VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

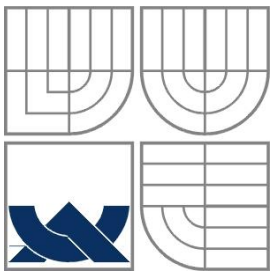
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ FORMÁLNÍ VERIFIKACE KOREKTNOSTI PROGRAMŮ V NÁSTROJÍCH SDV, COPPER A PODOBNÝCH

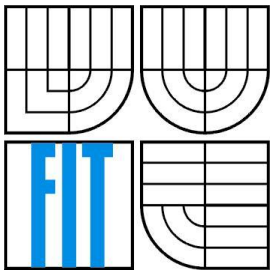
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
PETER KOVALIČ

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ FORMÁLNÍ VERIFIKACE KOREKTNOSTI PROGRAMŮ V NÁSTROJÍCH SDV, COPPER A PODOBNÝCH

AUTOMATED FORMAL VERIFICATION OF PROGRAM CORRECTNESS USING SDV, COPPER, OR SIMILAR TOOLS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER KOVALIČ

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. TOMÁŠ VOJNAR, Ph. D.

BRNO 2010

Abstrakt

Tato práce se věnuje verifikaci ovládačů. Používají se při tom model checkery, hlavní z nich je Static Driver Verifier. Pomocí něj se kontroluje zvolený ovládač Ext2Fsd. Patří do skupiny ovládačů souborových systémů. Kontrola probíhá podle zadaných pravidel, které nesmí ovládač porušovat. Cílem práce bylo vybraný ovládač verifikovat pomocí zvoleného nástroje. Ve výsledcích bylo dosaženo stavu, kdy se ve verifikovaném ovládači objevili všechny tři dostupné možnosti výsledku – ovládač splňoval některá pravidla, některá odhalily jeho chyby a jiné nebyly aplikovatelné. Na konci této práce se nachází ještě kapitola, věnována dalšímu model checkeru, s názvem Copper, která poskytuje základní poznatky o tomto nástroji.

Abstract

This thesis is concerning about verification of drivers. Principally is focused on model checking tools, from which the Static Driver Verifier is the most important. A driver Ext2Fsd is checked by this program. This driver belongs to group of file system drivers. Control is driven by entered rules, which the driver must not violate. The aim of this thesis was to verify chosen driver by selected tool. The results have covered all three types of verification´s end. There were rules that driver passed, that driver violated and also that driver didn´t accept. The final chapter of work is about another model checking tool – Copper. It offers the basic knowledge about this program.

Klíčová slova

verifikace, ovladač, statická analýza, model checking, static driver verifier, ovládač souborového systému, Copper, Ext2Fsd, BLAST, PREfast, Windows driver kit

Keywords

verification, driver, static analysis, model checking, static driver vernier, file systém driver, Copper, Ext2Fsd, BLAST, PREfast, Windows driver kit

Citace

Kovalič Peter: Automatická formální verifikace korektnosti programů v nástrojích SDV, Copper a podobných, bakalářská práce, Brno, FIT VUT v Brně, 2010

Automatická formální verifikace korektnosti programů v nástrojích SDV, Copper a podobných

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. Ing. Tomáše Vojnara. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jméno Příjmení (Peter Kovalič)
Datum (18.5.2010)

Poděkování

Chtěl bych vyjádřit poděkování mému vedoucímu doc. Vojnarovi za užitečné rady a usměrnění při práci tak i při psaní samotné technické správy.

Obsah

1 Úvod

Táto práca sa zaoberá nástrojmi, ktorých hlavnou úlohou je vykonávať automatickú verifikáciu. Týchto programov nie je veľa a väčšinou používajú podobné metódy overovania. Úlohou verifikácie je overenie, či daný program splňuje požiadavky. Využíva tri základné prístupy: kontrolu modelu (angl. model checking), dokazovanie viet (angl. theorem proving) a statickú analýzu (angl. static analysis). V tejto práci sa budú overovať požiadavky operačného systému na ovládač pomocou nástroja založeného na kontrole modelu.

Verifikácia má v dnešnej dobe veľký význam. Mnoho programov, ktoré pracujú úplne automaticky je nutné najskôr skontrolovať, či sú schopné pracovať samostatne tak, aby nenastali nepredpokladané situácie. V takýchto situáciách program nevie ako reagovať a dochádza k problémom. Práve takýmto udalostiam sa dá vyhnúť pomocou verifikácie, ktorá nastáva po skončení vývojovej fázy. Tieto problémy je možné vyriešiť i testovaním, avšak verifikácia môže k tomu dokázať správnosť systému a tak odhaliť všetky možné chyby, ktoré pri fáze testovania nemusia byť nájdené.

Cieľom práce je vybraný ovládač operačného systému verifikovať pomocou zvoleného nástroja na jeho požadované vlastnosti. Po verifikácii vyhodnotiť, ktoré vlastnosti splňuje, nesplňuje alebo ich nemá. Na základe získaných výsledkov zhodnotiť kvalitu ovládača, prípadne oznámiť nedostatky vývojárom.

1.1 Štruktúra práce

Práca obsahuje celkovo deväť kapitol, ktoré najskôr poskytujú všeobecné informácie, ktoré postupne dopĺňajú o podrobnejšie detaily. Obsah práce je štruktúrovaný tak, aby postupným prečítaním všetkých kapitol bolo možné venovať sa verifikácii ovládača bez ďalšieho vysvetľovania princípu fungovania nástroja, verifikácie alebo ovládača samotného. Prvú kapitolu tvorí tento úvod, ktorý obsahuje stručný popis práce, jej cieľ a základné členenie.

Druhá kapitola obsahuje všeobecné informácie o verifikácii. Najskôr definuje pojmy, ďalej vysvetľuje použitie verifikácie a jej prístupy. Potom nasleduje časť, ktorá predstavuje programy, ktoré využívajú kontrolu modelu (angl. model checking). Záver tejto kapitoly vysvetľuje dôvody zvolenia SDV (Static Driver Verifier) ako hlavného nástroja, ktorým sa bude daný ovládač verifikovať.

V kapitole tri je popis zvoleného ovládača, ktorý sa bude verifikovať. Na začiatku je charakterizovaná skupina ovládačov, do ktorej patrí, a sú zhrnuté ich základné vlastnosti, črty a rozdiely oproti iným ovládačom a funkcie, ktoré sú pri týchto ovládačoch používané. Nasledujúca podkapitola sa zaoberá konkrétne zvoleným ovládačom pre verifikáciu Ext2Fsd. Popíše jeho vlastnosti a zloženie zdrojových súborov.

V ďalšej kapitole sa text zameria na prostredie WDK (Windows Driver Kit), z ktorého nástroj na verifikáciu pochádza. Budú spomenuté možnosti práce s týmto prostredím. Zároveň sa popíšu i iné časti tohto prostredia, ktoré úzko súvisia s verifikačným procesom a verifikačným programom samotným.

Pred samotnou verifikáciou je nutné kód verifikovaného ovládača pripraviť. Túto úlohu zohráva ďalší program PREfast, ktorý sa spúšťa pred samotným overením ovládača. Tomuto programu sa venuje kapitola päť. Podľa výpisu z tohto programu je nutné upraviť zdrojové súbory ovládača, tak aby splňovali podmienky pre vstup do verifikačného programu. Tento popis je ďalšou

časťou práce. V tejto kapitole začína príklad, ktorý bol prevzatý z WDK a na ňom je vysvetlená funkcia PREfastu. Tento príklad sa rozoberá i v nasledujúcej kapitole.

Kapitola šesť sa zaoberá popisom zvoleného verifikačného programu SDV. Jeho vlastnosťami, obmedzeniami na ovládače resp. kritériami, ktoré musia ovládače splňovať pre úspešný prechod programom. Bude vysvetlené jeho ovládanie pomocou vstavaných príkazov, vytváranie pravidiel, ktorými sa program pri verifikácii riadi. Nasledujúcim bodom je popis špecializovaných definícií, ktoré sa využívajú k príprave ovládača – typy rolí (angl. role types). V skratke budú zhrnuté ich vlastnosti a niektoré z nich budú bližšie vysvetlené, pretože boli použité pri úprave zdrojových kódov ovládača.

V nasledujúcej kapitole sa práca zamerá na verifikáciu zvoleného ovládača. Najskôr je nutné zdrojové súbory upraviť tak, aby vyhovovali vstupu do SDV. Ďalej sa pripraví súbor pravidiel, podľa ktorých sa bude verifikovať. Nasleduje popis priebehu verifikácie. Na konci tejto časti budú rozobrané výsledky, ktoré sa dosiahli aplikovaním verifikačných pravidiel na ovládač. Niektoré časti výsledku budú podrobnejšie vysvetlené, pretože popisujú zaujímavé zistenia.

Kapitola osem obsahuje pred zhodnotením práce popis iného programu pre verifikáciu – Copper, ktorý bol uvažovaný ako možný kandidát pre hlavný proces verifikovania, ale vzhľadom k podmienkam sa nepoužil, ale je zaujímavým nástrojom, ktorý sa po dostatočne dlhom vývoji môže stať úspešným.

V závere práce budú spomenuté dosiahnuté výsledky verifikačného procesu. Porovná sa jeho úspešnosť s požadovanými kritériami, ktoré boli definované na začiatku práce. Ďalej sa v závere pripomenú nové možnosti pokračovania v tomto testovaní, i možnosť zaslania výsledkov autorom verifikovaného ovládača, ktorý takto môžu svoj výrobok zdokonaliť. Týmto je možné získať účasť na vývoji tohto ovládača a spolupracovať pri jeho vylepšovaní.

2 Verifikácia

2.1 Definícia pojmu

Všeobecne sa dá verifikácia definovať ako proces, pri ktorom sa overujú vlastnosti určitého objektu s cieľom otestovať tieto vlastnosti tak, aby sa dokázalo, že sú platné, čiže nemôžu predstavovať možný problém. V presnej definícii sa verifikácia spomína s príbuznosťou k logike, kde predstavuje postup, ktorého výsledkom je zistenie, že daný výrok je pravdivý.

V počítačovej terminológii sa skôr používa výraz formálna verifikácia, pri ktorej sa použitím logiky stanovia vlastnosti softwarového alebo hardwarového návrhu. Ďalej musia byť dostupné požiadavky, formálne modelovaná implementácia a presné pravidlá záveru, ktoré stanovia kedy implementácia vyhovuje špecifikácii, podľa [1].

2.2 Použitie verifikácie

Verifikácia sa používa k rôznym úlohám na dokazovanie ich vlastností. Podľa [2] je možné verifikáciu rozdeliť podľa použitia na štyri skupiny. Prvou skupinou je využitie verifikácie v aplikáciách – overovanie CAPTCHA obrázkov, kde verifikuje, že daná aplikácia je používaná človekom. Ďalej sa používa v pri overovaní celistvosti súboru alebo pri overení reči.

Druhá skupina používa verifikáciu pri vývoji softwaru a obsahuje formálnu, inteligentnú a verifikáciu za behu programu. Inteligentná verifikácia sa využíva pri aplikovaní testov na zariadenia. Formálna verifikácia bola vysvetlená skôr.

Ďalšia skupina využíva verifikáciu pri konštruovaní obvodov. Používa funkcionálnu, analógovú a fyzikálnu verifikáciu.

Posledná skupina sa týka systémového inžinierstva, kde sa verifikujú vlastnosti systému, či splňujú špecifikované požiadavky.

2.3 Typy verifikačného procesu

Verifikácia zahŕňa simuláciu, testovanie a formálnu verifikáciu. Ďalej formálna verifikácia obsahuje prístupy ako model checking, statická analýza a dokazovanie teorémov. Postupne budú vysvetlené všetky prístupy, ale model checking bude objasnený podrobnejšie, pretože hlavne tomuto prístupu sa bakalárska práca venuje. Nasledujúce časti sú voľne prevzaté z [3].

Dokazovanie teorémov je založené na podobnom prístupe ako sa využívajú pri matematických dôkazoch. V tomto prípade sa však vychádza z dokázaného systému, z ktorého sa vyvodzujú teorémy o skúmanom systéme. Tieto teorémy vychádzajú zo známych údajov o systéme alebo zo všeobecných teorémov. Využívajú sa počítačové programy, ktoré si dokážu pamätať už dokázané teorémy. Väčšinou ide o poloautomatický prístup. Výhodou tohto prístupu je všeobecnosť, naopak nevýhodou je jeho zložité použitie a problémy pri generovaní protipríkladov.

Princíp statickej analýzy je založený na postupnom prechádzaní zdrojového kódu systému, v ktorom zbiera informácie. Preto sa väčšinou využíva ako časť kompilácie alebo optimalizácie. Môže byť vykonaná v rôznych typoch zložitosti od jednoduchej syntaktickej kontroly až po zložité kontroly grafu toku systému. Výhodou statickej analýzy je rýchlosť kontroly rozsiahlych zdrojových

kódov. Avšak len čiastočná kontrola hodnôt systémových premenných je nevýhodou a môže viesť k falošným upozorneniam.

Model checking je proces automatickej kontroly, či model systému splňuje požiadavky špecifikácie. Systematicky prechádza stavový priestor modelu. Hlavnou nevýhodu model checkingu je problém stavovej explózie. Ďalší popis tohto prístupu je uvedený podľa [4].Skladá sa z troch častí modelovania, špecifikácie a verifikácie.

Pri modelovaní sa konvertuje systém do formalizmu, ktorý akceptuje nástroj pre model checking. Väčšinou ide o prekladovú časť avšak ak by bola táto časť náročná, tak sa využíva abstrakcie od niektorých vlastností, ktoré sú nepotrebné.

Špecifikácia definuje vlastnosti, ktoré musí testovaný softvér splňovať. Využívajú sa logické formalizmy, u softvérových systémov sa väčšinou využíva temporálna logika. Hlavným problémom v tejto časti model checkingu je úplnosť špecifikácie, pretože nie je možné určiť, či špecifikácia naozaj pokrýva všetky potrebné vlastnosti.

Samotný proces verifikovania ako posledná časť model checkingu je väčšinou automatický. Niekedy je však nutné do nej zasiahnuť. Typickým zásahom do tejto časti je analýza výsledkov po skončení verifikovania. Ďalším zásah je možný ak nastane chyba a proces verifikácie sa nemohol dokončiť. Vtedy je nutné pomocou chybového výpisu zistiť, prečo chyba nastala. Tá mohla nastať už pri návrhu abstraktného systému, napríklad nevhodne zvolenou abstrakciou niektorých vlastností. Vtedy je nutné celý systém prepracovať. Ďalšou chybou, ktorá môže spôsobiť neúspešný koniec verifikácie, je veľkosť systému, ktorý sa verifikuje. Nástroj, ktorý uskutočňuje verifikáciu nedokáže uložiť všetky stavy do pamäti. Riešením tohto problému môže byť využitie ďalšej abstrakcie pri špecifikácii vlastností.

Po tomto uvedení typov formálnej verifikácie bude nasledovať výber nástroja, s pomocou ktorého sa bude zvolený ovládač verifikovať. Všetky uvedené nástroje využívajú model checking. Niektoré z nich používajú i iný prístup, ale hlavnou časťou nástroja je vždy verifikácia pomocou model checkingu.

2.4 Uvažované verifikačné nástroje

Na začiatku je nutné spomenúť, že všetky nástroje, ktoré boli uvažované, sú založené na model checkingu. Nástroje nie sú všetky možné, ktoré existujú, ale patria k tým overenejším a ich fungovanie je doložené ich častým používaním. Pri predstavení každého nástroja budú uvedené jeho vlastnosti, podpora zo strany autorov, rozsiahlosť dokumentácie a vhodnosť pre účel tejto práce. Po uvedení všetkých týchto nástrojov bude nasledovať ich krátke zhodnotenie a výber jedného z nich, ktorý bude slúžiť ako hlavný nástroj využívaný v tejto práci.

2.4.1 Berkeley Lazy Abstraction Software Verification Tool

Nástroj patrí medzi staršie a dlhú dobu využívané nástroje. Skrátene sa volá BLAST(Berkeley Lazy Abstraction Software Verification Tool). Vyvinutý bol na univerzite v Berkeley. Autormi sú Ranjit Jhala, Rupak Majumdar, a Gregoire Sutre. Program je pre zdrojové kódy v jazyku C. Využíva model checking a je založený na predikátovej abstrakcii, ktorá je automaticky zjemňovaná na základe nájdených neplatných protipríkladoch k overovaným vlastnostiam (ide o tzv. CEGAR – Counter-Example Guided Abstraction Refinement). BLAST sa zameriava na verifikáciu vlastností typu bezpečnosť špecifikovaných pomocou assertions. Tým kontroluje hlavne bezpečnosť pamäte voči únikom pri používaní ukazateľov. Časť bola prebraná z [5].

Posledná verzia programu je z 11.7.2008 a odvtedy nebola vydaná ďalšia verzia. Programová dokumentácia je dostatočne obsiahla.

2.4.2 Copper

Copper je ďalší program využívajúci model checking. Vyvinul ho Software Engineering Institute z CarnegieMellon University. Využíva sa pre verifikáciu paralelných programov v jazyku C, ktoré komunikujú pomocou správ. Dokáže analyzovať, či program splňuje bezpečnostné a spoľahlivostné požiadavky. Táto časť bola prebraná z [6].

Program je v súčasnosti vo verzii 2.0 a je súčasťou väčšieho balíku s názvom Comfort, ktorý je dedukčným prostredím(angl. reasoning framework), bližšie vysvetlené v Kapitole 8, a jeho jadro tvorí Copper. Copper je možné inštalovať i samostatne bez systému Comfort. Dokumentácia k tomuto programu je kratšia, uvažuje sa, že užívateľ, ktorý by Copper použil, už má dostatočné základy s verifikačným mechanizmom.

2.4.3 Static Driver Verifier

SDV(Static Driver Verifier) je program vyvinutý spoločnosťou Microsoft. Používa sa na analýzu zdrojových súborov ovládačov pre systém Microsoft Windows. Je založený na sade základných pravidiel a modeli operačného systému. SDV určí, či ovládač správne komunikuje s jadrom operačného systému Windows tak, že overuje zadané vlastnosti na základe systematického prechodu stavovým priestorom booleovskej abstrakcie verifikovaného programu, ktorá je automaticky vytvorená pomocou predikátovej abstrakcie postupne zjemňovanej na základe nájdených neplatných protipríkladov, podobne ako je to u nástroja BLAST. K tomu zahŕňa i niektoré pomocné techniky z oblasti statickej analýzy (uvedené v kapitole 5) , ktoré sa aplikujú pred model checkingom.

Najnovšia verzia SDV je závislá na balíku, s ktorým je dodávaná. Dodáva sa štandardne v balíku Microsoft Windows Driver Kit (WDK), kde slúži ako jedna časť pri vytváraní ovládača. Aktuálna verzia balíku v dobe písania bakalárskej práce je 7.1. Program má rozsiahlu dokumentačnú časť a vzhľadom k tomu, že je SDV obsiahnutý vo WDK je často používaný.

2.4.4 Výber vhodného programu

Pri výbere vhodného model checkeru (nástroj, ktorý využíva model checking), pre experimenty v rámci tejto práce, boli zohľadnené viaceré faktory. Program musí obsahovať dostatočne kvalitnú dokumentáciu. Tým zrejme odpadá možnosť využitia programu Copper. Ďalším faktorom bola aktuálnosť programu. V týchto parametroch viedli Copper a SDV. Blast sa dva roky nevyvíjal a pri použití s novými operačnými systémami by nemusel správne pracovať. Podľa týchto faktorov bol vybraný ako hlavný program SDV od Microsoftu, pretože poskytuje dostatočne obsiahlu dokumentáciu. Stále sa vyvíja spolu s balíkom WDK vždy pre najnovší operačný systém Windows, ktorého predposlednú verziu používam (Windows Vista). Keďže je súčasťou balíka WDK, využíva ho mnoho programátorov, ktorí vytvárajú ovládače pre tento systém, preto je na internete dostatok odkazov na témy z tejto oblasti. Pri nejakom probléme, ktorý by sa mohol vyskytnúť, je možné tieto fóra využiť.

3 File system drivers

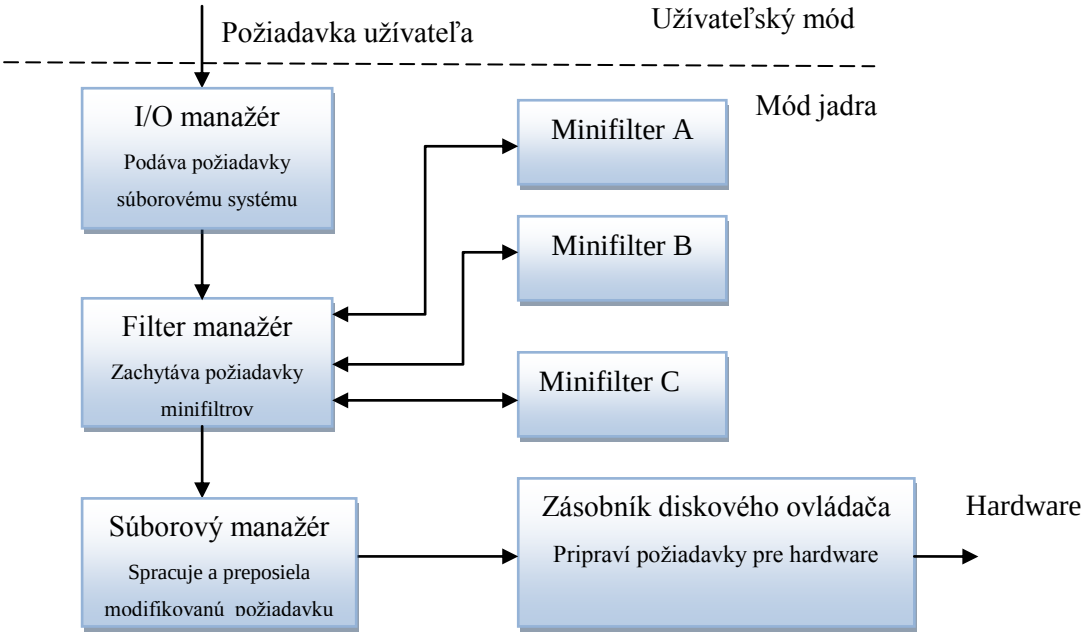
V tejto časti budú priblížené informácie o ovládačoch pre súborové systémy, pretože ovládač, ktorý bude verifikovaný pomocou SDV, patrí do tejto skupiny. Po tomto všeobecnejšom úvode bude nasledovať časť, ktorá podrobnejšie rozoberie vlastnosti vybraného ovládača a jeho schopností.

3.1 Charakteristika ovládača

Ovládač súborového systému je odlišný od ovládačov, ktoré obsluhujú zariadenia. Hlavným rozdielom je ovládanie vstupu a výstupu nezávisle od fyzického zariadenia. Normálne ovládače len poskytujú vstupne/výstupné služby zariadeniu. V ovládači súborového systému sa nachádzajú i parametre, ktoré určujú, o aký typ súborového systému ide (ext2, FAT, NTSF). Tieto ovládače obsahujú dve podskupiny: filtrované ovládače súborového systému (angl. File system filter drivers) a ovládače súborového systému s minifilterom (angl. File system minifilter drivers). Rozdelenie ovládačov je prevzaté z [7] a [8].

Prvé menované sa vyznačujú tým, že schopnosti obyčajných ovládačov systémových súborov rozširujú o ďalšie služby alebo napríklad o ošetrenie proti vírusom (angl. virus screening). V tomto prípade slovo filter v názve skupiny neoznačuje len schopnosť filtrovať požiadavky, ale znamená i zapísanie log súboru, modifikovanie požiadavku alebo jeho obmedzenie. Ich typické využitie je v antivírusových, kryptovacích programoch a v správe hierarchického úložiska.

Druhá skupina sa odlišuje od predchádzajúcej tým, že tieto ovládače môžu byť napísané externými vývojármi, a priamo nesúvisia s ovládačom pre súborový systém. Komunikujú s ním pomocou manažéra filtrov (filter manager), ktorý je priamo vyvinutý pre spojenie s ovládačom súborového systému (Obrázok 1 prevzatý z [8]). Filter manažér spracúva požiadavky filtrov definovaných v jednotlivých minifiltroch. Tieto požiadavky ďalej spracuje súborový manažér, ktorý priamo komunikuje so zásobníkom diskového ovládača. Preto je vhodné skôr napísať minifilter, ktorý nemusí mať všetky oprávnenia byť priamo ovládačom súborového systému.



Obrázok 1: Princíp fungovania ovládača súborového systému typu minifilter

3.2 Vlastnosti ovládača

Vlastnosti ovládača súborového systému sa najlepšie vysvetlia porovnaním s vlastnosťami normálneho ovládača. Budú ďalej v tejto kapitole rozobrané podobné i odlišné prvky, ktoré sa vyskytujú v ovládači.

3.2.1 Podobnosti medzi ovládačom súborového systému a ovládačom zariadenia

Tieto dva typy ovládačov majú isté podobnosti v štruktúre i funkcionalite. V štruktúre obidva typy obsahujú obslužné rutiny (angl. routines): `DriverEntry`, dispatch a I/O completion.

Rutina je funkcia, ktorú musí program volať, aby systém pochopil, čo má vykonať. V tomto prípade rutina `DriverEntry` sa používa na inicializáciu ovládača, rutiny typu dispatch spracovávajú pakety, ktoré sa využívajú na komunikáciu. Rutiny typu I/O completion sa používajú v ovládačoch, ktoré riadia ovládače na nižšej úrovni. Napríklad (na Obrázok 1) rutinu typu I/O completion bude využívať súborový manažér aby spracoval požiadavky pre Zásobník diskového ovládača.

Podobnosť vo funkcionalite je v tom, že obidva typy patria do vstupne/výstupnej časti systému, čiže dostávajú požiadavky s paketmi (angl. I/O requests packet – IRP) a vykonávajú nad nimi úlohy. Ďalším spoločným prvkom je tvorba vlastných IRP paketov a ich zasielanie nižšie položeným ovládačom. Rovnako môžu odchytávať udalosti pomocou tzv. callback funkcií[1].

Ďalej vedia rovnako ako ovládače zariadenia odchytávať IOCTL (Introduction to I/O Control Codes)[2], k tomu môžu prijímať a definovať i FSCTL (File System Control Codes)[3]. Podobnosť medzi týmito dvomi typmi je i v konfigurácii, kedy sa nahrávajú do systému. Môžu sa nahrávať pri spustení systému, alebo až po ukončení procedúry spúšťania systému. Informácie boli voľne prevzaté z [9].

3.2.2 Rozdiely medzi ovládačom súborového systému a ovládačom zariadenia

Prvý rozdiel je v správe napájania zariadenia. Keďže ovládače súborového systému priamo nekomunikujú so zariadením, tieto príkazy na správu napájania spravuje priamo zásobník diskového ovládača (angl. storage device stack – na Obrázok 1). Vo vzácnych prípadoch môže nastať situácia, že ovládač zasahuje do správy napájania. Preto vo funkcii `DriverEntry` nesmie byť volaná rutina pre správu napájania. Rovnako sú obmedzené volania rutín pre pridanie zariadenia (`AddDevice`) a začatie vstupne/výstupného prenosu (StartIO). Obidva typy ovládačov môžu vytvárať objekty zariadení (angl. Device Object)[4], avšak sa líši počet a druh týchto objektov. Ďalším rozdielom je priamy prístup k pamäti, ktorý nemôže ovládač súborového systému využiť. Voľne prebrané z [10].

[1] Callback funkcia sa využíva pri volaní funkcie, ktorá sa nachádza v externej knižnici. Je deklarovaná v programe a zariaďuje úspešný návrat z externe volanej funkcie.
[2] Tieto kontrolné kódy sa využívajú na komunikáciu medzi užívateľskou aplikáciou a ovládačom alebo ovládačmi, ktoré sa nachádzajú na zásobníku. Odosielajú sa v IRP paketoch.
[3] Kontrolné kódy, ktoré využívajú práve ovládače systémových súborov na komunikáciu.
[4] Objekt zariadenia slúži na jeho identifikáciu a zároveň definuje jeho funkcie, vlastnosti a služby.

3.3 Ext2Fsd

Tento ovládač súborového systému bol vybraný na základe otvorených zdrojových kódov, ktoré sú nutné pre verifikáciu. Ďalším dôvodom pre výber tohto ovládača bol jeho pokračujúci vývoj, ktorý síce zastal v roku 2008, ale bol jedným z najnovších, ktoré boli dostupné. V prospech ovládača hralo významnú rolu i to, že bol vyvíjaný v prostredí Windows Driver Kit, čiže komplikácie spojené s nefunkčnosťou ovládača pre verifikačný nástroj by nemali nastať. Je tu istá pravdepodobnosť, že niektoré súbory bude nutné prepísať alebo skonvertovať do podoby, ktorá bude vyhovovať najnovším nástrojom, ktoré sa teraz používajú. Napríklad súbor obsahujúci projekt pre Microsoft Visual Studio bol vytvorený pre staršiu verziu tohto vývojového prostredia.

Tento ovládač patrí svojím štýlom do skupiny Windows driver model (WDM), pretože používa funkcie, ktoré sa práve vyskytujú v tomto type ovládača, ale z hľadiska používania ide o ovládač súborového systému. Na základe tohto zistenia je možné určiť vlastnosti, ktoré by mal daný ovládač spĺňať z hľadiska verifikácie. Tieto vlastnosti budú špecifikované až pri zostavovaní pravidiel pre samotný proces model checkingu. Rôzne typy ovládačov a ich odlišné spôsoby úpravy pre SDV budú popísané v kapitole venovanej práve verifikačnému nástroju.

3.3.1 Vlastnosti ovládača

Ext2Fsd je ovládač, ktorý sa používa v systémoch Microsoft Windows. Jeho úlohou je dovoliť pripojenie diskov k počítaču, ak majú súborový systém ext2, ext3. Tieto súborové systémy sa používajú v operačných systémoch Linux a vo Windows nemajú štandardne podporu. Tento ovládač tento nedostatok opravuje.

Základnými vlastnosťami tohto ovládača sú podpora čítania a zápisu pre ext2, ext3 súborový systém, rôzna podpora kódovania, automatické priradenie bodu pripojenia, veľká veľkosť i-uzlov, podpora súborov väčších ako 4 GB, podpora operačných systémov Windows 2000, XP, Vista, 2008 Server.

Ovládač má i niektoré nedostatky, resp. vlastnosti, ktoré nepodporuje. Sú to podpora ext4 súborového systému, LVM a zapojenie diskov do poľa (RAID), nepodporuje Windows NT4 a podpora systémov Windows 7 a Windows 2008 r2 je nestabilná.

3.3.2 Zloženie ovládača

Ovládač má zdrojové kódy napísané v jazyku C. Konkrétne obsahuje 71 zdrojových súborov a 48 hlavičkových súborov. Súčasťou priečinka s ovládačom sú aj súbory typu *Makefile*, pomocou ktorých je možné aplikáciu preložiť a spustiť. Je však nutné dodržať hierarchiu kompilovania, pretože je nutné najskôr skompilovať súbory v priložených priečinkoch a až potom kompilovať aplikáciu celkovo. Postup kompilácie je čiastočne popísaný v súbore *FAQ.txt* a ostatné detaily je možné vyčítať zo súborov *Makefile*. V zložke ovládača je i pred-pripravený projekt pre Microsoft Visual Studio, ktorý po spustení načíta projekt s názvom ext3fsd. Samotný projekt je členený na dva väčšie celky Ext3Fsd a jbd, z ktorých je hlavný projekt Ext3Fsd. K projektu sú umiestnené hlavičkové súbory v adresároch asm a linux. Hlavým zdrojovým súborom je *init.c*, v ktorom je inicializácia ovládača cez funkciu `DriverEntry`. Všetky zdrojové súbory sú prístupné cez SVN adresár, cez ktorý je možné stiahnuť i predošlé verzie ovládača. Štruktúra zložiek a súborov v adresári ovládača je v Príloha č.1.

4 Windows driver kit

V tejto kapitole bude v skratke vysvetlený obsah tohto balíka, ktorého hlavnou úlohou je vytváranie ovládačov pre operačný systém Windows. Proces vytvorenia ovládača pozostáva z viacerých fáz, ktoré na seba nadväzujú. Jednou z týchto častí je aj verifikácia vytváraného ovládača, ktorú vykonáva práve Static Driver Verifier.

Windows driver kit (WDK) je rozdelený na dve časti. Jednou z nich je programová časť, ktorá obsahuje príkazový riadok, z ktorého je možné spúšťať jednotlivé moduly. Je umiestnený v zložke Build Environment, ktorá je rozdelená podľa typov operačných systémov Windows a ďalej podľa typu architektúry. Je hlavným prostredím, cez ktoré sa komunikuje s verifikačným nástrojom. Ďalšou časťou programovej časti je simulátor zariadení, ktorý nie je inštalovaný automaticky.

Druhá časť tohto balíka je jeho dokumentácia, ktorá je rozsiahla a poskytuje dostatočné zázemie pre naprogramovanie ovládača. Podľa štruktúry tejto dokumentácie je možné prakticky prejsť celým cyklom vývoja určitého typu ovládača. Týmto smerom sa bude uberať i táto kapitola a v skratke osvetlí jednotlivé princípy, ktoré vedú k získaniu vhodného ovládača pre systém Windows.

4.1 Štruktúra Windows driver kitu

Pri vytváraní ovládača je dôležité si najskôr ujasniť, pre aké zariadenie bude a ako ho bude možné naprogramovať. Týmito otázkami sa zaoberá jedna časť WDK. Pomáha s výberom správneho modelu ovládača, vhodného programovacieho jazyka. Ďalej ukazuje rôzne techniky návrhu a implementačné stratégie. Neskôr upozorňuje na vytváranie spoľahlivých a bezpečných ovládačov a poskytuje jednoduché ukážky z ich programovania. V nasledujúcom bloku vysvetľuje princíp preprogramovania ovládačov, ktoré boli používané v minulých systémoch a teraz by už neboli podporované. Zároveň opisuje i spôsoby ako vytvárať multiplatformné ovládače, ovládače kompatibilné so 64-bitovou architektúrou a lokalizáciu ovládačov do rôznych jazykov.

WDK je vhodný i pre návrh ovládačov pre prácu v jadre systému. Obsahuje i ukážky takýchto ovládačov. Poskytuje prostriedky pre tvorbu WDM (Windows Driver Model), KMDF (Kernel-mode driver framework) a UMDF (User-mode driver framework) ovládačov, ku ktorým sú zahrnuté i ukážky zdrojových kódov (tieto typy sú vysvetlené v 6.3).Ďalej poskytuje prostriedky pre rôzne typy ovládačov podľa technológie, na ktorej sú založené.

Obsahuje nástroje pre tvorbu INF súborov, konkrétne ChkInf a Stampinf. Ďalšie nástroje sú pre vytváranie ovládačov – BinPlace, Build, MakeDirs, Microsoft Auto Code Review (OACR). Posledný menovaný sa využíva čiastočne pri použití SDV. Nasledujúcimi nástrojmi sú programy pre podpisovanie ovládačov – CertMgr, Inf2Cat, MakeCat, MakeCert, Pvk2Pfx, Sign Tool. Obsiahla skupina nástrojov sa zaoberá testovaním – Data Execution Prevention Demo Program, DevCon, Device Path Exerciser, Driver Coverage Toolkit, Enhanced Storage Certificate Management Tool, IoSpy a IoAttack, Plug and Play Driver Test, PNPCPU, PoolMon, PwrTest, Windows Biometric Framework Tools, WSDAPI Basic Interoperability Tool, XpsAnalyzer.

Po tejto časti nasleduje najdôležitejšia časť týchto nástrojov. Týka sa verifikačných nástrojov. Patria sem PREfast (Kapitola 5), Static Driver Verifier (Kapitola 6), WDF Verifier Control Application, WdfTester. Prvým dvom menovaným budú venované samostatné kapitoly, pretože sa používajú pri verifikácii zvoleného ovládača. Ďalej obsahuje nástroje pre simuláciu zariadení, kde sa využíva simulačný systém pre USB zariadenia.

5 PREfast

V tejto kapitole bude vysvetlený princíp fungovania nástroja PREfast na ovládači, ktorý bol vybraný z príkladov uvedených vo WDK. Ešte najskôr bude nasledovať krátky popis vlastností.

5.1 Charakteristika nástroja

PREfast je nástroj, ktorý využíva statickú analýzu a používa sa na detekciu chýb, ktoré neboli odhalené pri kompilácii programu. Tento nástroj je odlišný od ostatných tým, že každú funkciu v kóde berie ako jeden samostatný celok a a prechádza všetky možné cesty, ktorými by sa mohla funkcia vyhodnotiť. Pri tomto prechode hľadá chyby ovládača ako aj nepraktické programovacie praktiky. Podľa WDK dokumentácie je nástroj rýchly i pre veľké ovládače. Podľa testov je značne rýchly pre menšie typy ovládačov, pri ktorých je doba jeho behu kratšia ako fáza skompilovania ovládača. Avšak pri väčších ovládačoch je doba jeho behu výrazne dlhšia ako doba kompilácie. Výsledkom jeho prechodu cez kód je tabuľka so zistenými chybami alebo varovaniami.

Spúšťanie PREfastu je automatické a vykonáva sa cez Windows Auto Code Review (OACR), ktorý beží na pozadí pri spustení Build Environmentu.

5.2 Príklad spustenia PREfastu

Ako referenčný príklad bude slúžiť chybný ovládač, ktorý sa neskôr použije i ako ukážka pre Static Driver Verifier. Príklad, ktorý tu je interpretovaný pochádza z WDK príkladov pre SDV. Nachádza sa v zložke:

```
{adresár WDK}\tools\sdv\samples\fail_drivers\wdm\fail_driver2
```

Programový kód 1: adresár s predvádzaným príkladom

Ide o jednoduchý WDM[5] ovládač, ktorý slúži len na demonštráciu fungovania SDV a nie je spustiteľným ovládačom. Obsahuje i knižnicu, ktorú je potrebné pripojiť, pre správne fungovanie (vysvetlené v kapitole 6.9). Konkrétne obsahuje funkcie, pre pridanie zariadenia, jeho nastavenie, odoslanie a prijatie požiadavky, kontrolu zariadenia a jeho odpojenie. PREfast sa v tomto prípade spúšťa automaticky po zavolaní SDV s príkazom `/scan`. Zavolá sa ako časť OACR[6], a automaticky prevedie kontrolu ovládača. Pre vynútené zavolanie PREfastu je nutné zadať príkaz `prefast -parametre`[7]. Tieto príkazy budú vysvetlené v kapitole o SDV, ktorého sa bezprostredne týkajú.

Postup pri spustení Build Environmentu:

1. Z ponuky Štart sa vyberie položka Windows Driver Kits, z ktorej sa vyberie verzia WDK.
2. Nasledujúci výber je jednoznačný – „Build Environment“.
3. Ďalej sa vyberie zložka s názvom systému, v ktorom sa bude spúšťať.
4. Na výber je potom spustenie rôzneho typu tohto programu (vysvetlené ďalej), výber závisí na type architektúry, ktorú počítač využíva. Na výber je vždy prostredie

[5] Vysvetlenie pojmu je v 6.3
[6] Windows Auto Code Review: sada nástrojov vykonávajúcu statickú analýzu nad ovládačmi, zapája do WDK PREfast. Automaticky sa spúšťa pri každom spustení WDK.
[7] Pre ďalšie informácie o ovládaní PREfastu vynútene: http://msdn.microsoft.com/en-us/library/ms925496.aspx

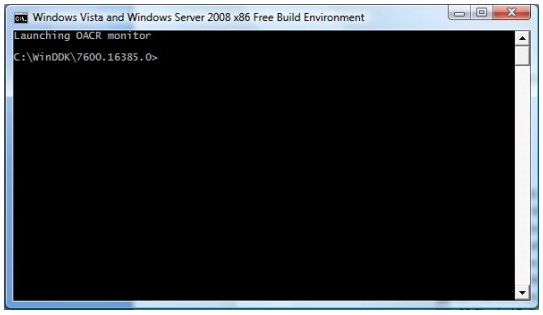
Checked alebo Free. V našom prípade to neovplyvní výsledky, ale v prípadoch komplexnejšieho ovládača je rozumné sa zamyslieť nad výberom (vysvetlené ďalej).

5. Po výbere sa spustí okno s príkazovým riadkom (Obrázok 2).

Pri výbere Free Build Environment je podľa WDK dokumentácie aplikovaná optimalizácia kódu tak, aby najviac vyhovoval práve vydaniu na ktorom sa spúšťa. U Checked Build Environment sa kód optimalizuje tak, že sú pridané nové schopnosti, ktoré napomáhajú testovaniu a ladeniu.

Štruktúra zložky obsahujúcej spúšťanie Build Environmentu (BE): za zarážkou je vždy uvedený prvok, ktorý obsahuje a za ním jeho popis.

- Windows Driver Kits: zložka - obsahuje BE.
  - WDK <verzia>: zložka - verzia Windows driver kitu.
    - Build Environment: zložka
      - Windows 7: zložka: obsahuje BE pre vypísaný systém.
        - ia64 Free BE: spustí BE vo verzii Free pre procesor AMD64.
        - ia64 Checked: spustí BE vo verzii Checked pre procesor AMD64.
        - x64 Free/Checked BE: spustí BE vo verzii Free/Checked pre procesor Intel 64-bit.
        - x86 Free/Checked BE: spustí BE vo verzii Free/Checked pre procesor Intel 32-bit.
    - Device Simulation Framework: zložka – štandardne obsahuje dokumentáciu, pre využitie simulačného prostredia je nutné ho doinštalovať.
    - Help: zložka s dokumentáciou k WDK.
    - Tools: zložka – obsahuje odkaz na stránku so zoznamom nástrojov vo WDK.



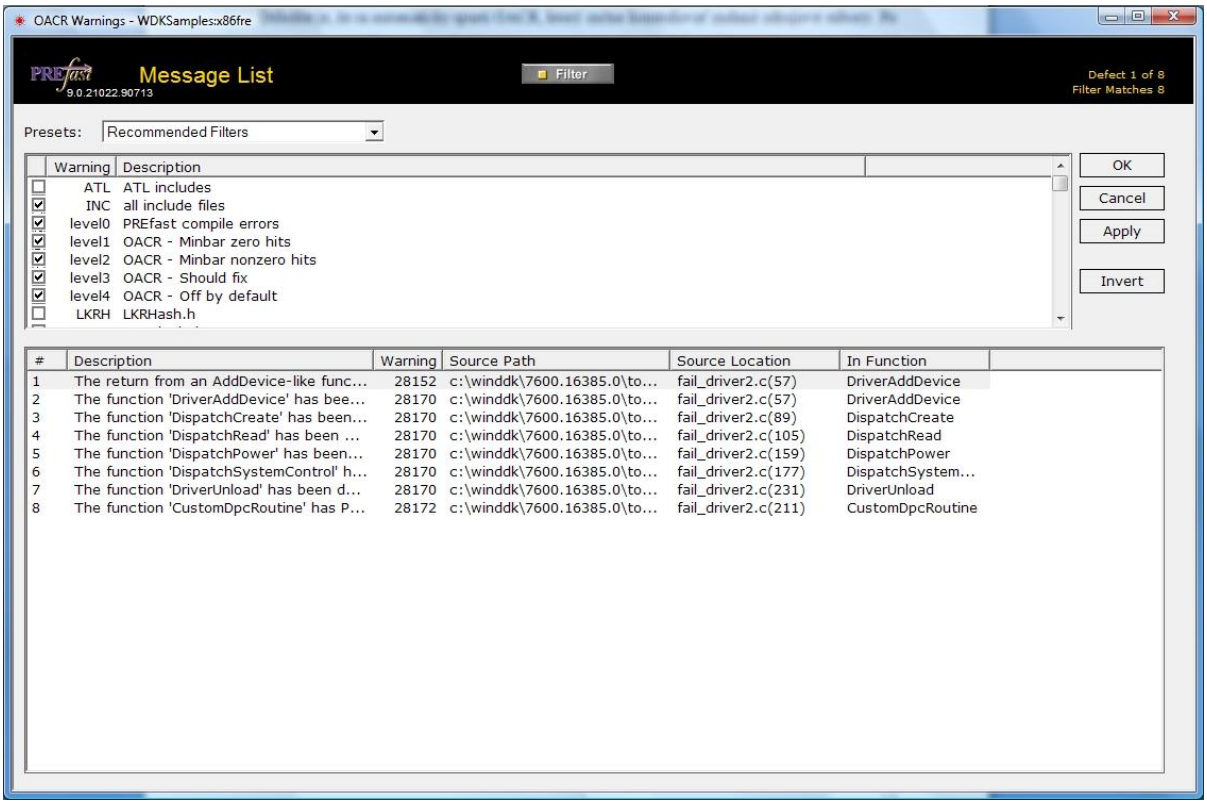
Obrázok 2: Spustený Build Environment

Na začiatku sa spúšťa OACR monitor. Tento program automaticky volá nástroj PREfast. Jeho aktivitu je možné pozorovať v paneli pri hodinách. Môže sa nachádzať v troch stavoch vzhľadom k jeho výsledkom (Obrázok 3) alebo v troch stavoch vzhľadom k jeho aktivite. Pri neaktívnosti je zobrazený v jeho ikone v pravom dolnom rohu znak Windows, pri príprave na začatie kontroly sa tento znak zmení na štýl tlačidla Play, pri vykonávaní analýzy je tento symbol zmenený na štýl tlačidla Pause.

Obrázok 3: Stavy vzhľadom k výsledkom

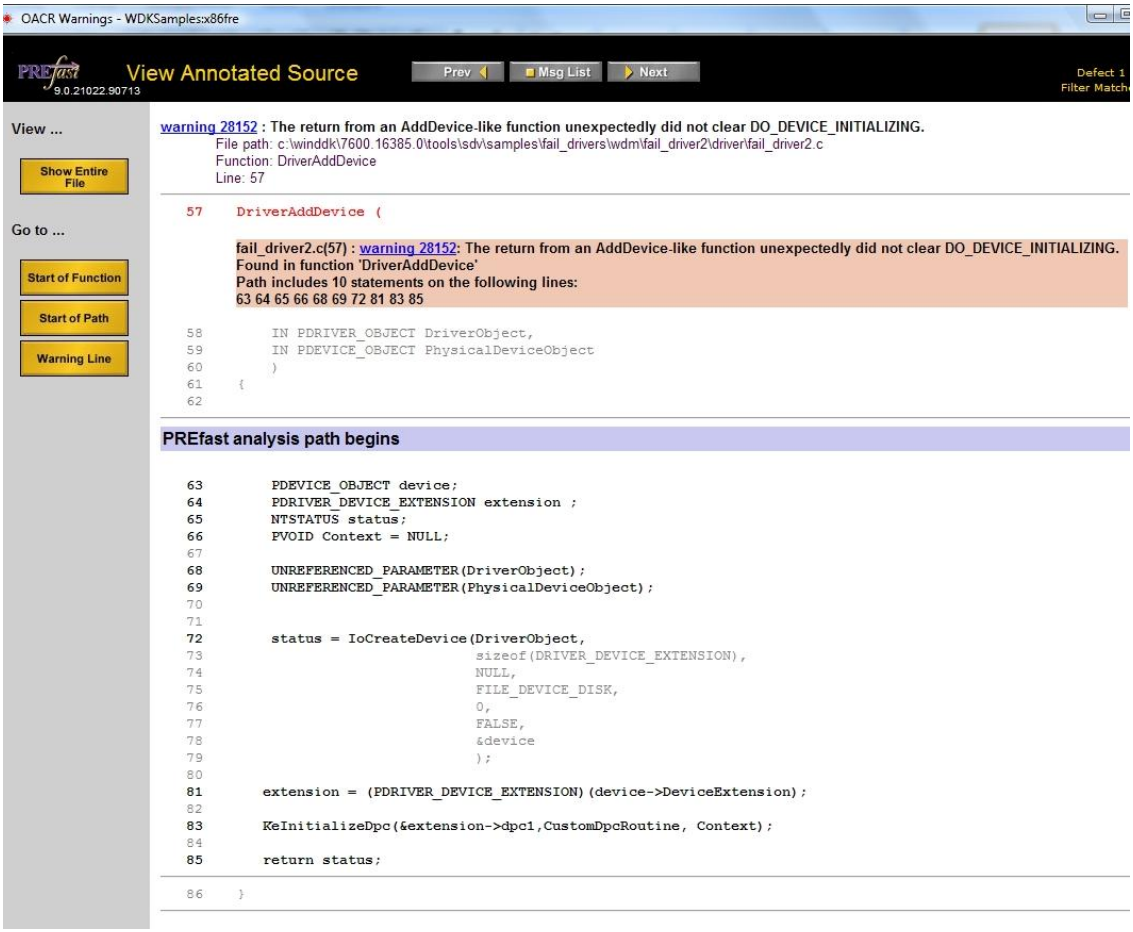
Na Obrázok 3 je vidieť stavy ,vzhľadom k výsledkom, v ktorých sa môže nachádzať OACR. Ak je ikona zelená, tak žiadna chyba alebo varovanie nebolo nájdené alebo OACR nebol ešte spustený. Červená označuje, že zdrojové kódy obsahujú chyby, ktoré bude treba vyriešiť. Žltá farba signalizuje, že sa vyskytli varovania, ktoré by bolo vhodné opraviť.

Po spustení Build Environmentu je nutné prejsť do priečinka so zdrojovými kódmi ovládača a spustiť SDV pomocou príkazu `staticdv /scan`. Vysvetlenie tohto kroku bude v nasledujúcej kapitole. Dôležité je, že sa automaticky spustí OACR, ktorý začne kontrolovať zadané zdrojové súbory. Po skončení sa v pri ikone v paneli objaví upozornenie, že OACR dokončil kontrolu a môže zobraziť výsledky. Zobrazí ich v okne, ktoré je na Obrázok 4.



Obrázok 4:Výpis PREfastu

Popis okna s výsledkami PREfastu: V ľavom hornom rohu je vidieť označenie, že sa naozaj volal nástroj PREfast. V strede obrázku je vidieť tlačítko Filter, po ktorého stlačení sa otvorí lišta, ktorá je vidieť i na obrázku (je to plocha vedľa tlačidiel OK, Cancel). Pod týmto filtrom je vidieť varovania, ktoré našiel PREfast v ovládači, konkrétne v zdrojovom súbore *fail_driver.c* . Chyby sú implicitne radené podľa toho, v akom poradí boli nájdené. Táto tabuľka s chybami obsahuje šesť stĺpcov. V prvom je číslo riadku tabuľky, v druhom je popis chyby, v treťom je číslo varovania, vo štvrtom je cesta k zdrojovému súboru, v piatom je názov zdrojového súboru, v ktorom bolo varovanie nájdené spolu s riadkom, na ktorom sa nachádza a v poslednom stĺpci je názov funkcie, kde sa varovanie vyvolalo. V názve tohto okna je nadpis OACR Warnings, čiže v tomto ovládači neboli nájdené chyby a môže sa pokračovať v procese verifikácie.



Obrázok 5: Náhľad varovania v okne OACR

V okne s výpismi chýb po dvojkliku na jednu z chýb, resp. varovaní sa zobrazí náhľad tohto varovania priamo v okne i s presnejším popisom varovania a odkazom do dokumentácie pre ešte presnejšiu identifikáciu varovania (Obrázok 5).

Varovania sú len upozornenia na nesprávny zápis kódu, resp. chýbajúce inicializácie, ktoré by z celkového hľadiska nemali mať veľký vplyv na výsledky verifikácie. Ak by OACR našlo chyby v zdrojových súboroch, tak by sa pravdepodobne nemohlo pokračovať, pretože by sa zdrojové súbory nepodarilo skompilovať a tým by nemohol prejsť proces verifikácie.

Po opravení chýb je možné kontrolu zdrojových súborov znova vyvolať kliknutím pravým tlačidlom myši na ikonu OACR a najskôr vybrať položku Clean a v nej meno projektu, ktorý sa kontroluje a vymazať nájdené chyby. Tým by sa mala farba ikony zmeniť na zelenú. Rovnakým spôsobom sa potom z menu vyberie Check Now. Existuje i ďalší spôsob ako vyvolať znovu kontrolu súborov. Stačí spustiť príkaz SDV `staticdv /scan` a OACR sa znova automaticky spustí.

Týmto úloha PREfastu končí a môže nastúpiť proces kontroly modelu (angl. model checking).

6 Static driver verifier

Táto kapitola oboznámi s najdôležitejšou časťou projektu, a to nástrojom pre kontrolu zdrojových súborov ovládača pomocou model checkingu – SDV. Najskôr bude uvedený stručný popis vlastností a charakteristiky nástroja. Potom obmedzenia na použitie s rôznymi typmi ovládačov a obmedzenia týkajúce sa zdrojových súborov. Ďalej budú uvedené príkazy, ktoré používa Static Driver Verifier (SDV), princíp písania pravidiel pre verifikáciu, nutné úpravy súbory pred verifikovaním.

6.1 Definície pojmov

SDV pravidlo definuje požiadavku na správnu komunikáciu medzi ovládačom a rozhraním jadra. Pravidlá sú napísané v špeciálnom jazyku SLIC (Specification Language for Interface Checking) vyvinutou firmou Microsoft .

SDV model operačného systému pozostáva z čiastočných a abstraktných kódov Windows, ktoré sa správajú počas verifikácie ako operačný systém. SDV zostavuje operačný systém počas fázy Check verifikačného procesu.

SDV verifikuje, či aktuálne správanie ovládača vyhovuje pravidlu, ktoré definuje správne chovanie.

Verifikačný proces pozostáva z troch častí:

Prvou časťou je Build. SDV vytvorí, skompiluje a nalinkuje ovládač pomocou štandardného programu vo WDK.

Druhou časťou je Scan. Počas tejto časti SDV prechádza zdrojový kód a vyhľadáva typy rolí (vysvetlené v 6.6) vo funkciách (angl. role type). Potom zostaví zoznam vstupných bodov a vytvorí súbor *Sdv-map.h* v adresári so zdrojovými kódmi kde bol SDV spustený.

Treťou časťou je Check. SDV v tejto fáze použije pripravené pravidlá. Najskôr skontroluje, či pravidlo nepotrebuje dodatočné komponenty operačného systému. Potom vytvorí jeden spustiteľný súbor zo zdrojových súborov, knižníc, súboru s pravidlami a modelu operačného systému. Nasleduje samotná verifikácia, kedy prechádza verifikačný program postupne jedno pravidlo za druhým. Pre každé verifikované pravidlo vytvorí zložku v umiestnení ovládača v priečinku /sdv/check dir.

Tieto definície boli voľne prevzaté z [11].

6.2 Charakteristika nástroja

SDV je nástroj používajúci model checking tak, že preskúma kód ovládača. Preskúmanie kódu prebieha symbolickým vykonaním kódu s najmenším možným predpokladom o stave systému a počiatočného stavu ovládača. Tým SDV nájde cesty, ktoré môžu byť vynechané pri normálnom testovaní. Pri každej verifikácii SDV dostáva sadu pravidiel, ktoré definujú správnu komunikáciu medzi jadrom systému a ovládačom. Počas priebehu verifikácie sa SDV snaží nájsť každú aplikovateľnú cestu zdrojovým kódom a kódom knižnice tak, aby dokázal, že ovládač porušuje stanovené pravidlá. Ak SDV neuspeje s nájdením porušenia pravidiel, označí daný ovládač za verifikovaný.

6.3 Typy ovládačov

Keďže v ďalších kapitolách budú často používané skrátené termíny označovania ovládačov, je potrebné definovať ich mená a spomenúť ich odlišnosti. Prebrané z [12].

KMDF (Kernel mode driver framework) : ide o knižnicu, ktorú je možné využiť pre vytváranie ovládačov do jadra systému, ktoré podporujú WDM ovládače. Využívajú rozhranie, ktoré im dodáva prostredie (angl. framework). Toto prostredie ďalej komunikuje s operačným systémom pomocou WDM rozhrania. Poskytuje ovládaču jednoduchšie rozhranie ako WDM a zároveň rieši veľa operácií, ktoré by musel WDM ovládač spracovávať sám. Poslednou výhodu je, že poskytuje synchronizačný kód, ktorý je potrebný pri podpore multiprocesorového prostredia.

WDM (Windows driver model): bol vytvorený, aby vývojári mohli vyvíjať ovládače, ktorých zdrojové kódy budú kompatibilné vo všetkých operačných systémoch Windows.

NDIS (Network Driver Interface Specification): ide o rozhranie, ktoré sa používa na komunikáciu s ovládačmi sieťových kariet. Väčšina funkcionality linkovej vrstvy OSI (Opens Systems Interconnection) modelu je implementovaná pomocou tohto rozhrania.

6.4 Obmedzenia SDV

V skratke budú uvedené len niektoré významnejšie obmedzenia[8], pri ktorých nemôže pracovať. Ide o vlastnosti buď samotného SDV alebo ovládača, pre ktoré nemôže prebehnúť proces verifikácie. Táto časť vychádza z [13].

Všeobecné obmedzenia:

- SDV dokáže verifikovať len jeden ovládač naraz.
- Verifikácia nemôže prebiehať zo vzdialeného miesta pripojeného cez sieť, zdrojové súbory musia byť lokálne.
- Len jeden proces SDV môže bežať v čase, nedokáže bežať paralelne ani konkurentne.
- SDV je lokalizované v angličtine, preto do verifikácie nezahŕňa elementy, ktoré sú závislé na lokálnom formátovaní, napríklad formátovanie reťazca.

Obmedzenia verifikačného nástroja:

- Nerozpoznáva, že 32-bitové čísla dátového typu integer sú limitované 32 bitmi, preto nerozpoznáva chybu pretečenia a podtečenia.
- Ak sú vstupné body definované ako static, je nutné ich upraviť v súbore *SDV-map.h*.
- Ignoruje aritmetiku ukazateľov a únie.
- Nerozpoznáva pretypovanie.
- Vždy používa prvý element v poli, nehľadiac na to, že bol volaný iný element poľa.
- Nepodporuje modulo (%) operátor.

6.5 Podporované typy ovládačov

Pre správne verifikovanie ovládač musí spĺňať túto základnú charakteristiku:

- SDV analyzuje ovládače napísané len v jazyku C a so zdrojovými súbormi s príponou .c.
- WDM ovládače dokáže verifikovať ak ich funkcie typu dispatch sú definované pomocou postavenia typov (angl. role type) z SDV-WDM.
- Ovládač musí mať menej než 50 000 riadkov kódu vrátane knižníc, ktoré sú časťou verifikácie.

[8] Všetky obmedzenia je možné nájsť v [13].

- Nemusí byť schopné verifikovať veľmi komplexné ovládače alebo ovládače z veľmi veľa závislosťami na knižniciach.

Základné požiadavky na ovládač:

Pre WDM ovládače:

- Musia mať zahrnutú hlavičkový súbor *wdm.h* alebo *ntddk.h*.
- Musia vytvárať objekty pomocou metód na to určených v dokumentácii SDV.
- Majú rutinu `Unload`, ktorá je odporúčaná.
- Každá funkcia typu dispatch je deklarovaná pomocou postavenia typov.

Pre ostatné typy ovládačov požiadavky neuvádzam, pretože ovládač, ktorý budem verifikovať patrí do skupiny WDM ovládačov.

Rezervované mená, ktoré používa SDV:

- Kód obsahuje meno funkcie začínajúce s `_init` a pokračujúce číslami. Napríklad _init123.
- Kód obsahuje meno funkcie začínajúce s `sdv_`, alebo obsahuje reťazec _sdv_
- Knižnica využíva .def súbor na premenovanie exportovanej funkcie a externé meno je rovnaké ako meno statickej funkcie v knižnici.

Voľne prebrané z [14].

6.6 Role types

Táto podkapitola ozrejmí tento pojem, pretože má zásadný vplyv na verifikačný proces. Prakticky pri každom type ovládača je nutné použiť typ role (angl. role type), ktorý označuje vstupný bod ovládača.

Typ role je prakticky typ funkcie, ktorý neoznačuje aký dátový typ funkcia vracia, ale do akej skupiny funkcia patrí. Podľa toho je možné pre PREfast i SDV jednoducho definovať vstupný bod ovládača, keď pozná typ funkcie, resp. typ role, ktorú funkcia v danom ovládači zohráva. Majú presnú syntax v definícii funkcií typu callback (u KMDF) a dispatch (u WDM). Tieto deklarácie prepisujú tradičné deklarácie funkcií. Táto časť sa bude zaoberať definovaním typov rolí pre WDM ovládače, pretože tento typ sa bude verifikovať.

6.6.1 WDM role types

Použitie definícií pre typy rolí WDM je podmienené zahrnutím hlavičkového súboru *wdm.h* alebo *ntddk.h,* v ktorých sú tieto typy už preddefinované. V Tabuľka 1 sú zobrazené základné preddefinované typy a typy, ktoré sa majú prepisovať. Tabuľka je prevzatá z dokumentácie k WDK.

V prvom stĺpci tabuľky je typ role, ktorý sa má použiť pre danú funkciu, ktorá je uvedená v druhom stĺpci. V Programový kód 2 je vysvetlený zápis typu rolí. V programovom kóde naľavo je definovanie jednoduchého typu role. Vidíme, že `DriverObject` pristupuje k nejakej funkcii `DriverStartIo`. Podľa tabuľky ide o funkciu (konkrétne 2. Riadok tabuľky), ktorej je treba definovať typ role, pretože je to možný vstupný bod. Definovanie typu role sa vykoná tak, že funkcii preddefinujeme typ podľa tabuľky a to `DRIVER_STARTIO`, tým bude verifikačný nástroj (PREfast alebo SDV) vedieť, že môže využiť tento vstupný bod.

DRIVER_INITIALIZE	*DriverEntry*
DRIVER_STARTIO	*StartIO*
DRIVER_UNLOAD	*Unload*
DRIVER_ADD_DEVICE	*AddDevice*
__drv_dispatchType(*typ* **) DRIVER_DISPATCH**	Táto rutina je používaná ovládačom. **__drv_dispatchType(***typ* **)** annotácia musí byť skombinovaná s postavením typu DRIVER_DISPATCH, aby určovala vstupné body ovládača.
IO_COMPLETION_ROUTINE	*IoCompletion* *Rutina IoCompletion* je nastavená volaním **IoSetCompletionRoutine** alebo **IoSetCompletionRoutineEx** a predaním ukazateľa na funciu do *IoCompletion* rutiny ako druhý parameter.
DRIVER_CANCEL	*Cancel* Rutina *Cancel* je nastavená volaním **IoSetCancelRoutine** a predaním ukazateľa na funkciu do rutiny na zrušenie pre IRP ako druhý parameter funkcie.
IO_DPC_ROUTINE	*DpcForIsr* Rutina *DpcForIs*r *je* registrovaná volaním **IoInitializeDpcRequest** a predaním ukazateľa na funkciu do rutiny *DpcForIsr* routine ako druhý parameter. Pre zaradenie DPC je nutné zavolať **IoQueueDpc** z rutiny ISR použitím rovnakého DPC objektu.
KDEFERRED_ROUTINE	*CustomDpc* Rutina *CustomDpc* je nastavená volaním **KeInitializeDpc** a predaním ukazateľa na funkciu do *CustomDpc* ako druhý parameter. Pre zaradenie *CustomDpc* pre ovládač je nutné zavolať **KeInsertQueueDpc** z rutiny ISR použitím rovnakého DPC objektu.
WORKER_THREAD_ROUTINE	*Routine* *Routine* je typu callback, čo je špecifikované druhým parametrom pre funciu **ExInitializeWorkItem.** Routine musí byť deklarovaná len týmto spôsobom **ExQueueWorkItem** pre pridanie položky (Work Item) do radu.

Tabuľka 1: Definovanie typov rolí pre WDM ovládače

Pri aplikácii typov rolí je nutné dať pozor na funkcie typu dispatch. Pri nich sa musí doplňovať ešte riadok, ako je uvedené v tabuľke, kde sa definuje vstupne/výstupný kód. V Programový kód 2 v pravom rámčeku je definovaný vstupne/výstupný kód pomocou riadku `__drv_dispatchType(IRP_MJ_PNP)`, kde `IRP_MJ_PNP` označuje daný vstupne/výstupný kód, ktorý v tomto prípade označuje správu Plug and Play zariadenia. Za týmto riadkom nasleduje normálna definícia typu role pre funkciu. Príklad pre jednoduchú definíciu typu role bol uvedený skôr. Nasleduje príklad prevzatý z dokumentácie WDK.

```
DriverObject->DriverStartIo
DriverObject->Unload
DriverObject->DriverExtension->AddDevice


DRIVER_STARTIO DriverStartIo;
DRIVER_UNLOAD Unload;
DRIVER_ADD_DEVICE AddDevice;
```

```
__drv_dispatchType(IRP_MJ_PNP)
DRIVER_DISPATCH FilterDispatchPnP;


__drv_dispatchType(IRP_MJ_POWER)
DRIVER_DISPATCH FilterDispatchPower;


__drv_dispatchType_other
DRIVER_DISPATCH FilterPass;
```

Programový kód 2: Príklad jednoduchého definovania typov rolí a definovania typov dispatch

6.7 Princíp verifikačných pravidiel pre SDV

Pravidlá, ktoré má dodržiavať ovládač a verifikačný program, ktorý ich má kontrolovať, sú rozdelené znova do troch skupín podľa typu ovládača, pre ktorý sú napísané. Tu budú uvedený len ten typ pravidiel, ktorý sa týka ovládača, ktorý sa bude verifikovať.

Všeobecne o pravidlách platí, že buď sa verifikuje len jedno pravidlo, ktoré sa zadá do SDV cez príkaz `/rule:NazovPravidla` alebo sa vytvorí samostatný súbor s pravidlami *config.sdv*. V ňom bude na každom novom riadku jedno pravidlo.

Pravidlá sú napísané v špeciálnom jazyku SLIC. V nasledujúcej ukážke kódu (Programový kód 3) je práve v tomto jazyku zapísané pravidlo pre typ ovládača WDM – `AddDevice`.

Konkrétne toto pravidlo kontroluje, či pri pridávaní zariadenia ovládač volá všetky požadované funkcie. V tomto prípade je v ukážke Programový kód 3 vidieť, že sa inicializuje premenná `s`, ktorá vyjadruje stav pravidla (riadok 3). Môže nadobúdať stavy `INIT` – inicializovaná, `ENTERED` – vstúpenie do funkcie, `CREATED` – vytvorený ukazateľ, `ATTACHED` – pripojené zariadenie. Ďalej na riadku 5 je definovaná akcia ak je pridané zariadenie. Na riadku 9 je znova definovaná akcia, kde sa na začiatku kontroluje vstúpenie do funkcie pre vytvorenie ukazateľa na zariadenie. Riadok 16 obsahuje definíciu stavu, kedy sa zariadenie posiela na zásobník a kontroluje sa, či bol jeho ukazateľ naozaj vytvorený. Ak nie, pravidlo pre ovládač neplatí a SDV našiel chybu. Ak áno, tak sa do premennej priradí hodnota `ATTACHED`.

```
1   #include "ntddk_slic.h"
2   state{
3     enum {INIT, ENTERED, CREATED, ATTACHED} s = INIT;
4   }
5   fun_AddDevice.entry
6   {
7     s = ENTERED;
8   }
9   [IoCreateDevice
10  ,IoCreateDeviceSecure].entry
11  {
12    if (s == ENTERED) {
13      s = CREATED;
14    }
15  }
16  IoAttachDeviceToDeviceStack.entry
17  {
18    if (s == CREATED) {
19      s = ATTACHED;
20    } else {
21      abort "The driver is calling IoAttachDeviceToDeviceStack, but it has not first
22      called IoCreateDevice.";
23    }
24  }
```

Programový kód 3: ukážka pravidla pre verifikáciu zo súboru AddDevice.slic

Pravidlá pre WDM ovládače sa delia do siedmich skupín, pri použití niektorého z pravidiel bude pri ňom uvedený jeho typ, do ktorej skupiny patrí a jeho činnosť. Na základe týchto jeho vlastností je možné odvodiť i vlastnosti celej skupiny:

1. IRP pravidlá: slúžia na dokázanie, že ovládač korektne používa funkcie, ktoré sa starajú o vstupne/výstupné požiadavkové pakety.
2. IRQL (Interrupt request level) pravidlá: slúžia na dokázanie, že funkcie v ovládači pracujú na správnej úrovni prerušenia.
3. PnP (Plug and Play) pravidlá: slúžia na dokázanie, že ovládač používa konvencie spravovania PnP zariadení správne.
4. Synchronizačné pravidlá: slúžia na dokázanie, že ovládač správne využíva zdieľané zdroje.
5. WMI (Windows Management Instrumentation) pravidlá: slúžia na dokázanie správnosti práce s WMI, ktoré sú podmnožinou IRP.
6. Všeobecné pravidlá: umožňujú skontrolovať či ovládač správne pracuje s registrovými kľúčmi, reťazcami a ukazateľmi na objekty zariadení.
7. Pravidlá vlastností zariadenia: kontrolujú možnosti ovládača a podporované prvky pre využitie v budúcnosti.

6.8 Ovládanie nástroja

SDV sa ovláda z Build Environmentu, ktorý sa používa rovnako ako príkazový riadok. Pre vyvolanie verifikačného nástroja je nutné zavolať `staticdv`. K tomuto príkazu je potrebné pripojiť určitý prepínač, ktorý zariadi, aby SDV vykonal zadanú vec. Build Environment musí byť samozrejme v adresári, v ktorom sa nachádzajú zdrojové súbory ovládača. Tento postup verifikácie bude vysvetlený v nasledujúcej kapitole. Syntax SDV je zobrazená v nasledujúcom kóde (Programový kód 4). Kód je prevzatý z dokumentácie WDK. Za ním nasleduje výpis prepínačov, pri ktorých je vždy uvedené akú funkciu zapínajú. Nasledujúce informácie sú voľne prebraté z [15].

```
staticdv [/rule:{Rule | *} ] [ /refine /rule:{Rule | * } ]
         [/config:RuleList.sdv ] [ /refine /config:RuleList.sdv ]
         [/lib ]
         [/view ]
         [/scan ]
         [/clean [/force] ]
         [/cleanalllibs]
         [/? ] [/help ] [/showrules ]
```

Programový kód 4: Syntax nástroja Static Driver Verifier

Prepínače pre Static Driver Verifier:

- `/scan` – preskenuje zdrojové súbory ovládača s cieľom nájsť postavenie typov (angl. role types) a určiť vstupné body ovládača. Výsledok sa uloží do súboru *Sdv-map.h*, ktorý sa uloží do priečinka kde bol volaný SDV.
- `/rule:{Rule | *}` – začne verifikovať ovládač na základe zadaného pravidla, namiesto niektorých znakov je možné v názve pravidla použiť * a tým sa na tomto mieste môže objaviť hocijaké písmeno. Tým sa docieli aplikácia určitého typu pravidiel. Použitím len hviezdičky sa aplikujú všetky pravidlá. Vždy sa zadáva len jedno pravidlo.

- `/config:RuleList.sdv` – začne proces verifikácie a aplikuje pravidlá špecifikované v zadanom súbore. V súbore je možné použiť * ako v predchádzajúcom prípade.
- `/refine` – rozdelí proces verifikácie tak, že sa bude začínať po každom novom vstupnom bode nová verifikácia od začiatku.
- `/lib` – používa sa na preklad knižníc, ktoré daný ovládač potrebuje. Spúšťa sa v adresári knižnice, ktorá sa má preložiť.
- `/view` – otvorí SDV Report, správu, v ktorej je zahrnutý výsledok verifikácie.
- `/clean` – odstráni z aktuálne zvoleného priečinka všetky súbory vytvorené SDV, ale ak je povolený súbor *SDV-map.h*, tak ten odstraňovať nebude.
- `/force` – slúži rovnako na vymazanie súborov po SDV, avšak používa sa ak prepínač /clean nefunguje.
- `/cleanalllibs` – odstráni všetky knižnice, ktoré preložil SDV na celom disku počítača, tento proces neovplyvňuje dáta ovládača.
- `/?` – zobrazí použitie príkazov SDV, príkazy, ktoré používajú tento parameter nemusia byť spúšťané v Build Environment.
- `/help` – zobrazí dokumentáciu programu SDV, príkazy, ktoré používajú tento parameter nemusia byť spúšťané v Build Environment.
- `/showrules` – zobrazí sekciu dokumentácie týkajúcej sa pravidiel, príkazy, ktoré používajú tento parameter nemusia byť spúšťané v Build Environment.
- Ak sa SDV spustí bez parametrov zobrazí použitie jeho príkazov.

6.9 Príklad verifikačného postupu

Ako príklad poslúži rovnaký typ ovládača (popísaný v 5.2), na ktorom bol prezentovaný postup OACR(vysvetlený v 6) a PREfastu. SDV využíva pre preklad zdrojových súborov podobný spôsob, ako sa bežne používa v jazyku C a iných podobných jazykoch, so súborom *Makefile*. Na rozdiel od nich názov súboru pre SDV je *sources*. Tento súbor sa musí nachádzať v každom priečinku, kde sa má volať SDV.

Prvým krokom pri verifikácii je nalinkovanie knižníc. Spúšťa sa pomocou príkazu `staticdv /lib` v priečinku so súbormi knižnice. Vytvorí sa knižnica (preložením súborov uvedených v súbore sources), ktorú bude SDV pripájať pri skenovaní ovládača a model checkingu. Preklad by sa mal úspešne potvrdiť, ako to je vidieť v nasledujúcej ukážke kódu (Programový kód 5).

```
------------------------------------------------------------------------
Microsoft (R) Windows (R) Static Driver Verifier Version 2.0.372.0
Copyright (C) Microsoft Corporation.  All rights reserved.
------------------------------------------------------------------------
Build     'library'  ...Done
Publishing :fail_library2.lib.li to
c:\winddk\7600.16385.0\tools\sdv\samples\fai
l_drivers\wdm\fail_driver2\library\objfre_wlh_x86\i386\fail_library2.lib.li
The cache contains  4 libraries.
Compiled   'fail_library2.lib'  ...Done
```

Programový kód 5: Kompilácia knižnice pred samotnou verifikáciou

Je dôležité dodržať presný postup kompilovania knižníc, pretože i tie môžu na sebe závisieť. Najlepším overením správneho postupu pri kompilovaní jednotlivých knižníc je otvoriť súbor sources v adresári zdrojových súborov ovládača a hľadať parameter `TARGETLIBS`. Za ním by mali nasledovať knižnice, ktoré sa používajú a linkujú k hlavným zdrojovým súborom. Ich poradie za týmto parametrom je dobré dodržať a tak sa vyhnúť zbytočným problémom.

Ďalším krokom je skompilovanie zdrojových súborov pomocou príkazu `staticdv /scan`. Pri tomto príkaze sa automaticky spustí PREfast, jeho funkčnosť bola uvedená už skôr. Vykonávajú sa tu i nasledovné funkcie nezávisle od behu PREfastu. Prvou časťou je Build zdrojových súborov – zdrojové súbory sa preložia. Ďalšou časťou je nalinkovanie knižnice a poslednou je Scan, ktorý podľa predošlej časti tejto kapitoly, vytvorí súbor *SDV-map.h,* v ktorom sú definované typy, podľa ktorých bude SDV vedieť, ku ktorým funkciám má pristupovať (Programový kód 6).

```
------------------------------------------------------------------------
Microsoft (R) Windows (R) Static Driver Verifier Version 2.0.372.0
Copyright (C) Microsoft Corporation.  All rights reserved.
------------------------------------------------------------------------
Build     'driver'  ...Done
Link      'driver' for [fail_library2.lib] ...Done
Scan      'driver'  ...Done
Please review and approve the entry points in SDV-map.h.
```

Programový kód 6: Výpis SDV po príkaze staticdv /scan

Vo výpise (v Programový kód 6) je uvedené, že treba potvrdiť súbor *SDV-map.h* (Programový kód 7), v ktorom sú uložené vstupné body. Keď si otvoríme tento súbor tak v ňom sú podobné riadky, aké boli zadávané pri definovaní typov rolí (angl. role types), avšak na začiatku majú ešte pridané slovo `#define`. Úlohou človeka je skontrolovať, či SDV našlo všetky vstupné body, ktoré sú zapísané v zdrojových súboroch. Pre potvrdenie súboru stačí prvý riadok súboru prepísať na `//Approved=true` . Tým súbor overíme a môžeme pokračovať verifikáciou.

```
//Approved=false
#define fun_DriverUnload DriverUnload
#define fun_IRP_MJ_SYSTEM_CONTROL DispatchSystemControl
#define fun_IRP_MJ_WRITE DispatchWrite
#define fun_IRP_MJ_POWER DispatchPower
#define fun_DriverEntry DriverEntry
#define fun_IRP_MJ_READ DispatchRead
#define fun_AddDevice DriverAddDevice
#define fun_KSERVICE_ROUTINE_1 InterruptServiceRoutine
#define fun_IRP_MJ_CREATE DispatchCreate
#define fun_KDEFERRED_ROUTINE_1 CustomDpcRoutine
```

Programový kód 7: obsah súboru SDV-map.h

Za zmienku stojí, že pri tomto príklade (vysvetlený v 5.2) je dostupný súbor *config.sdv* (obsah súboru je v Programový kód 8) , ktorý obsahuje pravidlá, ktoré sa použijú ak tento súbor zadáme pre SDV ako parameter. Konkrétne obsahuje 4 pravidlá, ktorým musí ovládač vyhovieť. Prvým je pravidlo `CancelSpinLock`, patrí k synchronizačným pravidlám a deklaruje, že ovládač

musí najskôr zavolať funkciu pre bezpečné synchronizovanie zrušiteľných stavov pre IRP pakety (tým „zamkne“ prenos medzi týmito stavmi) a až tak zavolať funkciu pre zrušenie zamknutia po zmene zrušiteľného stavu. Druhé pravidlo je `LowerDriverReturn`, patrí k IRP pravidlám a kontroluje prenos návratovej hodnoty z ovládača do funkcie typu dispatch s príslušným kontrolným kódom. Tretie pravidlo má názov `PagedCode` a patrí do skupiny IRQL pravidiel. Definuje podmienku, že ovládač volá makro `PAGED_CODE`[9] len vtedy ak IRQL <= APC_LEVEL[10]. Posledným, štvrtým pravidlom je `SpinLock`, ktoré patrí do skupiny synchronizačných pravidiel, určuje poradie volania funkcií pre zamknutie zdieľaného obsahu pre jeden proces pre jeho zmenu a následne jeho odomknutie.

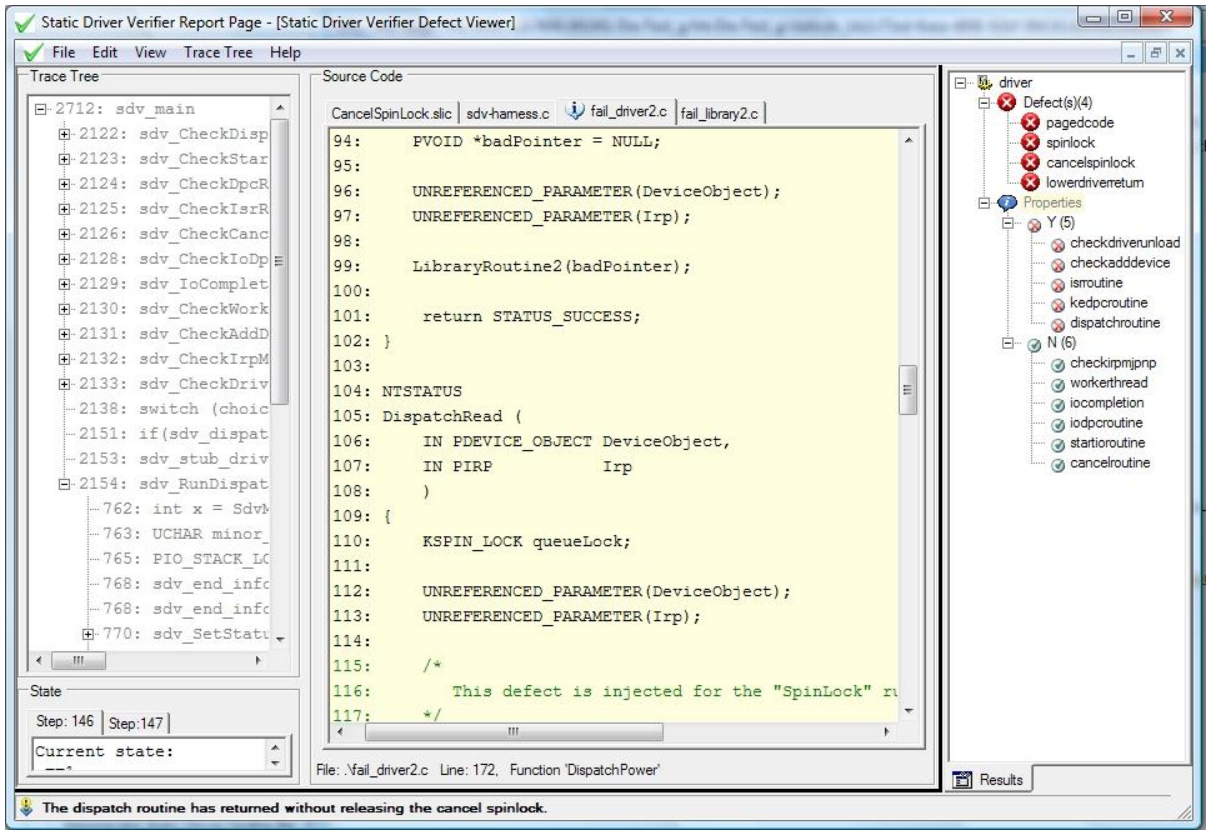
Teraz sú už zdrojové súbory pripravené na model checking. Spustíme ho príkazom `staticdv /config:Config.sdv`. Celý výpis SDV po vykonaní tohto príkazu je v Príloha č.3. Pri tomto procese sa znova skompilujú zdrojové súbory, nalinkujú knižnice a tu je vidieť, že sa pri linkovaní pridávajú i knižnice, ktoré neboli preložené pred verifikáciou. Ide o knižnice, ktoré obsahujú časti operačného systému, z ktorých sa vytvorí počiatočný model. Ďalej sa pri fáze PreCheck skontrolujú pravidlá vlastnosti súborov ktoré zisťujú, aké vlastnosti a možnosti má tento ovládač. Potom sa pravidlá načítavajú zo *.slic súborov. Hneď za touto fázou je vidieť, že je potrebné nalinkovať ešte dodatočne knižnice, pretože boli zistené také vlastnosti ovládača, ktoré pre svoju činnosť potrebujú pozmenené prostredie operačného systému ako je jeho súčasná načítaná abstrakcia.

```
CancelSpinLock
LowerDriverReturn
PagedCode
SpinLock
```

Programový kód 8: Obsah súboru pravidel config.sdv

Ďalšia časť výpisu pri vykonávaní model checkingu (Príloha č.3) má názov Check a sú v nej uvedené presne názvy pravidiel, ktoré sa overujú. Vo výpise nasleduje časť so zhodnotením verifikačného procesu. SDV skontroloval 11 vlastností, z toho 4 boli označené ako nepodporované týmto typom ovládača. Na poslednom riadku výpisu je príkaz na výpis správy (Report), ktorá sa zobrazí pomocou príkazu `staticdv /view`. Po zadaní príkazu sa otvorí Driver Verifier Report Page (Obrázok 6).

[9] Makro, ktoré zaručuje, že volané vlákno beží na úrovni, ktorá je dostatočne nízka na zakázanie stránkovania.
[10] Asynchronous procedure call level – určuje prioritu volania funkcie.



Obrázok 6: Správa vytvorená po úspešne ukončenom verifikovaní

Okno správy, ktoré sa otvorilo po zavolaní príkazu, je rozdelené na štyri časti. Na pravej strane je panel s výsledkami, kde v stromovej štruktúre je na vrchu meno ovládača, pod ním sú uvedené pravidlá, ktoré sa testovali. Pod nimi sú vlastnosti, ktoré sa kontrolovali vo fáze PreCheck. Na obrázku je vidieť, že ovládač neprešiel úspešne verifikáciou. Označujú to červené krížiky vo výsledkoch. Pre zobrazenie niektorého pravidla stačí podržať myš nad názvom. Pravidlo môžeme zobraziť v ostatných oknách v správe. Dvojklikom na určitú vlastnosť alebo pravidlo sa načíta zvolená hodnota načíta do ostatných troch častí okna.

Stredná časť okna zobrazuje zdrojový kód ovládača, ak je vybraté zobrazenie pravidla. Pri vybratej vlastnosti sa otvoria v okne dva panely. Na jednom z nich je kód časti operačného systému, v ktorom sa daná vlastnosť testovala a na druhom paneli je zobrazený kód pravidla v jazyku SLIC.

Na ľavej časti okna je komponenta s názvom Trace Tree (slov. sledovací strom). V tejto komponente je zobrazená cesta, ktorú kontroloval SDV a pri ktorej došiel k chybe.

Posledná časť okna má názov Stav (angl. State). Zobrazuje pravdivostné výrazy premenných zo zdrojového kódu, operačného systému a pravidla. Tieto hodnoty využíva pre vytvorenie abstrakcie ovládača, modelu operačného systému a pravidla. Túto hodnotu dosadzuje do pravidiel za premennú `s` a tak kontroluje platnosť pravidiel v ovládači. Tento panel zobrazuje tieto výrazy pre vybrané riadky buď v sledovacom strome alebo v zdrojovom kóde, ak sa dá zvýraznený riadok takto interpretovať. Týmto je verifikačný proces ukončený.

V ďalšej kapitole, pri verifikácii zvoleného ovládača už nebudú rozoberané jednotlivé časti takto podrobne, ale budú sa vyzdvihovať skôr oddiely, ktoré boli niečím zaujímavé.

7 Verifikácia ovládača Ext2Fsd

V tejto kapitole sa bude verifikovať vybraný ovládač Ext2Fsd. Ešte skôr ako bude verifikovaný, musí prejsť úpravami tak, aby ho SDV akceptoval. Ďalej musia byť zvolené pravidlá, ktorých platnosť bude pre zvolený ovládač overovaná. Až po skončení verifikácie budú výsledky odhalia, či bol Static driver verifier schopný nájsť vstupné body ovládača a ak áno tak, ktoré chyby odhalil.

7.1 Príprava ovládača

Najskôr bolo potrebné ovládač, resp. jeho zdrojové súbory upraviť. Podľa toho, v ktorých priečinkoch sa nachádzal súbor *Sources*[11], bolo jasné, že tieto priečinky budú musieť byť skompilované pomocou SDV. Po preskúmaní jednotlivých súborov *Sources* sa odhalili tri knižnice a jeden hlavný zdrojový adresár.

Príprava prebiehala spustením `staticdv /lib` v jednotlivých priečinkoch knižníc a potom v priečinku so zdrojovými súbormi. Najskôr bolo všetko v poriadku, avšak PREfast po automatickom preskúmaní, ktoré sa spustilo spolu s príkazom `staticdv /scan`, odhalil osem závažných chýb.
Išlo o chyby spôsobené pri písaní kódu, kde boli použité funkcie, ktoré sú v ovládačoch zakázané. Konkrétne išlo o funkcie pre prácu s reťazcami a ich kopírovaním – `swprintf`, `strcpy` a `strncpy`. Po dlhom hľadaní som dospel k riešeniu, že Microsoft vydal knižnicu, kde tieto funkcie nahradil bezpečnejšími. Boli to funkcie, ktoré mali rovnaké meno ako zakázané, ale na konci mali pridané `_s`, mali byť definované v súbore *strsafe.h*, ktorý som pomocou príkazu `include`, pridal do zdrojových súborov, kde som tieto funkcie použil. Avšak, ani potom nebolo možné ovládač preložiť. Pretože ich prekladač ich označoval ako nedeklarované. Znova po dlhšom pátraní som objavil hlavičkový súbor *ntstrsafe.h*, ktorý som dokonca našiel v knižniciach WDK. Po preskúmaní funkcií, ktoré sa nachádzali v tomto hlavičkovom súbore, som nakoniec našiel náhrady za zakázané funkcie. Boli to po poradí `RtlStringCbPrintf`, `RtlStringCbCopy` a `RtlStringCbCopyN`.

Každá táto funkcia má dva rôzne tvary, odlišujú sa príponou W alebo A k ich názvu. Prípona W znamená, že funkcia pracuje s dátovým typom wchar a prípona A znamená, že pracuje s normálnym typom char. Tieto funkcie mali dokonca i ďalší tvar, kde v názve miesto Cb mali písmená Cch. Označenie Cch znamená, že funkcie získavajú dĺžku reťazca počtom znakov. Cb znamená, že dĺžku reťazca vyjadrujú v bytoch.

Po nahradení zakázaných funkcií funkciami z hlavičkového súboru *ntstrsafe.h* už PREfast nehlásil žiadnu chybu. Ale po ďalšom spustení ohlásil množstvo varovaní, ktorých príčiny bolo nutné opraviť v zdrojových súboroch ovládača.

7.1.1 Oprava varovaní nájdených nástrojom PREfast

Na začiatku bolo varovaní 520. Pri opravovaní ich počet kolísal a niekedy dokonca neohlásilo žiadne varovanie. Možné vysvetlenie som nachádzal v náročnosti ovládača na PREfast, keďže každý jeho priebeh trval minimálne 5 minút, čo bolo oproti skompilovaniu zdrojových kódov podstatne dlhšia doba. Tým mohli nastať určité chyby pri spracovaní varovaní. Avšak po podrobnejšom preskúmaní varovaní, som zistil, že asi 180 varovaní je rovnakých. Bolo to varovanie

[11] Význam súborov *sources* vysvetlený v 6.9

č. 28107, ktoré upozorňovalo, že prostriedok, ktorý program musí získať pred volaním funkcie nie je a funkcia môže spadnúť. Ošetrením v tomto prípade bolo pred volaním funkcie oznámiť ovládaču, že ide o kritickú sekciu kódu, pri ktorej je nutné kontrolovať požadované zdroje. Po ukončení volanej funkcie oznámiť ovládaču, že kritická sekcia skončila. Tieto dva prípady, začatie a koniec kritickej sekcie sa označuje volaním funkcií `KeEnterCriticalRegion` a `KeLeaveCriticalRegion`. Týmto zásahom do kódu sa mi podarilo eliminovať všetkých uvádzaných 180 varovaní. Toto jediné varovanie sa mi podarilo odstrániť úplne.

Ďalším varovaním, ktoré sa mi podarilo čiastočne odstrániť bolo varovanie č. 28170, ktoré sa týkalo stránkového segmentu. V ňom bolo na odstránenie tohto varovania potrebné definovať makro `PAGE_CODE` alebo `PAGE_CODE_LOCKED` vo funkcii, ktorá sa volala v direktíve `#pragma`. Avšak po zadaní uvedeného makra do funkcií sa varovania neodstránili zo všetkých funkcií, podarilo sa ich odstrániť asi 20.

Posledné varovanie, ktorých bolo celkovo desať sa mi podarilo odstrániť osem. Bolo to varovanie č. 28103, ktoré označovalo, že použitý prostriedok nebol uvoľnený. Toto varovanie sa dalo odstrániť pomocou využitia „opačnej“ funkcie k funkcii, ktorá obsahovala slovo acquire. Opačnú funkcionalitu mala vždy funkcia kde namiesto uvedeného slova bolo slovo release.

Po týchto opravách kódu ostalo 255 varovaní. Tieto varovania, ktoré sa mi nepodarilo odstrániť som nakoniec po zvážení, že ide len o varovania, ignoroval a začal som pracovať na definovaní typov rolí (angl. role types), podľa ktorých si SDV určí vstupné body.

7.1.2 Nájdenie typov rolí

Po naštudovaní potrebných materiálov som dokázal určiť deväť typov rolí. Sú uvedené v Programový kód 9. Všetky z nich boli nájdené rovnakým spôsobom. Podľa tabuľky uvedenej v 6.6.1 som postupne prechádzal všetky zdrojové súbory a funkcie. Funkciám, ktoré využívali niektorú z funkcií uvedených v tabuľke som zapísal typ roly. Išlo väčšinou o funkcie `CompletionRoutine`, konkrétne ich bolo šesť. Ďalšou bola funkcia inicializácie ovládača `DriverEntry`, potom funkcia typu callback definovaná pomocou `WORKER_THREAD_ROUTINE` a poslednou bola funkcia, ktorá využívala rutinu `CustomDpc` a jej typ roly bol `KDEFERRED_ROUTINE`. Týmto boli nastavené vstupné body ovládača. Nasledujúcim krokom bolo vybratie pravidiel vytvorenie súboru *config.sdv*, ktorý ich bude obsahovať.

```
#define fun_IO_COMPLETION_ROUTINE_1 Ext2DeviceControlCompletion
#define fun_IO_COMPLETION_ROUTINE_2 Ext2FlushCompletionRoutine
#define fun_IO_COMPLETION_ROUTINE_3 Ext2MediaEjectControlCompletion
#define fun_IO_COMPLETION_ROUTINE_4 Ext2PnpCompletionRoutine
#define fun_IO_COMPLETION_ROUTINE_5 Ext2ReadWriteBlockAsyncCompletionRoutine
#define fun_IO_COMPLETION_ROUTINE_6 Ext2ReadWriteBlockSyncCompletionRoutine
#define fun_WORKER_THREAD_ROUTINE_1 Ext2FloppyFlush
#define fun_DriverEntry DriverEntry
#define fun_KDEFERRED_ROUTINE_1 Ext2FloppyFlushDpc
```

Programový kód 9: Obsah súboru SDV-map.h pre ovládač Ext2Fsd

7.2 Príprava súboru pravidiel

Pravidlá sú zo skupiny pre WDM ovládače, pretože ovládač využíva práve tieto typy funkcií. Táto skupina obsahuje sedem typov pravidiel (uvedené v 6.7). Skupinu, ktorá by mala mať najväčšie zastúpenie je skupina synchronizačných pravidiel, pretože súborový systém prakticky vždy pracuje so zdieľanými zdrojmi. Ako druhú najväčšiu skupinu som označil IRQL, lebo sa v nej využívajú prerušenia, ktoré sa využívajú pri prístupe na pevný disk. Nasledujúcou skupinou sú IRP pravidlá. Tie využívajú vstupne/výstupné požiadavky cez pakety, čo je tiež relatívne blízke k súborovému systému. Naopak ako nevhodnú skupinu som označil pravidlá pre PnP, pretože súborov systém sa nestará o správu napájania a pripojenia zariadenia. Nakoniec som však pre otestovanie zahrnul i jedno pravidlo z tejto skupiny. Po tejto úvahe som zostavil súbor pravidiel, v ktorom sú štyri IRP pravidlá, štyri IRQL pravidlá, deväť pravidiel zo synchronizačných, 2 pravidlá zo všeobecných a jedno pravidlo z PnP skupiny.

Zo synchronizačných pravidiel boli vybrané tieto pravidlá. `WithinCriticalRegion`, ktorý kontroluje, či operácie synchronizačného typu sa vykonávajú len v označenom kritickom regióne (označenie pomocou `KeEnterCriticalRegion` a `KeLeaveCriticalRegion`. Týmto pravidlom som chcel i overiť, či vykonané opravy zdrojových súborov voči varovaniam, ktoré sa týkali tohto pravidla boli úspešné. Ďalším pravidlom bolo `SpinLockRelease`, ktoré určuje, že zabratie zdroja je vždy nasledované jeho uvoľnením. Nasledujúcim pravidlom bolo `QueuedSpinLockRelease`. Je podobné pravidlo ako predchádzajúce, ale v tomto prípade sa zabratie a uvoľňovanie týka zaradených zariadení na zásobníku. `MarkingQueueIrps` pravidlo kontroluje, či sa volá funkcia, ktorá oznamuje očakávanie IRP paketu, len v prípade spin locku (vysvetlené v 6.9). Pravidlo `MarkingInterlockedQueuedIrps` špecifikuje, že IRP paket bude označený za očakávaný ešte pred zaradením do fronty paketov. Nasledujúce pravidlo je `ExclusiveResourceAccessRelease`, ktoré určuje, že funkcie pre uvoľnenie a zabratie určitého zdroja sa musia vždy striedať. Pravidlo `SpinLockSafe` kontroluje, aby volanie nasledujúceho paketu a volanie ukončenia požiadavky, nenastalo počas spin locku (vysvetlené v 6.9). Predposledným pravidlom z tejto skupiny je `CriticalRegions`, ktoré definuje, že funkcia `KeEnterCriticalRegion` musí byť vždy nasledovaná funkciou `KeLeaveCriticalRegion` a nikdy nesmú byť volané v opačnom poradí. Posledným pravidlom z tejto skupiny je pravidlo `CancelSpinLock`, ktoré bolo vysvetlené v 6.9.

Pravidlá zo skupiny IRP sú nasledovné. Prvým je `WmiComplete`, ktorý kontroluje, či pri spracovávaní WMI IRP (vysvetlené v 6.7) paketu je najskôr toto spracovávanie ukončené a až potom sa predá riadenie programu. Druhé pravidlo `WmiForward` kontroluje preposielanie paketov WMI ak je preposielanie nastavené. Pomocou tretieho pravidla `StartIoCancel` sa zdrojový kód kontroluje, či sa nevolá funkcia, ktorá nastavuje I/O prenos, s nezrušiteľným parametrom. Posledným pravidlom v tejto skupine je `LowerDriverReturn`, ktoré bolo vysvetlené v 6.9.

Zo skupiny IRQL pravidiel boli vybraté štyri pravidlá. Pravidlo `IrqlKeApcLte2` určuje že vybrané funkcie z jadra sa môžu volať len v prípade IRQ level je menší alebo rovnaký ako APC_LEVEL(vysvetlené v 10). Nasledujúce pravidlo `IrqlReturn` špecifikuje, že IRQ level návratovej hodnoty rutiny typu dispatch je rovnaký ako level, pri ktorom bola volaná. Ďalšie pravidlo `IrqlKeSetEvent` špecifikuje volanie nastavenia udalosti len v prípade keď IRQ level je menší alebo rovnaký ako level dispatch. Posledné pravidlo `IrqlExApcLte3` kontroluje, či vybrané podporné rutiny sú volané len v prípade keď IRQ level je menší alebo rovný ako APC_LEVEL.

Ostali posledné dve pravidlá. Prvé z nich je zo skupiny všeobecných pravidiel a má názov `SafeStrings`. Špecifikuje, že ovládač používa len také funkcie, ktoré sú ochránené voči manipulácii systému. Posledné pravidlo v zozname pravidiel je `AddDevice`, ktoré bolo rozobrané v kapitole 6.7.

Týmto sú definované pravidlá, ktoré budú použité pri kontrole ovládača Ext2Fsd pomocou SDV. Pri zhodnotení výsledku budú podrobnejšie vysvetlené princípy pravidiel, pre ktoré ovládač nevyhovel. Použitý súbor pravidiel *config.sdv* je znázornený v Príloha č.2.

7.3 Priebeh verifikácie

Verifikácia ovládača so súborom, ktorý obsahoval dvadsať pravidiel trvala približne 14 minút[12]. Z pozorovania priebehu model checkingu sa zistili nasledovné časy trvania jednotlivých častí priebehu, ktoré sú zobrazené v Tabuľka 2. Súbor s pravidlami sa načítal od posledného zadaného pravidla po prvé, niektoré pravidlá sa testovali dlhšie a tým vznikalo prekrývanie pravidiel a niektoré sa vykonali zase extrémne rýchlo. Po analýze priebehu bolo zistené, že je to spôsobené testovaním na dvojjadrovom procesore, ktorý SDV využíval naplno, čiže model checking bežal na oboch jadrách. Záťaž procesor vo fázach model checkingu bola takmer vždy 100 %, pri ostatných častiach bola okolo 50 %. SDV pri fázach PreCheck a Check spúšťal proces *slam.exe*, ktorý využíval priemerne 150 MB operačnej pamäti pre jedno jadro. Najvyššia hodnota 371 MB zabranej operačnej pamäti bola pri kontrole pravidla `MarkingInterlockedQueuedIrps`.

Fáza priebehu SDV	**Trvanie (hh:mm:ss)**
Build	0:00:20
Scan	0:00:57
Kompilovanie a linkovanie knižníc I.	0:01:01
PreCheck	0:04:12
Kompilovanie a linkovanie knižníc II.	0:01:39
Check	0:06:18
Celkovo:	0:14:27

Tabuľka 2: Doba trvania jednotlivých fáz SDV

7.4 Overenie výsledkov

Prvým náznakom úspechu verifikácie je výpis po ukončení verifikačného procesu. Úspech verifikácie v tomto prípade znamená, že SDV bol schopný rozoznať vstupné body a overiť niektoré pravidlá. Neúspechom by bolo, keby sa nepodarilo aplikovať ani jedno pravidlo. Vtedy by to vypovedalo o zlom definovaní vstupných bodov. Pri tomto ovládači bol výpis celkom uspokojivý. Je možné ho vidieť okne programového kódu 10. Z týchto riadkov vieme vyčítať, že Static Driver Verifier skontroloval 12 vlastností programu – fáza PreCheck.

```
SDV checked 12 properties(s).
SDV performed 20 check(s)
with:
        1  Defect(s)
        15 Rule Passes
        4 Not Applicable
```

Programový kód 10: konečný výpis SDV po verifikácii Ext2Fsd

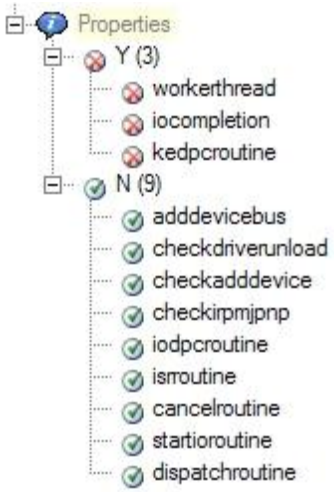
[12] Testované na stroji dvojjadrové 2,2 GHz CPU 32-bit, 4 GB RAM, použité MB pamäti.

Ďalej je uvedené, že nástroj vykonal 20 kontrol, čiže boli skontrolované všetky zadané pravidlá zo súboru. Na nasledujúcich riadkoch uvádza úspešnosť verifikácie. Jedno pravidlo spôsobilo zaseknutie sa a ukončilo sa neúspešne, pätnásť pravidiel prešlo verifikačným procesom. Zaseknutie pravidla znamená, že pri jeho kontrole došlo k jeho porušeniu a proces verifikácie sa „zasekol“ na bode, ktorý spôsobuje poruchu. Tým sa toto pravidlo označí ako porušené (podľa Programový kód 10 – angl. Defect). Zostávajúce 4 sa nedali na tento typ aplikácie použiť, pravdepodobne aplikácia nevyužíva funkcie, ktorých činnosť tieto pravidlá kontrolujú alebo vstupné body ovládača pre tieto pravidlá neboli vytvorené.

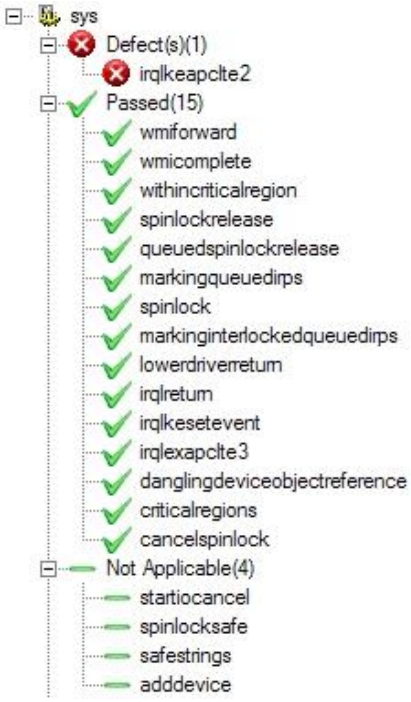
Oveľa komplexnejší prehľad o výsledkoch verifikácie dáva Static Driver Verifier Report. Jednotlivé okná aplikácie boli vysvetlené už skôr. Preto bude pozornosť zameraná na typy pravidiel, cez ktoré ovládač úspešne prešiel, ktoré nedokázali byť interpretované pre jeho zdrojové súbory a nakoniec, pre ktoré pravidlo ovládač vo verifikačnom procese zlyhal. Na nasledujúcich obrázkoch (Obrázok 7 a Obrázok 8) je vidieť detailnejší pohľad na výsledky.

Potvrdili sa predpoklady, že synchronizačné pravidlá budú vhodné pre tento ovládač, pretože až na prípad pravidla `SpinLockSafe`, na všetky ostatné dokázal reagovať. V neaplikovateľných skončilo, rovnako predpokladane, pravidlo pre PnP, ktoré tu nemalo prakticky žiadnu podporu v ovládači. Jediné pravidlo, ktoré dokázalo zlomiť korektnosť ovládača bolo pravidlo typu IRQL, konkrétne `IrqKeApcLte2`, ktoré odhalilo chybu ovládača. Vlastnosťou tohto pravidla je špecifikovať stav, kedy ovládač volá nasledujúcu rutinu jadra v len v prípade kedy IRQ Level <= APC_Level. IRQ level je hodnota, ktorá určuje prioritu prerušenia, akú má dané zariadenie, ktoré ovládač obsluhuje. APC level je rovnako hodnota určujúca prioritu, avšak prioritu volania funkcie asynchrónne – mimo nastavené synchronizované impulzy. Čiže toto pravidlo kontroluje, aby sa rutiny v pravidle uvedené volali len ak priorita prerušenia je menšia alebo rovnaká ako priorita asynchrónneho volania funkcie. Vysvetlenie ďalších použitých pravidiel je v dokumentácii Windows Driver Kit.

Nasleduje popis výsledkov z fázy PreCheck, ktorá pri každom model checkingu SDV kontroluje vlastnosti ovládača. Táto fáza nemá žiadne spojenie s činnosťou nástroja PREfast. Takáto funkcia pomáha vo vývoji ovládača, pretože vieme, ktoré vlastnosti ovládač už má a ktoré ešte nie. Vlastnosti kontrolované fázou PreCheck sú rozdelené do dvoch skupín. Prvá skupina s červeným krížom sú vlastnosti, ktoré daný ovládač podporuje. Preto sa dajú vybrať a zobraziť ich zdrojové kódy. Naopak však vlastnosti so zeleným označením sú nepodporované týmto ovládačom. Na záver testovania ovládača Ext2Fsd budú uvedené jeho metriky kódu.



Obrázok 7: Kontrolované vlastnosti



Obrázok 8: Výsledok verifikácie 20 pravidiel

7.5 Metriky kódu ovládača Ext2Fsd

Ovládač obsahuje celkovo 92 súborov (počítajú sa zdrojové a hlavičkové súbory). V nich je dokopy s komentármi 99 549 riadkov, mimo komentárov to je 21 531. Ovládač obsahuje 652 funkcií. Najviac riadkov má súbor *nls_cp949.c*, konkrétne 13 947, ale väčšina jeho riadkov (90,1 %) je tvorená komentármi. Ako funkčne najviac potrebný súbor, ktorý má najviac funkčných riadkov je *ntfis.gnu.h*, obsahuje 2 945 riadkov. Súbor s najväčším počtom funkcií je *memory.c*, ktorý obsahuje 56 funkcií. Najobsiahlejšou je funkcia `Ext2NtStatusToString`, ktorá obsahuje 921 riadkov. Najčastejšie volanou funkciou je funkcia `Ext2CreateFile`, volá sa konkrétne 189 krát. Maximálne zanorenie celkovo v súboroch ovládača je 9, najčastejšie je zanorenie prvého stupňa. Je zaujímavé, že komentáre tvoria až 51,3 % zo všetkých riadkov zdrojových súborov. Priemerný počet príkazov na jednu funkciu je 37. Štatistiky boli vygenerované pomocou programu SourceMonitor[13].

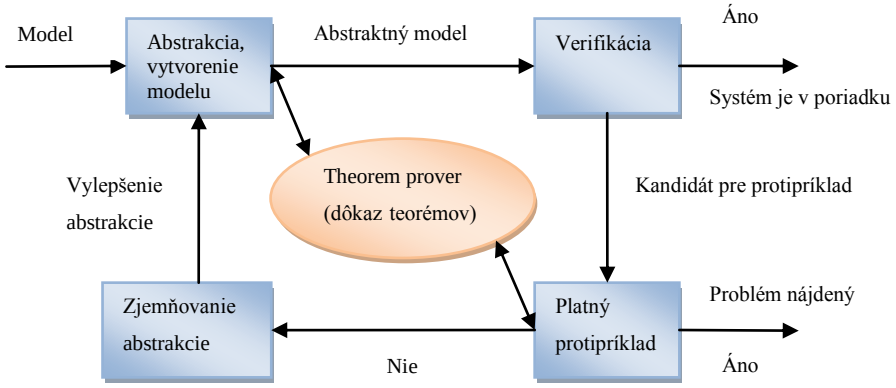
[13] www.campwoodsw.com

8 Copper

Keďže základ o tomto nástroji bol už vysvetlený, v tejto kapitole sa bude tento verifikačný nástroj rozoberať podrobnejšie. Bude tu uvedená charakteristika tohto nástroja. Nasledujúce informácie sú voľne prebrané z [6].

8.1 Charakteristika nástroja

Copper je model checker, ktorý využíva CEGAR, čo je prístup (angl. framework), ktorý využíva princíp protikladom riadené vylepšenie abstrakcie. Tento prístup sa hlavne využíva na verifikovanie softwaru. V CEGARe (Obrázok 9) sú modely cieľového programu automaticky vytvorené pomocou predikátovej abstrakcie a iteratívne sú zjemňované na základe falošného protikladu. Proces pokračuje dovtedy, pokým je dokázaná žiadaná vlastnosť alebo je nájdený reálny protiklad.



Obrázok 9: princíp fungovania CEGARu, obrázek prevzatý z [6].

Z programovacieho hľadiska je Copper veľmi flexibilný. Dokáže riešiť jednoduché úlohy ako napríklad porušenie assertu (makro, ktoré sa využíva na testovacie účely) alebo na druhej strane pracovať so špecifikáciami overovaných vlastností v podobe konečných automatov alebo formúl lineárnej temporálnej logiky. Dokáže dokonca verifikovať viacvláknové aplikácie, kde vlákna komunikujú pomocou zdieľaných premenných alebo na princípe handshake.

Tento nástroj ma i niektoré nevýhody, napríklad nepodporuje desatinné čísla. Všetky typy premenných, ktoré nepozná označí typom integer, rovnako má i problémy s ukazateľmi, nepodporuje rekurzívne programy.

Copper tvorí jadro systému zvaného Comfort, čo je dedukčné prostredie, vytvára predpoklady o bezpečnosti a spoľahlivosti systému. Keďže využíva Copper, jeho základom je model checking.

Princíp dedukčného prostredia, podľa [16], je v zabalení určitého nástroja pre potrebu overenia kvality nejakého spúšťaného systému. Kľúčovou vlastnosťou tohto prostredia je poskytnúť programátorom takú konštrukciu, ktorá už obsahuje všetky potrebné komponenty na určenie kvalít systému. Pri tom prostredie poskytuje jednoznačné výpisy, ktoré je možné analyzovať a tak zlepšiť vlastnosti kontrolovaného systému.

8.2 Príklad verifikácie

Jednoduchý príklad používa syntax Copperu. Je to príklad prevzatý z dokumentácie k tomuto nástroju. Príklad sa týka obedovania filozofov, ktorý sedia pri okrúhlom stole a každý z nich má vedľa seba vidličku na ľavej i pravej strane. Príklad rieši obedovanie dvoch filozofov, ktorí majú práve dve vidličky.

Copper tento problém podľa Obrázok 10 rieši definovaním vlastných procedúr pre dvoch filozofov a pre dve vidličky. Deklarácia filozofa je jednoduchá. Ak je aktívny, tak najskôr skúsi vybrať vidličku, po jeho ľavej strane. Príkazom `COPPER_HANDSHAKE` sa jeho výber inicializuje tak, že sa oznámi, že vidlička je obsadená. Po vykonaní rovnakého príkazu i s vidličkou na pravej strane, začína filozof jedenie. V tomto prípade je pri jedení príkaz `assert(0),` ktorý oznamuje uplatnenie jedenia. Po skončení jedenia, filozof odloží ľavú vidličku a potom pravú vidličku rovnakým príkazom, avšak s argumentom „put ...“. Druhý filozof vykonáva to isté, avšak má definované vlastné argumenty, ktoré používa (argumenty v tomto prípade u filozofov môžu predstavovať ich ruky). Vidličky majú v tele definovanú len jednu procedúru. Pomocou nich sa v špecifikácii určí, čo konkrétne sa robí pri vyberaní vidličky.

```
void phil1()
{
   int eating;
   eating = 0;
   while(1) {

__COPPER_HANDSHAKE__("pick_left_1");

__COPPER_HANDSHAKE__("pick_right_1");
      eating = 1;
      if(eating != 1) assert(0);
      eating = 0;

__COPPER_HANDSHAKE__("put_left_1");

__COPPER_HANDSHAKE__("put_right_1");
   }
}
```

```
void phil2()
{
   int eating;
   eating = 0;
   while(1) {

__COPPER_HANDSHAKE__("pick_left_2");

__COPPER_HANDSHAKE__("pick_right_2");
      eating = 1;
      if(eating != 1) assert(0);
      eating = 0;

__COPPER_HANDSHAKE__("put_left_2");

__COPPER_HANDSHAKE__("put_right_2");
   }
}
```

```
/* fork 1 */
void
fork1()
{

do_fork1();
}


/* fork 2 */
void
fork2()
{

do_fork2();
}
```

Obrázok 10: Dvaja filozofi a jedna vidlička

Špecifikácia má tvar uvedený v Programový kód 11. Sú definované dve procedúry, ktoré majú nastavené správanie pri vyberaní vidličiek. Napríklad procedúra `do_fork1` má definované správanie ako výber a položenie pravej vidličky alebo výber a položenie ľavej vidličky. Správanie procedúry `do_fork2` je podobné.

```
DoFork1 = ( pick_right_1 -> put_right_1 -> DoFork1 | pick_left_2 -> put_left_2 -> DoFork1 ).
procedure do_fork1 { abstract { 1 , DoFork1 }; }
DoFork2 = ( pick_right_2 -> put_right_2 -> DoFork2 | pick_left_1 -> put_left_1 -> DoFork2 ).
procedure do_fork2 { abstract { 1 , DoFork2 }; }
```

Programový kód 11: Špecifikácia pravidel brania vidličiek

Ešte je však nutné nastaviť obmedzenie, ktoré definuje, že obaja filozofi nemôžu jesť naraz. Vykoná sa to príkazom pre Copper, ktorým sa zadáva temporálna logika. Je uvedený v Programový kód 12.

```
ltl DpSpec1 { #G [ (P0::eating == 0) || (P1::eating == 0) ]; }
```

Programový kód 12: Výraz označujúci podmienku, že filozofi nemôžu jesť obaja naraz

Nakoniec sa musí špecifikácia prepojiť s procedúrami filozofov a vidličiek. Toto spojenie sa vytvorí ako je uvedené v Programový kód 13.

```
program phil1,phil2,fork1,fork2 {
  specification abs_1,{P0::eating == 0,P1::eating == 0,1,1},DpSpec1;
}
```

Programový kód 13: Prepojenie procedúr so špecifkáciou.

Tým sú pripravené všetky súbory (Obrázok 10 v phil2.pp a Programový kód 11, Programový kód 12 v phil2.spec). Copper sa spustí príkazom `copper --default --specification abs_1 dp-2.pp dp-2.spec --ltl`, kde sú dôležité parametre `--|t|` a `--specification`. Prvý menovaný označuje, že sa má použiť SE-LTL[14] model checking a druhý označuje použitú špecifikáciu. Za ňou nasledujú mená súborov s procedúrami pre filozofov a vidličky a so špecifikáciou. Tým sa nástroj spustí a Copper vypíše, že uspel pri vytváraní tohto modelu a nenašiel chybu.

Toto bol v krátkosti zhrnutý nástroj Copper, ktorý využíva model checking. Tento príklad bol voľne prevzatý z [17].

[14] State/event – linear temporal logic: stavová, udalostná lineárna temporálna logika

9 Záver

Táto práca sa zaoberala verifikáciou, konkrétne model checkingom, ktorý vykonávali programy automaticky s miernymi zásahmi človeka. Od začiatku bola práca usmerňovaná tak, aby po jej postupnom čítaní, čitateľ zisťoval spojitosti medzi kapitolami a nakoniec dospel ku kapitole 7, kde sa vykonávala najdôležitejšia časť práce. Bola to verifikácia ovládača, ktorý bol zvolený ako dobrý testovací subjekt. Výsledok bol do poslednej chvíle neistý, pretože pri tak veľkom počte zdrojových kódov sa nedá obsiahnuť celá komplexnosť programu naraz. Ako ovládač prechádzal postupným procesom verifikovania, boli do neho robené určité zásahy, ktoré boli nevyhnutné. Nakoniec sa všetky chyby podarilo odstrániť a ovládač bol pripravený na samotný model checking. Na tejto časti práce bolo zaujímavé práve to, že mohla dopadnúť rôznymi spôsobmi. Napríklad by nebolo možné aplikovať ani jedno pravidlo na ovládač. Iná možnosť by bola, že ovládač prešiel všetkými testami bez poruchy. Avšak nastala tá najlepšia možná situácia a to tá, že sa podarilo dosiahnuť všetky možné stavy, ako môže model checking pomocou SDV dopadnúť. Podarilo sa dosiahnuť stav, keď sa stali všetky tri možnosti (bolo pravidlo, ktorého testom ovládač neprešiel, rovnako sa objavili pravidlá, s ktorými si ovládač poradil a nakoniec ostali pravidlá neaplikovateľné na tento typ ovládača).

V závere je dobré položiť si otázku, akým smerom by sa mal uberať výsledok tejto práce. Je možné, že sa aplikujú i všetky tieto smery do budúcnosti uvedené v tomto odstavci. Prvý by mohol ísť smerom k vývojárom ovládača, ktorý sa testoval. Oznámiť im, že automatická verifikácia odhalila chybu v ich ovládači a pokúsiť sa navrhnúť im riešenie. Ďalší smer si môže vybrať nový ovládač, najlepšie podobný tomuto ovládaču, ktorý bolo vhodné verifikovať a potom porovnať ich bezpečnosť vzhľadom na pravidlá, ktoré boli pri ich verifikovaní použité. Posledný smer je z môjho pohľadu najzložitejší, ale i perspektívny. Skúsiť napísať vlastný ovládač, ktorý by vyhovel všetkým verifikačným pravidlám.

Literatúra

1 SANDEEP, Kumar Shukla. *U.S. National Institute of Standards and Technology : Dictionary of Algorithms and Data Structures* [online]. 17 December 2004 [cit. 2010-05-15]. Formal verification.
Dostupné z WWW: <http://www.itl.nist.gov/div897/sqg/dads/HTML/formalverf.html>.

2 *Wikipedia, the free encyclopedia* [online]. 5 April 2010 [cit. 2010-05-16]. Verification. Dostupné z WWW: <http://en.wikipedia.org/wiki/Verification>.

3 VOJNAR, Tomáš. *Cut-offs and Automata in Formal Verification of Infinite-State Systems.* Brno, 2007. 189 s. Habilitační práce. Vysoké Učení Technické v Brně, Fakulta Informačních Technologií.

4 CLARKE, Edmund M. Jr.; GRUMBERG, Orna; PELED, Doron A. *Model Checking*. Cambridge : The MIT Press, 1999. 314 s. ISBN 0-262-03270-8, 978-0-262-03270-4 .

5 BEYER, Dirk, et al. The software model checker BLAST : Applications to software engineering. *International Journal on Software Tools for Technology Transfer* [online]. 13 September 2007, 9, [cit. 2010-05-16]. Dostupný z WWW: <http://www.sosy-lab.org/~dbeyer/Publications/2007-STTT.The_Software_Model_Checker_BLAST.pdf>. ISSN 1433-2787.

6 *Predictability by Construction* [online]. 2008 [cit. 2010-05-16]. Copper Model Checker. Dostupné z WWW: <http://www.sei.cmu.edu/predictability/tools/copper/>.

7 *MSDN Library* [online]. 9-4-2010 [cit. 2010-05-16]. Introduction to File System Filter Drivers. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff548202%28v=VS.85%29.aspx>.

8 *MSDN Library* [online]. 9-4-2010 [cit. 2010-05-16]. Filter Manager Concepts. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff541610.aspx>.

9 *MSDN Library* [online]. 9-4-2010 [cit. 2010-05-16]. How File System Filter Drivers Are Similar to Device Drivers. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff548084%28VS.85%29.aspx>.

10 *MSDN Library* [online]. 9-4-2010 [cit. 2010-05-16]. How File System Filter Drivers Are Different to Device Drivers. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff548075%28VS.85%29.aspx>.

11 *MSDN Library* [online]. 10-5-2010 [cit. 2010-05-16]. Static Driver Verifier Concepts. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff552813%28VS.85%29.aspx>.

12 *MSDN Library* [online]. 19-3-2010 [cit. 2010-05-16]. Choosing a Driver Model. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff554652%28VS.85%29.aspx>.

13 *MSDN Library* [online]. 10-5-2010 [cit. 2010-05-16]. Static Driver Verifier General Tool and Technical Limitations. Dostupné z WWW:<http://msdn.microsoft.com/en-us/library/ff552819%28VS.85%29.aspx>.

14 *MSDN Library* [online]. 10-5-2010 [cit. 2010-05-16]. Supported Drivers. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff552864%28VS.85%29.aspx>.

15 *MSDN Library* [online]. 10-5-2010 [cit. 2010-05-16]. Static Driver Verifier Commands. Dostupné z WWW:<http://msdn.microsoft.com/en-us/library/ff552811%28v=VS.85%29.aspx>.

16 *Predictability by Construction* [online]. 2008 [cit. 2010-05-17]. Reasoning Frameworks. Dostupné z WWW: <http://www.sei.cmu.edu/predictability/start/reasoning.cfm>.
17 *Copper Manual* [online]. Pittsburgh: Software Engineering Institute, 2008 [cit. 2010-05-17]. Dostupné z WWW:<http://www.sei.cmu.edu/library/assets/copper.pdf>.

Prílohy

Príloha č.1

Zoznam súborov testovaného ovládača. Každá zarážka predstavuje zložku. Budú uvedené len hlavičkové a zdrojové súbory, súbory sources a Makefile.

- ext3: *generic.c, recover.c, Sources, Makefile*.
- include: *common.h, ext2fs.h, ntifs.gnu.h, resource.h*
    - asm: *page.h, semaphore.h, uaccess.h.*
    - linux: *bitops.h, bit_spinlock.h, buffer_head.h, config.h, debugfs.h, errno.h, Ext2_fs.h, ext3_fs.h, ext3_fs_i.h, ext3_fs_sb.h, ext3_jbd.h, freezer.h, fs.h, highmem.h, init.h, jbd.h, journal-head.h, kernel.h, kthread.h, list.h, lockdep.h, log2.h, magic.h, mm.h, module.h, mutex.h, nls.h, pagemap.h, poison.h, proc_fs.h, sched.h, slab.h, spinlock.h, stddef.h, string.h, time.h, timer.h, types.h, version.h.*
- jbd: *recovery.c, replay.c, revoke.c, Sources, Makefile*.
- nls: *nls_ascii.c, nls_base.c, nls_cp1250.c, nls_cp1251.c, nls_cp1255.c, nls_cp437.c, nls_cp737.c, nls_cp775.c, nls_cp850.c, nls_cp852.c, nls_cp855.c, nls_cp857.c, nls_cp860.c, nls_cp861.c, nls_cp862.c, nls_cp863.c, nls_cp864.c, nls_cp865.c, nls_cp866.c, nls_cp869.c, nls_cp874.c, nls_cp932.c, nls_cp936.c, nls_cp949.c, nls_cp950.c, nls_euc-jp.c, nls_iso8859-1.c, nls_iso8859-13.c, nls_iso8859-14.c, nls_iso8859-15.c, nls_iso8859-2.c, nls_iso8859-3.c, nls_iso8859-4.c, nls_iso8859-5.c, nls_iso8859-6.c, nls_iso8859-7.c, nls_iso8859-9.c, nls_koi8-r.c, nls_koi8-ru.c, nls_koi8-u.c, nls_utf8.c, Sources, Makefile*.
- sys: *Makefile, Sources*.
- *resource.h, block.c, cleanup.c, close.c, cmcb.c, create.c, debug.c, devctl.c, dirctl.c, dispatch.c, except.c, fastio.c, fileinfo.c, flush.c, fsctl.c, init.c, linux.c, lock.c, memory.c, misc.c, nls.c, pnp.c, read.c, shutdown.c, volinfo.c, write.c*.

Príloha č.2

Obsah súboru s pravidlami pre verifikáciu ovládača Ext2Fsd – *config.sdv*

```
AddDevice
SafeStrings
SpinLockSafe
StartIoCancel
CancelSpinLock
CriticalRegions
DanglingDeviceObjectReference
IrqlKeApcLte2
IrqlExApcLte3
IrqlKeSetEvent
IrqlReturn
LowerDriverReturn
MarkingInterlockedQueuedIrps
MarkingQueuedIrps
SpinLock
QueuedSpinLockRelease
SpinLockRelease
WithinCriticalRegion
WmiForward
WmiComplete
```

Príloha č.3

Výpis SDV pri verifikácii ovládača z príkladov vo WDK.

```
---------------------------------------------------------------------
Microsoft (R) Windows (R) Static Driver Verifier Version 2.0.372.0
Copyright (C) Microsoft Corporation.  All rights reserved.
---------------------------------------------------------------------
Build     'driver'  ...Done
Link      'driver' for [fail_library2.lib] ...Done
Scan      'driver'  ...Done
Compile   'driver' for [sdv_flat_dispatch_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_dispatch_startio_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_simple_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
Compile   'driver' for [sdv_flat_harness] ...Done
Link      'driver' for [fail_library2.lib] ...Done
PreCheck  'driver' for 'checkdriverunload' ...Running
PreCheck  'driver' for 'checkirpmjpnp' ...Running
PreCheck  'driver' for 'checkdriverunload'  ...Done
PreCheck  'driver' for 'checkirpmjpnp'  ...Done
PreCheck  'driver' for 'workerthread' ...Running
PreCheck  'driver' for 'checkadddevice' ...Running
PreCheck  'driver' for 'checkadddevice'  ...Done
PreCheck  'driver' for 'iocompletion' ...Running
PreCheck  'driver' for 'workerthread'  ...Done
PreCheck  'driver' for 'iodpcroutine' ...Running
PreCheck  'driver' for 'iocompletion'  ...Done
PreCheck  'driver' for 'kedpcroutine' ...Running
PreCheck  'driver' for 'iodpcroutine'  ...Done
PreCheck  'driver' for 'isrroutine' ...Running
PreCheck  'driver' for 'isrroutine'  ...Done
PreCheck  'driver' for 'cancelroutine' ...Running
PreCheck  'driver' for 'kedpcroutine'  ...Done
PreCheck  'driver' for 'startioroutine' ...Running
```

```
    PreCheck  'driver' for 'startioroutine'  ...Done
    PreCheck  'driver' for 'dispatchroutine' ...Running
    PreCheck  'driver' for 'cancelroutine'  ...Done
    PreCheck  'driver' for 'dispatchroutine'  ...Done
    Compile   'driver' for [sdv_flat_harness] ...Done
    Link      'driver' for [fail_library2.lib] ...Done
    Compile   'driver' for [sdv_flat_simple_harness] ...Done
    Link      'driver' for [fail_library2.lib] ...Done
    Compile   'driver' for [sdv_flat_harness] ...Done
    Link      'driver' for [fail_library2.lib] ...Done
    Compile   'driver' for [sdv_flat_harness] ...Done
    Link      'driver' for [fail_library2.lib] ...Done
    Check     'driver' for 'spinlock' ...Running
    Check     'driver' for 'pagedcode' ...Running
    Check     'driver' for 'pagedcode'  ...Done
    Check     'driver' for 'lowerdriverreturn' ...Running
    Check     'driver' for 'spinlock'  ...Done
    Check     'driver' for 'cancelspinlock' ...Running
    Check     'driver' for 'cancelspinlock'  ...Done
    Check     'driver' for 'lowerdriverreturn'  ...Done
    SDV checked 11 properties(s).
    SDV performed 4 check(s) with:
            4  Defect(s)
Run this command to view the results: StaticDV /view
```

Zoznam príloh

Príloha 1: obsah adresára Ext2Fsd (vypísané len súbory *.c, *.h, Sources a Makefile).
Príloha 2: obsah súboru *config.sdv* obsahujúci pravidlá pre SDV.
Príloha 3: výpis SDV po vykonaní model checkingu pre uvedený príklad z WDK.
Príloha 4: DVD s inštalačnými súbormi pre WDK a Copper, adresárom so súbormi Ext2Fsd a upraviteľnou kópiou bakalárskej práce.