))
	56	55	-12	1	1	1	0	1	1	1	0	119	77	NAND(ya_, yb_)
	57	-37	12	0	0	1	1	1	1	0	1	188	BC	NOT(RS-flipflop, R-dominant, ya_:R, yb_:S,y_=NOT(Q_))
	58	26	-20	0	1	0	1	0	0	0	0	10	A	NOT(ya_=>y_)
Outputs	59	34	10	0	0	0	0	1	1	0	1	176	B0	AND(y_,yb_=>ya_ (implikace))
bit-y7:	60	56	20	0	1	0	0	0	1	0	1	162	A2	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=ya_
bit-y6:	61	-16	-55	1	0	0	1	1	0	1	1	217	D9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
bit-y5:	62	17	7	1	0	0	1	1	0	0	1	153	99	EQUAL(ya_, yb_)
bit-y4-not used	63	-23	5	1	0	0	1	1	0	1	1	217	D9	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)

Obr. 11.3.4 Výsledný binární vektor B nalezený během druhého průběhu evoluce při hledání implementace pro výstupy y5 – y7 specifikace hledaného algoritmu.

11.4 Identifikace typů ASCII znaků ze 7mi binárních vstupů

Cílem hledané implementace je v tomto příkladě to, aby implementace dokázala spolehlivě identifikovat ASCII znaky ze 7mi binárních vstupů a přiřadit je do správné kategorie znaků podle tabulky uvedené v **Tab. 11.4.1.** Protože se zdálo, že hledání řešení pro všechny uvedené kategorie trvalo příliš dlouho a nebylo jisté, jestli se podaří nalézt řešení pro všechny kategorie, byl problém zjednodušen dle tabulky **Tab. 11.4.2** a závorky byly převedeny do kategorie „ostatní“.

Tab. 11.4.1 Původní rozdělení znaků základní ASCII tabulky pro identifikaci

řídící kódy	znaky mezery	matematické znaky	čísla	velká písmena	malá písmena	levé závorky	pravé závorky	ostatní
ASCII 0	ASCII 32	!	0	A	a	(	)	"
ASCII 1	_	%	1	B	b	[	]	#
ASCII 2		&	2	C	c	{	}	$
ASCII 3		*	3	D	d			‘
ASCII 4		+	4	E	e			?
ASCII 5		,	5	F	f			@
ASCII 6		-	6	G	g			\
ASCII 7		.	7	H	h			'
ASCII 8		/	8	I	i			ASCII 127
ASCII 9		:	9	J	j			
ASCII 10		;		K	k			
ASCII 11		<		L	l			
ASCII 12		=		M	m			
ASCII 13		>		N	n			
ASCII 14		^		O	o			
ASCII 15		|		P	p			
ASCII 16		~		Q	q			
ASCII 17				R	r			
ASCII 18				S	s			
ASCII 19				T	t			
ASCII 20				U	u			
ASCII 21				V	v			
ASCII 22				W	w			
ASCII 23				X	x			
ASCII 24				Y	y			
ASCII 25 - ASCII 31				Z	z			

Tab. 11.4.2 Rozdělení znaků ASCII tabulky pro identifikaci po zjednodušení

řídící kódy	znaky mezery	matematické znaky	čísla	velká písmena	malá písmena	ostatní
ASCII 0	ASCII 32	!	0	A	a	"
ASCII 1	_	%	1	B	b	#
ASCII 2		&	2	C	c	$
ASCII 3		*	3	D	d	‘
ASCII 4		+	4	E	e	?
ASCII 5		,	5	F	f	@
ASCII 6		-	6	G	g	\
ASCII 7		.	7	H	h	'
ASCII 8		/	8	I	i	ASCII 127
ASCII 9		:	9	J	j	(
ASCII 10		;		K	k	[
ASCII 11		<		L	l	{
ASCII 12		=		M	m	)
ASCII 13		>		N	n	]
ASCII 14		^		O	o	}
ASCII 15		|		P	p	
ASCII 16		~		Q	q	
ASCII 17				R	r	
ASCII 18				S	s	
ASCII 19				T	t	
ASCII 20				U	u	
ASCII 21				V	v	
ASCII 22				W	w	
ASCII 23				X	x	
ASCII 24				Y	y	
ASCII 25 - ASCII 31				Z	z	

Evoluční proces hledající implementaci pro identifikaci ASCII znaků genetickým programováním byl provozován s těmito parametry:

- 8 evolučních procesů běžících paralelně na CPU Intel CORE i7 Ivy bridge,
- 2000 binárních vektorů pro každou paralelní evoluci (16 000 celkem),
- 86 buněk v každém binárním vektoru,

- Vstupní data reprezentující ASCII kódy v rozsahu 0 – 127 jsou mapovány na 7 prvních buněk jako bit-x_0 – bit-x_6.
- Výstupy jsou sledovány na posledních 6ti buňkách bit-y_0 – bit-y_5, které odpovídají jednotlivým kategoriím: „řídící kódy“, „znaky mezery“, „matematické znaky“, „čísla“, „velká písmena“ a „malá písmena“. Každý z přidružených výstupů je nastaven na hodnotu 1 pokud je kategorie nalezena, jinak na hodnotu 0. Skupina „ostatní“ je signalizována tak, že všechny výstupy jsou nastaveny na hodnotu 0.
- Přechodový děj byl nastaven na 16 kroků aby po jejich odeznění byly výstupy stabilní alespoň 5 následujících kroků.
- Nejvyšší možná vypočítaná shoda se specifikací je rovna: 6 stabilních kroků * 6 výstupních bitů * 128 definic (vstup/výstup) = 4608 bodů shody.
- Během posledního pokusu pro evoluční hledání implementace, který trval přiližně 4 dny, byla nalezena nejvyšší možná shoda rovna 4596 bodů: Jen dvě definice nejsou splněny, znaky “^” a “_” jsou nalezenou implementací mylně přiřazeny ke skupině „ostatní“. Všechny další ASCII znaky z tabulky **Tab. 11.4.2** jsou identifikovány již správně, což pořád ještě vykazuje dobrou přesnost.

Na obrázcích **Obr. 11.4.1** a **Obr. 11.4.2** se nachází první a druhá část nalezené implementace.

Pro ověření správného chování nalezené implementace byly otestovány všechny ASCII znaky v rozsahu 0 – 127. Byly nalezeny pouze dvě chybné identifikace: znak „^“ který měl být identifikován jako matematický znak a „_“ měl být identifikován jako znak kategorie mezery. Místo toho byly přiřazeny do kategorie „ostatní“. Na obrázcích **Obr. 11.4.3 - 5.** jsou znázorněny příklady ověřených identifikací a jejich iterační kroky s přechodným dějem nastaveným na 16 kroků a na posledních 6ti buňkách jsou poté ustáleny stabilní výstupy bit-y0 – bit-y5 reprezentující identifikované kategorie.

	address	ra	rb	0	1	2	3	4	5	6	7	FX-HEX	function
Inputs	0	49	20	0	0	0	0	1	1	1	1	F0	y_
	1	61	-40	0	0	0	0	1	1	1	1	F0	y_
	2	-48	59	0	0	0	0	1	1	1	1	F0	y_
	3	48	59	0	0	0	0	1	1	1	1	F0	y_
	4	48	-1	0	0	0	0	1	1	1	1	F0	y_
	5	41	34	0	0	0	0	1	1	1	1	F0	y_
	6	15	1	0	0	0	0	1	1	1	1	F0	y_
	7	59	52	1	0	1	1	1	0	0	0	1D	NOT(yb_=NOT(WRITE),ya_=data, y_=Q_)
	8	34	10	1	1	0	0	1	0	0	1	93	RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
	9	48	7	0	1	0	0	1	1	1	1	F2	OR(y_,NOT(ya_=>yb_))
	10	55	38	1	1	0	1	1	0	1	1	DB	NOT(ya_=NOT(yb_)=yc_)
	11	10	33	0	1	1	1	0	1	0	0	2E	yb_=NOT(WRITE),ya_=data, y_=NOT(Q_)
	12	45	29	0	0	0	0	0	0	1	0	40	AND(y_,NOT(yb_=>ya_))
	13	47	34	1	1	1	1	1	1	1	0	7F	NAND(ya_,yb_,y_)
	14	29	38	0	0	1	1	1	1	1	0	7C	switch:Q_=0: y=yb_/Q_=1: y=NAND(ya_,yb_)
	15	4	43	0	1	0	0	0	0	0	0	2	NOR(NOT(ya_),yb_,y_)
	16	49	47	1	0	0	1	1	1	0	1	B9	RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_
	17	-7	50	0	0	0	0	0	0	1	0	40	AND(y_,NOT(yb_=>ya_))
	18	41	31	0	1	1	1	1	0	0	0	1E	XOR(OR(ya_, yb_),y_)
	19	51	47	0	1	1	1	0	1	1	1	EE	OR(ya_, yb_)
	20	25	8	0	1	1	0	0	0	1	1	C6	NOT(RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=Q_)
	21	12	11	1	0	0	1	1	0	0	1	99	EQUAL(ya_, yb_)
	22	17	-45	1	1	1	1	1	1	1	0	7F	NAND(ya_,yb_,y_)
	23	13	48	1	0	0	1	1	0	0	0	19	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NOR(ya_,yb_)
	24	25	49	0	0	0	1	1	0	0	0	18	EQUAL(ya_,yb_,NOT(y_))
	25	63	42	0	1	0	0	0	0	1	1	C2	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=yb_
	26	19	36	1	0	0	1	1	0	0	1	99	EQUAL(ya_, yb_)
	27	39	61	1	0	0	0	0	1	0	0	21	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
	28	51	42	1	1	1	0	0	0	0	1	87	XOR(NAND(ya_,yb_),y_)
	29	45	46	0	1	1	0	1	1	1	1	F6	OR(y_,XOR(ya_, yb_))
	30	56	-28	1	1	1	1	1	0	0	0	1F	OR(NOR(ya_,yb_),NOT(y_))
	31	62	60	1	0	0	0	0	1	1	1	E1	XOR(NOR(ya_,yb_),y_)
	32	17	19	1	1	0	1	1	1	0	1	BB	yb_=>ya_ (implikace) (NAND(NOT(ya_),yb)
	33	48	18	0	1	1	0	0	1	0	1	A6	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=ya_
	34	55	21	0	1	1	1	1	0	0	0	1E	XOR(OR(ya_, yb_),y_)
	35	45	26	0	0	1	1	1	1	0	0	3C	XOR(yb_, y_)
	36	22	62	0	1	0	1	1	1	1	0	7A	switch:Q_=0: y=ya_/Q_=1: y=NAND(ya_,yb_)
	37	-39	1	0	1	1	0	0	1	1	0	66	XOR(ya_, yb_)
	38	33	-8	0	0	1	0	0	1	1	1	E4	NOT(RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
	39	52	-17	0	1	0	0	1	0	1	1	D2	XOR(NOT(ya_=>yb_),y_)
	40	11	45	1	1	1	0	0	0	0	1	87	XOR(NAND(ya_,yb_),y_)
	41	18	46	1	0	1	1	1	1	1	0	7D	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=NAND(ya_,yb_)
	42	18	7	1	1	0	0	0	1	1	1	E3	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(ya_,yb_)
	43	33	45	0	1	0	1	1	0	0	1	9A	switch:Q_=0: y=ya_/Q_=1: y=EQUAL(ya_,yb_)

Obr. 11.4.1 První část výsledné nalezené implementace: buňky 0 – 43

	address	ra	rb	0	1	2	3	4	5	6	7	FX-HEX	function
	44	12	54	0	0	1	1	0	0	1	1	CC	yb_
	45	56	29	0	1	0	0	0	1	1	1	E2	yb_=NOT(WRITE),ya_=data, y_=Q_
	46	62	13	1	1	1	1	0	1	1	0	6F	OR(XOR(ya_,yb_),NOT(y_))
	47	9	52	0	1	0	1	1	0	1	1	DA	switch:Q_=0: y=ya_/Q_=1: y=OR(NOT(ya_),yb_)
	48	57	10	0	0	1	0	0	0	0	1	84	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=AND(ya_,yb_)
	49	10	53	0	0	1	0	1	0	1	0	54	NOT(RS-flipflop, S-dominant, ya_:S, yb_:R,y_=NOT(Q_))
	50	56	7	1	1	0	0	1	0	0	0	13	NOT(RS-flipflop, S-dominant, ya_:NOT(R), yb_:S,y_=Q_)
	51	33	36	1	1	1	1	1	0	0	1	9F	OR(EQUAL(ya_,yb_),NOT(y_))
	52	35	1	1	1	1	1	1	0	0	1	9F	OR(EQUAL(ya_,yb_),NOT(y_))
	53	20	13	1	0	0	0	0	0	0	0	1	NOR(ya_,yb_,y_)
	54	55	16	0	0	0	1	1	0	1	1	D8	ya_=WRITE,yb_=data, y_=Q_
	55	9	7	0	0	0	1	0	1	1	1	E8	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=OR(ya_,yb_)
	56	-49	35	0	0	0	1	0	1	0	1	A8	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=ya_
	57	33	54	1	0	0	1	0	0	1	0	49	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=AND(NOT(ya_),yb_)
	58	-24	35	0	1	0	0	0	0	1	1	C2	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=yb_
	59	36	16	0	1	1	1	1	0	1	0	5E	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOT(ya_)
	60	26	54	0	1	1	1	0	1	1	1	EE	OR(ya_, yb_)
	61	32	56	1	1	0	1	1	1	1	0	7B	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=NAND(ya_,yb_)
	62	13	30	1	1	1	0	1	1	0	0	37	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NOT(yb_)
	63	29	-6	1	0	1	0	1	1	1	1	F5	ya_=>y_ (implikace)
	64	7	57	0	1	0	0	0	1	0	1	A2	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=ya_
	65	40	60	0	1	1	0	0	1	0	0	26	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
	66	54	3	0	1	0	0	1	1	1	1	F2	OR(y_,NOT(ya_=>yb_))
	67	60	49	0	1	1	1	0	0	0	1	8E	ya_=NOT(WRITE),yb_=data, y_=NOT(Q_)
	68	59	38	1	0	1	0	1	0	1	0	55	NOT(ya_)
	69	31	55	1	1	1	0	1	1	0	1	B7	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
	70	5	50	1	1	0	0	0	1	1	1	E3	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(ya_,yb_)
	71	7	19	1	1	0	0	1	0	0	0	13	NOT(RS-flipflop, S-dominant, ya_:NOT(R), yb_:S,y_=Q_)
	72	55	41	1	0	1	0	1	1	0	1	B5	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NAND(NOT(ya_),yb_)
	73	50	-44	0	0	1	0	1	0	1	1	D4	NOT(RS-flipflop, D-dominant, ya_:S, yb_:R,y_=NOT(Q_))
	74	2	62	1	1	0	0	1	0	1	1	D3	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(NOT(ya_),yb_)
	75	17	12	0	0	1	0	0	1	1	1	E4	NOT(RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
	76	59	29	1	0	0	0	1	0	1	1	D1	NOT(yb_=NOT(WRITE),ya_=data, y_=NOT(Q_))
	77	52	4	0	1	0	0	1	0	0	1	92	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=EQUAL(ya_,yb_)
	78	24	38	1	0	1	0	1	0	1	0	55	NOT(ya_)
	79	47	60	0	1	0	1	1	1	1	1	FA	OR(ya_, y_)
Outputs	80	62	12	0	0	0	1	0	0	0	1	88	AND(ya_, yb_)
	81	-19	36	0	0	1	0	0	0	1	1	C4	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=yb_
	82	27	26	0	1	0	0	1	1	0	1	B2	RS-flipflop, D-dominant, ya_:S, yb_:R,y_=Q_
	83	-25	50	0	0	0	1	1	0	1	1	D8	ya_=WRITE,yb_=data, y_=Q_
	84	25	50	0	0	1	0	1	1	1	0	74	NOT(yb_=WRITE,ya_=data, y_=NOT(Q_))
	85	63	-24	0	1	0	0	1	1	1	1	F2	OR(y_,NOT(ya_=>yb_))

Obr. 11.4.2 Druhá část výsledné nalezené implementace: buňky 44 – 85

STEP	bit-x0	bit-x1	bit-x2	bit-x3	bit-x4	bit-x5	bit-x6																																						
y0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y1	1	0	0	0	1	0	0	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1	0
y2	1	0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	1	1	0	1	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	0	1	1	1
y3	1	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	0
y4	1	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	1	1	1	1
y5	1	0	0	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	0	1	0	1	1	1	1	0	0	0	0	1	0	0	1	1	1	1
y6	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0	1	0	1	1	1	1
y7	1	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	1
y8	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	0	0	0	0	1	0	1	1	1	1
y9	1	0	0	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	0	0	1	1	1	1
y10	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	1	1	1
y11	1	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	1
y12	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	0	0	0	0	1	0	1	1	1	1
y13	1	0	0	0	1	0	0	1	0	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	0	0	1	1	1	1
y14	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	1	1	1
y15	1	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	1	1	0	0	1	0	1	0	0	1	1	1	1
y16	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	0	0	0	0	1	0	1	1	1	1
Cell	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	INPUT:			"DC1-XOn"					ASCII 17																																		

																																				bit-y5 small letters	bit-y4 capitals	bit-y3 numbers	bit-y2 math chars	bit-y1 space	bit-y0 control codes	STEP
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	y0
0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	y1
0	0	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	0	1	0	1	0	0	1	0	0	0	0	0	0	y2
0	0	1	0	0	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	1	0	0	0	0	0	1	y3
0	0	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	0	1	y4
0	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1	y5
0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y6
0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	y7
0	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y8
0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	y9
0	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y10
0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	y11
0	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y12
0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	y13
0	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y14
0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	y15
0	1	1	0	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0	1	y16
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	Cell
																																		OUTPUT:			control code					

Obr. 11.4.3 Verifikace nalezené implementace u ASCII 17, kategorie řídící kód

STEP	bit-x0	bit-x1	bit-x2	bit-x3	bit-x4	bit-x5	bit-x6																																					
y0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y1	0	0	1	0	1	1	0	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	0
y2	0	0	1	0	1	1	0	1	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	0	1
y3	0	0	1	0	1	1	0	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	1	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0	0	1	1	0	1
y4	0	0	1	0	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	0	0	1	1	0	1	0	0	1	0	0	1	0	1
y5	0	0	1	0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	1	0	1	1	1	1	0	1	0	0	0	1	1	0	1	0	0	0	0	1	0	0	1	1	0	0
y6	0	0	1	0	1	1	0	1	0	1	0	1	0	1	1	0	1	0	1	1	0	0	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	0
y7	0	0	1	0	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0	0	1	0	1	1	0	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	1
y8	0	0	1	0	1	1	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1	1	0	1	1	0	0	0	1	0	1	0	0	1	0	0	1	1	1
y9	0	0	1	0	1	1	0	0	0	1	1	1	0	0	1	1	0	0	1	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	1	0	0	1	1	1	1
y10	0	0	1	0	1	1	0	1	0	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	1
y11	0	0	1	0	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	1
y12	0	0	1	0	1	1	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0	1	0	1	0	0	1	0	0	1	1	1
y13	0	0	1	0	1	1	0	0	0	1	1	1	0	0	1	1	0	0	1	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	1	0	0	1	1	1	1
y14	0	0	1	0	1	1	0	1	0	1	0	0	0	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	1	1	1
y15	0	0	1	0	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0	0	1	0	1	1	0	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	1
y16	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**
Cell	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	INPUT:			"4"		ASCII 52																																						

																																				bit-y5 small letters	bit-y4 capitals	bit-y3 numbers	bit-y2 math chars	bit-y1 space	bit-y0 control codes	STEP
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	y0
0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	y1
0	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	1	1	1	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	0	1	0	0	0	0	0	0	y2
0	0	0	1	0	1	0	1	1	1	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	1	0	1	1	1	1	0	0	1	0	0	0	y3
0	0	1	1	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	1	0	0	0	0	0	1	0	0	1	0	0	0	y4
0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	0	0	0	1	1	1	0	0	1	0	0	0	y5
0	1	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1	1	1	1	0	0	1	0	0	0	y6
0	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	1	1	0	1	1	0	0	1	1	0	1	0	0	1	0	0	0	y7
0	1	1	1	0	0	0	1	1	0	1	0	0	0	0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	0	1	1	0	0	1	0	0	0	y8
0	0	0	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0	y9
0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	1	0	0	0	y10
0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	0	0	0	y11
0	1	1	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	0	1	1	0	0	1	0	0	0	y12
0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	0	0	0	y13
0	1	1	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	1	0	0	0	y14
0	1	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1	0	0	0	y15
0	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	y16
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	Cell
																																				OUTPUT:		number				

Obr. 11.4.4 Verifikace nalezené implementace u ASCII 52, kategorie číslo

STEP	bit-x0	bit-x1	bit-x2	bit-x3	bit-x4	bit-x5	bit-x6																																						
y0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y1	1	1	0	0	1	1	1	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	1	0
y2	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	0	1	1	1
y3	1	1	0	0	1	1	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0	0	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	0
y4	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	1	1	0	1	1	1	0	0	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y5	1	1	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y6	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y7	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	1	1	0	1	0	1	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y8	1	1	0	0	1	1	1	1	1	0	1	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y9	1	1	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	0	1	0	1	1	1	1	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y10	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y11	1	1	0	0	1	1	1	1	1	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y12	1	1	0	0	1	1	1	1	1	0	1	1	0	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y13	1	1	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	0	1	0	1	1	1	1	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y14	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
y15	1	1	0	0	1	1	1	1	1	0	0	0	0	1	0	1	1	0	1	0	1	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1
y16	1	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1
Cell	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
	INPUT:			"s"		ASCII 115																																						

																																					bit-y5 small letters	bit-y4 capitals	bit-y3 numbers	bit-y2 math chars	bit-y1 space	bit-y0 control codes	STEP
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	y0
0	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	y1
0	0	1	0	0	1	1	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	0	1	0	0	0	0	0	0	y2
0	0	1	1	0	1	0	1	0	1	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0	1	1	0	1	1	0	0	0	0	0	y3
0	0	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	y4
0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	y5
0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y6
0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	y7
0	1	1	0	0	1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y8
0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	y9
0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y10
0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	y11
0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y12
0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	y13
0	1	1	0	0	1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y14
0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	y15
0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0	0	0	y16
44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	Cell
																																				OUTPUT:	small letter					

Obr. 11.4.5 Verifikace nalezené implementace u ASCII 115 kategorie malé písmeno

12. Úplný seznam funkcí F_n

V této kapitole se nachází tabulka s přehledem všech možných funkcí, $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$, které se mohou objevovat v konfiguracích nalezených implementací. Pomocí nich je možné převádět již nalezené implementace do rovnic Booleovy algebry. Vstupy y_, ya_ a yb_ odpovídají výstupům z minulého iteračního cyklu celulárního procesoru logických funkcí u své vlastní buňky a u buněk A i B z jejího okolí. Hodnoty $f_{n,0}$ až $f_{n,7}$ odpovídají nové výstupní hodnotě y pro buňku na pozici n.

Tab. 12. 1 Úplný seznam všech logických funkcí F_n

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$** poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
yb_	yb_=1		yb_=0		yb_=1		yb_=0		
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
0	0	0	0	0	0	0	0	0	CONST=0
1	0	0	0	0	0	0	0	1	NOR(ya_,yb_,y_)
2	0	0	0	0	0	0	1	0	NOR(NOT(ya_),yb_,y_)
3	0	0	0	0	0	0	1	1	NOR(yb_, y_)
4	0	0	0	0	0	1	0	0	NOR(ya_,NOT(yb_),y_)
5	0	0	0	0	0	1	0	1	NOR(ya_, y_)
6	0	0	0	0	0	1	1	0	AND(XOR(ya_, yb_),NOT(y_))
7	0	0	0	0	0	1	1	1	AND(NAND(ya_, yb_),NOT(y_))
8	0	0	0	0	1	0	0	0	AND(ya_,yb_,NOT(y_))
9	0	0	0	0	1	0	0	1	AND(EQUAL(ya_, yb_),NOT(y_))
10	0	0	0	0	1	0	1	0	NOT(ya_=>y_)
11	0	0	0	0	1	0	1	1	AND(yb_=>ya_ ,NOT(y_))
12	0	0	0	0	1	1	0	0	NOT(yb_=>y_)
13	0	0	0	0	1	1	0	1	AND(ya_=>yb_ ,NOT(y_))
14	0	0	0	0	1	1	1	0	AND(OR(ya_, yb_) ,NOT(y_))
15	0	0	0	0	1	1	1	1	NOT(y_)
16	0	0	0	1	0	0	0	0	NOR(ya_,yb_,NOT(y_))
17	0	0	0	1	0	0	0	1	NOR(ya_, yb_)
18	0	0	0	1	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=NOR(ya_,yb_)
19	0	0	0	1	0	0	1	1	NOT(RS-flipflop, S-dominant, ya_:NOT(R), yb_:S,y_=Q_)
20	0	0	0	1	0	1	0	0	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=NOR(ya_,yb_)
21	0	0	0	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NOR(ya_,yb_)
22	0	0	0	1	0	1	1	0	only one from 3 equal to 1

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
23	0	0	0	1	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NOR(ya_,yb_)
24	0	0	0	1	1	0	0	0	EQUAL(ya_,yb_,NOT(y_))
25	0	0	0	1	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NOR(ya_,yb_)
26	0	0	0	1	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=NOR(ya_,yb_)
27	0	0	0	1	1	0	1	1	RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
28	0	0	0	1	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=NOR(ya_,yb_)
29	0	0	0	1	1	1	0	1	NOT(yb_=NOT(WRITE),ya_=data, y_=Q_)
30	0	0	0	1	1	1	1	0	XOR(OR(ya_, yb_),y_)
31	0	0	0	1	1	1	1	1	OR(NOR(ya_,yb_),NOT(y_))
32	0	0	1	0	0	0	0	0	NOR(NOT(ya_),yb_,NOT(y_))
33	0	0	1	0	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
34	0	0	1	0	0	0	1	0	NOT(ya_=>yb_)
35	0	0	1	0	0	0	1	1	NOT(RS-flipflop, S-dominant, ya_:R, yb_:S,y_=Q_)
36	0	0	1	0	0	1	0	0	ya_=NOT(yb_)=yc_
37	0	0	1	0	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NOR(NOT(ya_),yb_)
38	0	0	1	0	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
39	0	0	1	0	0	1	1	1	NOT(ya_=WRITE,yb_=data, y_=Q_)
40	0	0	1	0	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
41	0	0	1	0	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
42	0	0	1	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=NOR(NOT(ya_),yb_)
43	0	0	1	0	1	0	1	1	RS-flipflop, D-dominant, ya_:S, yb_:R,y_=NOT(Q_)
44	0	0	1	0	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=NOR(NOT(ya_),yb_)
45	0	0	1	0	1	1	0	1	XOR(ya_=>yb_,y_)
46	0	0	1	0	1	1	1	0	yb_=NOT(WRITE),ya_=data, y_=NOT(Q_)
47	0	0	1	0	1	1	1	1	OR(NOT(ya_=>yb_),NOT(y_))
48	0	0	1	1	0	0	0	0	NOT(y_=>yb_)
49	0	0	1	1	0	0	0	1	RS-flipflop, R-dominant, ya_:NOT(S), yb_:R,y_=Q_
50	0	0	1	1	0	0	1	0	RS-flipflop, R-dominant, ya_:S, yb_:R,y_=Q_
51	0	0	1	1	0	0	1	1	NOT(yb_)
52	0	0	1	1	0	1	0	0	RS-flipflop, R-dominant, ya_:R, yb_:S,y_=Q_
53	0	0	1	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1:y=NOT(yb_)
54	0	0	1	1	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOT(yb_)
55	0	0	1	1	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NOT(yb_)
56	0	0	1	1	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=NOT(yb_)
57	0	0	1	1	1	0	0	1	RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=Q_
58	0	0	1	1	1	0	1	0	RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=Q_
59	0	0	1	1	1	0	1	1	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=NOT(yb_)
60	0	0	1	1	1	1	0	0	XOR(yb_, y_)

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
61	0	0	1	1	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=NOT(yb_)
62	0	0	1	1	1	1	1	0	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=NOT(yb_)
63	0	0	1	1	1	1	1	1	NAND(yb_, y_)
64	0	1	0	0	0	0	0	0	AND(y_,NOT(yb_=>ya_))
65	0	1	0	0	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=AND(NOT(ya_),yb_)
66	0	1	0	0	0	0	1	0	EQUAL(ya_,yb_,NOT(y_))
67	0	1	0	0	0	0	1	1	RS-flipflop, R-dominant, ya_:R, yb_:S,y_=NOT(Q_)
68	0	1	0	0	0	1	0	0	NOT(yb_=>ya_) (AND(NOT(ya_),yb_))
69	0	1	0	0	0	1	0	1	NOT(RS-flipflop, S-dominant, ya_:S, yb_:R,y_=Q_)
70	0	1	0	0	0	1	1	0	NOT(RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_)
71	0	1	0	0	0	1	1	1	NOT(yb_=WRITE,ya_=data, y_=Q_)
72	0	1	0	0	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=AND(NOT(ya_),yb_)
73	0	1	0	0	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=AND(NOT(ya_),yb_)
74	0	1	0	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=AND(NOT(ya_),yb_)
75	0	1	0	0	1	0	1	1	XOR(yb_=>ya_,y_)
76	0	1	0	0	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=AND(NOT(ya_),yb_)
77	0	1	0	0	1	1	0	1	NOT(RS-flipflop, D-dominant, ya_:S, yb_:R,y_=Q_)
78	0	1	0	0	1	1	1	0	NOT(RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=Q_)
79	0	1	0	0	1	1	1	1	OR(NOT(yb_=>ya_),NOT(y_))
80	0	1	0	1	0	0	0	0	NOT(y_=>ya_)
81	0	1	0	1	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOT(ya_)
82	0	1	0	1	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=NOT(ya_)
83	0	1	0	1	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=NOT(ya_)
84	0	1	0	1	0	1	0	0	NOT(RS-flipflop, S-dominant, ya_:S, yb_:R,y_=NOT(Q_))
85	0	1	0	1	0	1	0	1	NOT(ya_)
86	0	1	0	1	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NOT(ya_)
87	0	1	0	1	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NOT(ya_)
88	0	1	0	1	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=NOT(ya_)
89	0	1	0	1	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NOT(ya_)
90	0	1	0	1	1	0	1	0	XOR(ya_, y_)
91	0	1	0	1	1	0	1	1	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=NOT(ya_)
92	0	1	0	1	1	1	0	0	NOT(RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=NOT(Q_))
93	0	1	0	1	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=NOT(ya_)
94	0	1	0	1	1	1	1	0	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=NOT(ya_)
95	0	1	0	1	1	1	1	1	NAND(ya_, y_)
96	0	1	1	0	0	0	0	0	AND(y_,XOR(ya_, yb_))
97	0	1	1	0	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=XOR(ya_,yb_)

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
98	0	1	1	0	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
99	0	1	1	0	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=XOR(ya_,yb_)
100	0	1	1	0	0	1	0	0	NOT(RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
101	0	1	1	0	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=XOR(ya_,yb_)
102	0	1	1	0	0	1	1	0	XOR(ya_, yb_)
103	0	1	1	0	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=XOR(ya_,yb_)
104	0	1	1	0	1	0	0	0	2 from 3 equal to 1
105	0	1	1	0	1	0	0	1	XOR(EQUAL(ya_,yb_),y_)
106	0	1	1	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=XOR(ya_,yb_)
107	0	1	1	0	1	0	1	1	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
108	0	1	1	0	1	1	0	0	NOT(RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
109	0	1	1	0	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=XOR(ya_,yb_)
110	0	1	1	0	1	1	1	0	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=XOR(ya_,yb_)
111	0	1	1	0	1	1	1	1	OR(XOR(ya_,yb_),NOT(y_))
112	0	1	1	1	0	0	0	0	AND(y_,NAND(ya_, yb_))
113	0	1	1	1	0	0	0	1	NOT(ya_=NOT(WRITE),yb_=data, y_=NOT(Q_))
114	0	1	1	1	0	0	1	0	NOT(ya_=WRITE,yb_=data, y_=NOT(Q_))
115	0	1	1	1	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=NAND(ya_,yb_)
116	0	1	1	1	0	1	0	0	NOT(yb_=WRITE,ya_=data, y_=NOT(Q_))
117	0	1	1	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NAND(ya_,yb_)
118	0	1	1	1	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NAND(ya_,yb_)
119	0	1	1	1	0	1	1	1	NAND(ya_, yb_)
120	0	1	1	1	1	0	0	0	XOR(AND(ya_,yb_),y_)
121	0	1	1	1	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=NAND(ya_,yb_)
122	0	1	1	1	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=NAND(ya_,yb_)
123	0	1	1	1	1	0	1	1	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=NAND(ya_,yb_)
124	0	1	1	1	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=NAND(ya_,yb_)
125	0	1	1	1	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=NAND(ya_,yb_)
126	0	1	1	1	1	1	1	0	NOT EQUAL(ya_,yb_,y_)
127	0	1	1	1	1	1	1	1	NAND(ya_,yb_,y_)
128	1	0	0	0	0	0	0	0	AND(ya_,yb_,y_)
129	1	0	0	0	0	0	0	1	EQUAL(ya_,yb_,y_)
130	1	0	0	0	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=AND(ya_,yb_)
131	1	0	0	0	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=AND(ya_,yb_)
132	1	0	0	0	0	1	0	0	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=AND(ya_,yb_)
133	1	0	0	0	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=AND(ya_,yb_)

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
134	1	0	0	0	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=AND(ya_,yb_)
135	1	0	0	0	0	1	1	1	XOR(NAND(ya_,yb_),y_)
136	1	0	0	0	1	0	0	0	AND(ya_, yb_)
137	1	0	0	0	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=AND(ya_,yb_)
138	1	0	0	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=AND(ya_,yb_)
139	1	0	0	0	1	0	1	1	yb_=WRITE,ya_=data, y_=NOT(Q_)
140	1	0	0	0	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=AND(ya_,yb_)
141	1	0	0	0	1	1	0	1	ya_=WRITE,yb_=data, y_=NOT(Q_)
142	1	0	0	0	1	1	1	0	ya_=NOT(WRITE),yb_=data, y_=NOT(Q_)
143	1	0	0	0	1	1	1	1	OR(AND(ya_,yb_),NOT(y_))
144	1	0	0	1	0	0	0	0	AND(y_,EQUAL(ya_, yb_))
145	1	0	0	1	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=EQUAL(ya_,yb_)
146	1	0	0	1	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=EQUAL(ya_,yb_)
147	1	0	0	1	0	0	1	1	RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
148	1	0	0	1	0	1	0	0	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=EQUAL(ya_,yb_)
149	1	0	0	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=EQUAL(ya_,yb_)
150	1	0	0	1	0	1	1	0	XOR(XOR(ya_,yb_),y_)
151	1	0	0	1	0	1	1	1	NOT(2 from 3 equal to 1)
152	1	0	0	1	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=EQUAL(ya_,yb_)
153	1	0	0	1	1	0	0	1	EQUAL(ya_, yb_)
154	1	0	0	1	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=EQUAL(ya_,yb_)
155	1	0	0	1	1	0	1	1	RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_)
156	1	0	0	1	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=EQUAL(ya_,yb_)
157	1	0	0	1	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=EQUAL(ya_,yb_)
158	1	0	0	1	1	1	1	0	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=EQUAL(ya_,yb_)
159	1	0	0	1	1	1	1	1	OR(EQUAL(ya_,yb_),NOT(y_))
160	1	0	1	0	0	0	0	0	AND(ya_, y_)
161	1	0	1	0	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=ya_
162	1	0	1	0	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=ya_
163	1	0	1	0	0	0	1	1	RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=NOT(Q_)
164	1	0	1	0	0	1	0	0	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=ya_
165	1	0	1	0	0	1	0	1	EQUAL(ya_, y_)
166	1	0	1	0	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=ya_
167	1	0	1	0	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=ya_
168	1	0	1	0	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=ya_
169	1	0	1	0	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=ya_
170	1	0	1	0	1	0	1	0	ya_
171	1	0	1	0	1	0	1	1	RS-flipflop, S-dominant, ya_:S, yb_:R,y_=NOT(Q_)

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
172	1	0	1	0	1	1	0	0	switch:Q_=0: y=yb_/Q_=1: y=ya_
173	1	0	1	0	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=ya_
174	1	0	1	0	1	1	1	0	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=ya_
175	1	0	1	0	1	1	1	1	y_=>ya_ (implikace)
176	1	0	1	1	0	0	0	0	AND(y_,yb_=>ya_ (implikace))
177	1	0	1	1	0	0	0	1	RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=Q_
178	1	0	1	1	0	0	1	0	RS-flipflop, D-dominant, ya_:S, yb_:R,y_=Q_
179	1	0	1	1	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
180	1	0	1	1	0	1	0	0	EQUAL(yb_=>ya_,y_)
181	1	0	1	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=NAND(NOT(ya_),yb_)
182	1	0	1	1	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
183	1	0	1	1	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
184	1	0	1	1	1	0	0	0	yb_=WRITE,ya_=data, y_=Q_
185	1	0	1	1	1	0	0	1	RS-flipflop, S-dominant, ya_:NOT(S), yb_:R,y_=Q_
186	1	0	1	1	1	0	1	0	RS-flipflop, S-dominant, ya_:S, yb_:R,y_=Q_
187	1	0	1	1	1	0	1	1	yb_=>ya_ (implikace) (NAND(NOT(ya_),yb)
188	1	0	1	1	1	1	0	0	NOT(RS-flipflop, R-dominant, ya_:R, yb_:S,y_=NOT(Q_))
189	1	0	1	1	1	1	0	1	NOT(ya_=yb_=NOT(yc_))
190	1	0	1	1	1	1	1	0	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=NAND(NOT(ya_),yb_)
191	1	0	1	1	1	1	1	1	OR(ya_,NOT(yb_),NOT(yc_))
192	1	1	0	0	0	0	0	0	AND(yb_, y_)
193	1	1	0	0	0	0	0	1	switch:Q_=0: y=NOR(ya_,yb_)/Q_=1: y=yb_
194	1	1	0	0	0	0	1	0	switch:Q_=0: y=NOR(NOT(ya_),yb_)/Q_=1: y=yb_
195	1	1	0	0	0	0	1	1	EQUAL(yb_, y_)
196	1	1	0	0	0	1	0	0	switch:Q_=0: y=AND(NOT(ya_),yb_)/Q_=1: y=yb_
197	1	1	0	0	0	1	0	1	NOT(RS-flipflop, Toggle-dominant, ya_:S, yb_:R,y_=Q_)
198	1	1	0	0	0	1	1	0	NOT(RS-flipflop, Toggle-dominant, ya_:NOT(S), yb_:R,y_=Q_)
199	1	1	0	0	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=yb_
200	1	1	0	0	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=yb_
201	1	1	0	0	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=yb_
202	1	1	0	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1:y=yb_
203	1	1	0	0	1	0	1	1	NOT(RS-flipflop, R-dominant, ya_:R, yb_:S,y_=Q_)
204	1	1	0	0	1	1	0	0	yb_
205	1	1	0	0	1	1	0	1	NOT(RS-flipflop, R-dominant, ya_:S, yb_:R,y_=Q_)
206	1	1	0	0	1	1	1	0	NOT(RS-flipflop, R-dominant, ya_:NOT(S), yb_:R,y_=Q_)
207	1	1	0	0	1	1	1	1	y_=>yb_ (implikace)
208	1	1	0	1	0	0	0	0	AND(y_,ya_=>yb_ (implikace))
209	1	1	0	1	0	0	0	1	NOT(yb_=NOT(WRITE),ya_=data, y_=NOT(Q_))

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
210	1	1	0	1	0	0	1	0	XOR(NOT(ya_=>yb_),y_)
211	1	1	0	1	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(NOT(ya_),yb_)
212	1	1	0	1	0	1	0	0	NOT(RS-flipflop, D-dominant, ya_:S, yb_:R,y_=NOT(Q_))
213	1	1	0	1	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=OR(NOT(ya_),yb_)
214	1	1	0	1	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
215	1	1	0	1	0	1	1	1	switch:Q_=0: y=NAND(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
216	1	1	0	1	1	0	0	0	ya_=WRITE,yb_=data, y_=Q_
217	1	1	0	1	1	0	0	1	switch:Q_=0: y=EQUAL(ya_,yb_)/Q_=1: y=OR(NOT(ya_),yb_)
218	1	1	0	1	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=OR(NOT(ya_),yb_)
219	1	1	0	1	1	0	1	1	NOT(ya_=NOT(yb_)=yc_)
220	1	1	0	1	1	1	0	0	RS-flipflop, S-dominant, ya_:R, yb_:S,y_=Q_
221	1	1	0	1	1	1	0	1	ya_=>yb_ (implikace)
222	1	1	0	1	1	1	1	0	switch:Q_=0: y=OR(ya_,yb_)/Q_=1: y=NOR(NOT(ya_),yb_)
223	1	1	0	1	1	1	1	1	OR(NOT(ya_),yb_,NOT(y_))
224	1	1	1	0	0	0	0	0	AND(y_,OR(ya_, yb_))
225	1	1	1	0	0	0	0	1	XOR(NOR(ya_,yb_),y_)
226	1	1	1	0	0	0	1	0	yb_=NOT(WRITE),ya_=data, y_=Q_
227	1	1	1	0	0	0	1	1	switch:Q_=0: y=NOT(yb_)/Q_=1: y=OR(ya_,yb_)
228	1	1	1	0	0	1	0	0	NOT(RS-flipflop, D-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
229	1	1	1	0	0	1	0	1	switch:Q_=0: y=NOT(ya_)/Q_=1: y=OR(ya_,yb_)
230	1	1	1	0	0	1	1	0	switch:Q_=0: y=XOR(ya_,yb_)/Q_=1: y=OR(ya_,yb_)
231	1	1	1	0	0	1	1	1	NOT EQUAL(ya_,yb_,NOT(y_))
232	1	1	1	0	1	0	0	0	switch:Q_=0: y=AND(ya_,yb_)/Q_=1: y=OR(ya_,yb_)
233	1	1	1	0	1	0	0	1	NOT(1 from 3 equal to 1)
234	1	1	1	0	1	0	1	0	switch:Q_=0: y=ya_/Q_=1: y=OR(ya_,yb_)
235	1	1	1	0	1	0	1	1	switch:Q_=0: y=NAND(NOT(ya_),yb_)/Q_=1: y=OR(ya_,yb_)
236	1	1	1	0	1	1	0	0	NOT(RS-flipflop, R-dominant, ya_:NOT(S), yb_:R,y_=NOT(Q_))
237	1	1	1	0	1	1	0	1	switch:Q_=0: y=OR(NOT(ya_),yb_)/Q_=1: y=OR(ya_,yb_)
238	1	1	1	0	1	1	1	0	OR(ya_, yb_)
239	1	1	1	0	1	1	1	1	OR(ya_,yb_,NOT(y_))
240	1	1	1	1	0	0	0	0	y_
241	1	1	1	1	0	0	0	1	OR(y_,NOR(ya_, yb_))
242	1	1	1	1	0	0	1	0	OR(y_,NOT(ya_=>yb_))
243	1	1	1	1	0	0	1	1	yb_=>y_ (implikace)
244	1	1	1	1	0	1	0	0	OR(y_,NOT(yb_=>ya_))
245	1	1	1	1	0	1	0	1	ya_=>y_ (implikace)
246	1	1	1	1	0	1	1	0	OR(y_,XOR(ya_, yb_))

y_	y_=1				y_=0				**Odpovídající booleova rovnice pro $F_n = [f_{n,0}, f_{n,1}, f_{n,2}, f_{n,3}, f_{n,4}, f_{n,5}, f_{n,6}, f_{n,7}]$**
yb_	yb_=1		yb_=0		yb_=1		yb_=0		poznámka: SWITCH lze chápat i jako podmínku IF a odpovídá rovnici: Q=OR(AND(NOT(y_),A),AND(y_,B)), A i B se nahradí jinou rovnicí kde A je část rovnice pro Q_=0 (y_=0) a B je část rovnice pro Q_=1 (y_=1)
ya_	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	ya_=1	ya_=0	
F_n	$f_{n,7}$	$f_{n,6}$	$f_{n,5}$	$f_{n,4}$	$f_{n,3}$	$f_{n,2}$	$f_{n,1}$	$f_{n,0}$	
247	1	1	1	1	0	1	1	1	NAND(ya_,yb_,NOT(y_))
248	1	1	1	1	1	0	0	0	OR(y_, AND(ya_, yb_))
249	1	1	1	1	1	0	0	1	OR(y_, EQUAL(ya_, yb_))
250	1	1	1	1	1	0	1	0	OR(ya_, y_)
251	1	1	1	1	1	0	1	1	OR(ya_,NOT(yb_),y_)
252	1	1	1	1	1	1	0	0	OR(yb_, y_)
253	1	1	1	1	1	1	0	1	OR(NOT(ya_),yb_,y_)
254	1	1	1	1	1	1	1	0	OR(ya_,yb_,y_)
255	1	1	1	1	1	1	1	1	CONST=1

13. Závěr

Z původně velkorysého záměru vyvinout zcela nový druh operačního systému, který by se průběžně vyvíjel podle momentálních potřeb uživatele a podle vhodným způsobem formulovaných cílů a individuálních přání kam se má vývoj systému ubírat, se vývoj aplikace v rámci dizertační práce omezil jen na počáteční fázi vývoje. Přestože výzkum umělé inteligence (kam samozřejmě patří i tato práce) zaznamenává v celosvětovém měřítku každoročně velké úspěchy v mnoha svých oblastech bádání, v některých jeho částech jsme stále ještě na počátku a mnoho práce máme před sebou, než se aplikované technologie ve větší míře začnou objevovat mimo výzkumná pracoviště i v běžných domácnostech.

Určitého pokroku bylo v předložené práci dosaženo, několik jednoduchých příkladů automatického vývoje algoritmů bylo nalezeno. Vývoj nového typu reprezentace algoritmů a odpovídajícího prostředí, ve kterém lze tyto algoritmy opakovaně spouštět, bylo k dosažení tohoto cíle také nezbytné a byl splněn. Prostředí celulárního procesoru logických funkcí je v této práci dostatečně popsáno i s pomocí matematického jazyka. Co se vývoje samotné aplikace psané v assemblerovém jazyce procesoru x86 týče, tuto aplikaci je možné dále rozšiřovat a její architektura je zvolena co možná nejotevřeněji, aby ji bylo možno použít později i jako subsystém v mnohem větším, navazujícím projektu, například v projektu umělé evoluce operačního systému nebo pro individuální specializované aplikace (datamining, šifrování dat, komprimace dat, optimalizace procesů, rozpoznávání obrazu, atd.).

14. Literatura

[1] ANDRE David, BENNETT Forrest H. III. and KOZA John. R., 1996, *Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem.* Genetic Programming 1996: Proceedings of the First Annual Conference. Stanford University, CA, USA, 28–31 July 1996. MIT Press.,
URL: http://www.genetic-programming.com/jkpdf/gp1996gkl.pdf

[2] MITCHELL Melanie, HRABER T. Peter, CRUTCHFIELD P. James, 1993, *Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations*. Complex Systems, *7, 89 -130.*, URL: http://web.cecs.pdx.edu/~mm/rev-edge.pdf

[3] BÄCK Thomas, FOGEL B. David., Michalewicz, Zbigniew, 1997, *Handbook of Evolutionary Computation*, Oxford: Oxford University Press, ISBN 0750303921

[4] MITCHELL Melanie, CRUTCHFIELD P. James, DAS Rajarshi, 1997, *G1.15: Computer Science Application: Evolving Cellular Automata to Perform Computations.* In Handbook of Evolutionary Computation, Oxford: Oxford University Press,
URL: http://web.cecs.pdx.edu/~mm/handbook-of-ec.pdf

[5] BÄCK Thomas, Breukelaar Ron, 2005, *Using Genetic Algorithms to Evolve Behavior in Cellular Automata*, UC'05 Proceedings of the 4th international conference on Unconventional Computation, Springer-Verlag Berlin, Heidelberg, ISBN:3-540-29100-8 978-3-540-29100-8, URL: http://www.liacs.nl/~rbreukel/publications/UC.pdf

[6] MARQUES-PITA Manuel, MITCHELL Melanie, ROCHA Luis M., 2008, *The Role of Conceptual Structure in Designing Cellular Automata to Perform Collective Computation*, Proceedings of the Conference on Unconventional Computation, UC 2008, Springer - Lecture Notes in Computer Science,
URL: http://web.cecs.pdx.edu/~mm/UC2008.pdf

[7] KOZA John. R., HALL Jacks Margaret, 1993, *Discovery of Rewrite Rules in Lindenmayer Systems and State Transition Rules in Cellular Automata via Genetic Programming*, Symposium on Pattern Formation (SPF-93) at Claremont, California
URL: http://www.genetic-programming.com/jkpdf/spf1993.pdf

[8] BANZHAF Wolfgang, NORDIN Peter, KELLER E. Robert, FRANCONE D. Frank, 1998, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, ISBN 1-55860-510-X

[9] CONOR Ryan, COLLINS J.J., O'NEILL Michael, 1998, *Grammatical evolution: Evolving programs for an arbitrary language.* Proceedings of the First European Workshop on Genetic Programming, volume 1391 of LNCS, Paris, Springer-Verlag.
ISBN 3-540-64360-5, URL http://www.lania.mx/~ccoello/eurogp98.ps.gz

[10] POLI Riccardo, LANGTON B, William, MCPHEE F. Nicholas, KOZA R. John, 2008, *A Field Guide to Genetic Programming*, Creative Commons Attribution- Noncommercial - No Derivative Works 2.0 UK: England & Wales License, ISBN 978-1-4092-0073-4, URL: http://www.gp-field-guide.org.uk/

[11] MAŘÍK, V.,ŠTĚPÁNKOVÁ, O., LAŽANSKÝ, J., a kolektiv. *Umělá inteligence (3)*, Academia 2001, page 79, ISBN 80-200-0472-6

[12] MAŘÍK, V.,ŠTĚPÁNKOVÁ, O., LAŽANSKÝ, J., a kolektiv. *Umělá inteligence (4)*. Academia, Praha, 2003., ISBN 80-200-0472-6

[13] HYNEK, J. *Genetické algoritmy a genetické programování*, Grada Publishing, a.s., 2008, pages 21 – 25, ISBN 978-80-247-2695-3

[14] ZELINKA I., OPLATKOVÁ Z., ŠEDA M., OŠMERA P., VČELAŘ F. *Evoluční výpočetní techniky, Principy a aplikace*. BEN - technická literatura, Praha, 2009. ISBN 978-80-7300-218-3

[15] SKORKOVSKÝ, P. *Cellular processor of logical functions dedicated for use with genetically programming and evolutionary algorithms*. Proceedings Seventh international conference on soft computing applied in computer and economic environments ICSC 2009. 2009. ISBN: 978-80-7314-163- 9.

[16] SKORKOVSKÝ, P. *Cellular processor of logical functions dedicated for use with genetically programming and evolutionary algorithms*. Proceedings of the 15th conference STUDENT EEICT 2009 Volume 4. 2009. ISBN: 978-80-214-3870- 5.

[17] SKORKOVSKÝ, P. *Solving of Logic Functions with Use of a Cellular Automaton*. XXVII International Colloquium on the Management of Educational Process aimed at current issues in science, education and creative thinking development. 2009. ISBN: 978-80-7231-650- 2.

[18] SKORKOVSKÝ, P. *Minimalist implementation of a genetically programming process written in assembly language*. Aplimat - Journal of Applied Mathematics volume III (2010), number III. 2010. ISBN: 978-80-89313-47- 1.

[19] SKORKOVSKÝ, P. *Development of a program written in assembly language for a genetically programming process - current progress*. Proceedings, Eighth international conference on soft computing applied in computer and economic environments, ICSC 2010. 2010. ISBN: 978-80-7314-201- 8.

[20] SKORKOVSKÝ, P. *Selection Algorithm for Implementation of Genetically Programming with an Assembly Language*. Proceedings of abstracts and electronic version of reviewed contributions, XXVIII International Colloquium on the Management of Educational Process. 2010. ISBN: 978-80-7231-722- 6.

[21] SKORKOVSKÝ, P. *Improvements leading to better performance of algorithms used in the genetically programming process in assembly language*. Aplimat - Journal of Applied Mathematics volume IV (2011). 2011. ISSN: 1337-6365.

[22] SKORKOVSKÝ, P. *First results of cellular logical processor used in genetically programming process.* Proceedings, Ninth international conference on soft computing applied in computer and economic environments, ICSC 2011. 2011. ISBN: 978-80-7314-221-6

[23] SKORKOVSKÝ, P. *Example of algorithm evolution based on cellular processor of logical functions.* Proceedings of abstracts and electronic version of reviewed contributions, XXIX International Colloquium on the Management of Educational Process. 2011. ISBN: 978-80-7231-779-0

[24] SKORKOVSKÝ, P. *Example of induction from a sequence of numbers by a genetic programming process.* Proceedings, Eleventh international conference on soft computing applied in computer and economic environments, ICSC 2013. 2013. ISBN: 978-80-7314-291-9

[25] SKORKOVSKÝ, P. *Genetic Evolution of Algorithm for Identification of ASCII Character Types from Seven Binary Inputs.* XXXI International Colloquium on the Management of Educational Process aimed at current issues in science, education and creative thinking development. 2013. ISBN: 978-80-7231-923-7

Příloha A: CD-ROM

Součástí této dizertační práce jsou soubory uložené jako příloha na datovém nosiči CD-ROM s následujícím obsahem:

- Složka **analýza ASCII** : Soubory použité pro analýzu implementace pro identifikaci ASCII znaků
  - F4596_MF4608 for analysis.csv
  - out-F4596_FM4608_used for analysis.txt
  - ascii_analysis.xls
- Složka **analýza dekoder LED**: Soubory použité pro analýzu implementace pro dekoder LED
  - dec- C64 V2000 F770 FM770 G17409!! - out.txt
  - dec- C64 V2000 F770 FM770 G17409!! - out - vector.txt
  - out_33bunek_US6.txt
- Složka **analýza mult3** : Soubory použité pro analýzu implementace pro indukci čísel z neznámé řady – násobení 3mi
  - mult3_out_y0y1y2y3y4_pouzit pro analyzu.txt
  - mult3_out_y4y5y6y7_pouzit pro analyzu.txt
  - mult3-vector-simulation of outputs_all bits.xls
- Složka **analýza MUX_8bit-out cases_staré-jen pro zajímavost** : Soubory použité pro analýzu implementace pro multiplexer 3bity => 8 bitů
  - V0-0.glg
  - V0-1.glg
  - V0-4.glg
  - V0-6.glg
  - MUX_8bit_result.txt
  - MUX_8bit_result2.txt
  - MUX_8bit-fdf.txt
  - out-v0-0.txt
  - out-v0-1.txt
  - out-v0-4.txt
  - out-v0-6.txt
  - out-v0-7.txt
  - out-v0-8.txt
  - V0-0.txt
  - V0-1.txt
  - V0-4.txt

  - V0-6.txt
- Složka **aplikace ke spuštění**
  - GenAlg.exe
- Složka **fdf specifikace**
  - ascii.fdf
  - dekoder LED segmenty.fdf
  - LEDnumbers.fdf
  - mult3.fdf
  - mult3_bits4567.fdf
  - MUX_8bit.fdf
  - Složka **fdf zdroje**
    - csv2bin.c
    - csv2bin.exe
    - ascii.txt
    - dekoder LED segmenty.txt
    - example.txt
    - LEDnumbers.txt
    - mult3.txt
    - mult3_bits4567.txt
    - MUX_8bit.txt
- Složka **GenAlg zdrojové soubory**
  - GenAlg.asm
  - Build.bat
  - GenAlg.ico
  - DialogProcedures.inc
  - GenAlg.inc
  - GenAlgBinData.inc
  - GeneralFunctions.inc
  - GlobalVariables.inc
  - ident.inc
  - ident_rc.inc
  - InitGenAlg.inc
  - IOmallocData.inc
  - MainAlg.inc
  - MainAlg_thrdX.inc
  - MenuItems.inc
  - running.inc
  - GenAlgRS.rc
- Složka **nalezené implementace v gen souborech**
  - ascii_F4596_FM4608_U16_S5_p70x4a4_mask63_used for analysis.gen

  - dec-C64_V2000_F770_FM770_G2638_60pc_10pc_10pc_asi minut.gen 45
  - dec-C64_V2000_F770_FM770_G3605_ukonceno_s_80pc_6pc_6pc.gen
  - dec-C64_V2000_F770_FM770_G4705_newrepr.gen
  - dec-C64_V2000_F770_FM770_G6057_6hodin.gen
  - dec-C64_V2000_F770_FM770_G6506_6hodin.gen
  - dec-C64_V2000_F770_FM770_G10472.gen
  - dec-C64_V2000_F770_FM770_G17409!!_used_for_analysis.gen
  - dec-C64_V2000_F770_FM770_G22338!!.gen
  - mult3-out_y4y5y6y7_funkcni-y5y6y7.gen
  - MUX_8bit - F704-MAX_G5223.gen
  - MUX_8bit - V2000 C64 F704-MAX_G6501.gen
- Složka **vývojové prostředí-instalace MASM 32 SDK V10**
  - install.exe
- GenAlg.exe

Příloha B: CURRICULUM VITAE

1. Surname, name: **SKORKOVSKÝ Petr**
2. Date of birth: 9th October 1977
3. Nationality: Czech
4. Education:

School / Institution	**from (month/year)** **to (month/year)**	**Qualification acquired**
VUT, Faculty of electrical Engineering and Communication in Brno	Sept. 2008 – 2013	Doctoral graduation (Ph.D.), doctoral thesis on Genetic programming
VUT, Faculty of electrical Engineering and Communication in Brno	Sept. 1997 – June 2002	Engineer graduation in Cybernetics with diploma thesis on Expert systems
Höhere technische Bundeslehranstalt Hollabrunn, Austria – Technical college	Sept. 1992 – June 1997	General certificate of education in Control engineering

5. Company: AREVA NP
6. Present position in the company: SW V&V Engineer.
7. Length of service with the company: From 01/09/2002

8. Professional experience:

from (month/year) to (month/year)	Country/ Location	Company	Position	Description
09/2002 to present	ČR / Brno, Dukovany, France / Paris,	AREVA NP	Software V&V Engineer	DUKOVANY I&C modernisation Project. SW Unit tests, SW System tests, periodic tests, interconnected tests, training of system and V&V engineers – PAMS (Post Accident Monitoring System).
02/2008 to present	ČR / Dukovany, France / Paris	AREVA NP	Software V&V Engineer	V&V tasks on QDS (Qualified Display System)

9. Summary and scope of obligations to other projects, which shall be executed during the performance of the contract:

   - Dedicated to DUKOVANY Project,
   - Dedicated to V&V – QDS (Qualified Display System).

10. Miscellaneous:

    - Software tools :

      Vallog, Banc De Test, Logiscope, TRACE32, Lint, Borland C++ Builder, Wind River Diab C/C++ compiler, Turbo C, GNU software tools, MASM32, gcc.

    - Programming languages:

      C/C++, Pascal, BASIC, PROLOG, scripting, WinAPI32, assembler languages for Z80, 8051C, x86.

    - Operation Systems:

      MS-DOS, Windows 3.11, 95, 98, XP, Windows 7/8, Linux, Palm OS

- Other skills:

Graphic editors, serial communication protocols, LAN networking, microprocessor technology, programming of logical devices, PCB CADs like Eagle, ORCAD, PCAD, electronics and microelectronics.

Foreign languages: English – fluently, German – fluently, French – beginner