Podpora frameworku Valgrind v prostředí IDE CodeLite

Support for Valgrind Framework in CodeLite IDE

Bc. Pavel Srb

Diplomová práce
2014

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2013/2014

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Pavel Srb**
Osobní číslo:	**A11530**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**

Téma práce:	**Podpora frameworku Valgrind v prostředí IDE CodeLite**

Zásady pro vypracování:

1. **Vytvořte rešerši na téma profilace aplikací vyvíjených pomocí jazyka C/C++ s důrazem na kontrolu korektního využití a správy operační paměti.**
2. **Vytvořte rozšiřující modul vývojového prostředí IDE CodeLite integrující nástroj memcheck profilačního frameworku Valgrind.**
3. **Výsledky profilace poskytované nástrojem memcheck zobrazujte standardní formou pomocí prostředků dostupných v IDE CodeLite.**
4. **Vytvořte programovou a uživatelskou dokumentaci.**
5. **Zásuvný modul publikujte pod licencí GPLv2 nebo novější.**

Rozsah diplomové práce:
Rozsah příloh:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **SEWARD, Julian, Nicholas NETHERCOTE a Josef WEIDENDORFER. Valgrind 3.3: Advanced Debugging and Profiling for GNU/Linux applications. Bristol: Network theory Limited, 2008. ISBN 0-9546120-5-1.**

2. **PRATA, Stephen. Mistrovství v C++. 3. aktualiz. vyd. Překlad Boris Sokol. Brno: Computer Press, 2007, 1119 s. ISBN 978-80-251-1749-1.**

3. **CORMEN, Thomas H. Introduction to algorithms. 3rd ed. Cambridge: MIT Press, 2009, xix, 1292 s. ISBN 978-0-262-03384-8.**

4. **KANISOVÁ, Hana a Miroslav MÜLLER. UML srozumitelně. 2. aktualiz. vyd. Brno: Computer Press, 2006, 176 s. ISBN 80-251-1083-4.**

5. **SMART, Julian. Cross-platform GUI programming with wxWidgets. Upper Saddle River: Prentice-Hall, 2006, xxxv, 700 s. ISBN 01-314-7381-6.**

6. **Valgrind Documentation. VALGRIND DEVELOPERS. Valgrind [online]. 3.9.0. 31.10.2013 [cit. 2014-01-23]. Dostupné z: http://valgrind.org/docs/manual/index.html**

7. **Reference: Standard C++ Library reference. Cplusplus.com: The C++ Resources Network [online]. 3.1. 2000-2014 [cit. 2014-01-23]. Dostupné z: http://www.cplusplus.com/reference/**

8. **SMART, Julian et al. Documentation. WxWidgets: Cross-platform GUI Toolkit [online]. 3.0.0. 11.11.2013 [cit. 2014-01-23]. Dostupné z: http://docs.wxwidgets.org/stable/index.html**

Vedoucí diplomové práce: **Ing. Michal Bližňák, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **21. února 2014**
Termín odevzdání diplomové práce: **20. května 2014**

Ve Zlíně dne 21. února 2014

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Tato práce se zabývá nástroji na profilování paměti a zaměřuje se na problematiku hledání úniků paměti v programech vyvíjených v C/C++. Realizačním výstupem práce je rozšíření (plugin) do vývojového prostředí CodeLite o podporu nástroje Memcheck, který je součástí frameworku Valgrind.

Teoretická část práce vysvětluje rozdíl mezi statickou a dynamickou analýzou programu, popisuje způsoby profilování programů, včetně popisu technické realizace profilerů. Práce se dále zaměřuje na profilery paměti a provádí srovnání několika nejvýznamnějších. Součástí první části je také teoretická diskuze o návrhu a tvorbě GUI a také porovnání existujících grafických front-endů pro Valgrind. V teoretické části je také popis IDE CodeLite jak z hlediska uživatelského, tak hlavně možností jeho rozšíření, včetně popisu knihovny wxWidgets, na které je CodeLite vystavěn.

Praktická část se věnuje samotnému návrhu a implementaci pluginu. Jsou zde popsány technické problémy při jeho tvorbě a diskutována možná řešení. Další součástí praktické části je popis rozhraní pro ještě další rozšíření pluginu. V poslední kapitole je uživatelská dokumentace.

Klíčová slova: CodeLite, Plugin, Valgrind, Memcheck, profilování paměti, únik paměti, wxWidgets, C++

ABSTRACT

This paper's main topics are memory profilers and generally it focuses on memory leak detection techniques in C/C++ based computer programs. The practical purpose of this paper was to implement a plugin for the CodeLite IDE to enhance the IDE with the Memcheck tool that is a part of the Valgrind framework.

The analytical part of this paper focuses on the differences between a static and dynamic program analysis, describes the various ways of program profiling. Technical

implementation of profilers is also examined to some extent. The paper continues with a comparison of memory profilers and looks at some of the most important ones in more detail. Discussion about GUI design and a comparison of existing graphical front ends for Valgrind are also covered. The analytical part ends with the CodeLite IDE description focusing on both the end user's perspective and the possibilities of extending the IDE with plugins mainly in relation to the wxWidgets library that CodeLite is built on.

The actual plugin design and implementation are the main topics of the practical part. The technical problems and limitations are discussed here as well as their possible solutions. Future plugin extension possibilities are discussed in the very end followed by a user documentation.

Keywords: CodeLite, Plugin, Valgrind, Memcheck, Memory profiling, Memory leak, wxWidgets, C++

Děkuji především vedoucímu práce panu Ing. Michalu Bližňákovi, Ph.D. za pomoc při práci na tomto projektu a zároveň za rady a konzultace tvůrcům CodeLite, panu Eranu Ifrahovi a panu Davidu Hartovi.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně ……………………
podpis diplomanta

OBSAH

ÚVOD 10
I TEORETICKÁ ČÁST 11
1 PROFILOVÁNÍ 12
2 PROFILOVACÍ NÁSTROJE 16
2.1 VALGRIND 16
2.1.1 Memcheck 17
2.1.2 Cachegrind 21
2.1.3 Callgrind 21
2.1.4 Massif 21
2.1.5 Helgrind a DRD 21
2.2 DR. MEMORY 22
2.3 MEMWATCH 23
2.4 BOUNDSCHECKER 23
2.5 IBM RATIONAL PURIFY 23
2.6 PARASOFT INSURE++ 24
3 IDE 26
3.1 NÁVRH UŽIVATELSKÉHO ROZHRANÍ 27
3.2 IDE S PODPOROU VALGRINDU 29
3.2.1 Valkyrie 29
3.2.2 Alleyoop 31
3.2.3 Memcheckview 32
3.2.4 Eclipse 33
3.2.5 Code::Blocks 34
3.2.6 KDevelop 36
4 CODELITE 37
4.1 HISTORIE 37
4.2 ROZHRANÍ A FUNKCE 37
4.3 WXWIDGETS 41
4.4 ROZŠÍŘENÍ POMOCÍ PLUGINU 42
II PRAKTICKÁ ČÁST 45
5 NÁVRH A IMPLEMENTACE PLUGINU 46
5.1 SPUŠTĚNÍ 46
5.2 KONTEJNERY PRO ZOBRAZENÍ CHYB 47
5.3 VNITŘNÍ REPREZENTACE 48
5.4 IKONY 50
6 PROGRAMOVÉ ROZHRANÍ PLUGINU 52

6.1 CELKOVÝ PŘEHLED A PRINCIP ČINNOSTI ... 52
6.1.1 Spuštění testu ... 53
6.1.2 Procházení chyb ... 54
6.1.3 Potlačování chyb ... 56
6.2 MOŽNOST ROZŠÍŘENÍ ... 57
6.2.1 Rozhraní ... 58
6.2.2 Vnitřní reprezentace ... 59
6.2.3 Třída nastavení ... 61
6.2.4 Dialog pro nastavení ... 61
6.2.5 Registrace nového procesoru ... 61
7 UŽIVATELSKÁ DOKUMENTACE ... 62
7.1 SPUŠTĚNÍ TESTU ... 62
7.2 PROCHÁZENÍ CHYB ... 62
7.3 POTLAČOVÁNÍ CHYB ... 63
7.4 NASTAVENÍ ... 64
ZÁVĚR ... 66
ZÁVĚR V ANGLIČTINĚ ... 68
SEZNAM POUŽITÉ LITERATURY ... 70
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 75
SEZNAM OBRÁZKŮ ... 76
SEZNAM PŘÍLOH ... 77

ÚVOD

V začátcích se počítače ovládaly pomocí děrných štítků, později v textové konzoli a dnes pomoci GUI. V budoucnu to bude možná holografy nebo přímým propojením do mozku? To se dozvědí nejspíš asi až naši vnuci.

Z hlediska práce s uživatelským rozhraním se programátoři neliší od ostatních uživatelů. Možná jen častěji dokáží lépe popsat jakými technickými prostředky dosáhnout toho, co by chtěli, a to je pracovat pohodlně, efektivně a produktivně.

Vývoj softwaru pro programátory dospěl k používání IDE. IDE je program, který integruje komponenty, jež programátor nejvíce potřebuje a používá. Mezi základní patří editor zdrojového kódu, kompilátor, interpret nebo debugger. Dalšími věcmi, bez kterých si neumím představit pohodlné IDE, je například automatické doplňování textu a kódu, nebo správa zdrojů celého projektu, jakožto i vyhledávání a navigace v daném projektu.

Součástí IDE bývají dále prvky, které neslouží přímo k produkci kódu, jedná se například o nástroje ke správě verzí a velmi často také různé analytické pomůcky. Strojová analýza je velmi důležitou součástí vývoje software, dává programátorovi možnost ověřit si, jak se mu podaří jeho myšlenky a záměry realizovat pomocí technických prostředků, které má k dispozici. Nejedná se jen o prosté ověřování a testování, zda program běží, je stabilní nebo provádí výpočty správně. Analytický nástroj informuje programátora o tom, jak dobře pracuje jeho program, a to například z hlediska efektivity časové nebo efektivity využívání zdrojů, případně bezpečnosti provozu.

Proces analýzy má dvě významné části, jednak jde o realizaci nástroje, který provádí samotné zkoumání, a pak otázka jak zjištěné skutečnosti prezentovat programátorovi tak, aby mu byly přínosem. Analytický software je schopen vyprodukovat ohromné množství dat, není však v silách průměrného člověka se v nich zorientovat, natož z nich vyvodit nějaké praktické opaření. Prezentace analytických dat a jejich případná integrace do IDE je podle mého názoru stejně důležitá jako kvalita analýzy samotné.

Tato práce se zaměřuje konkrétně na jednu oblast dynamické analýzy, tj. profilaci paměti, resp. na nástroje, které ji provádějí. Součástí práce je posléze implementace rozšíření do konkrétního vývojového prostředí s GUI, které prezentuje výstup vybraného profilačního nástroje.

I. TEORETICKÁ ČÁST

1 PROFILOVÁNÍ

Analýza programů je nedílnou součástí vývoje software, umožňuje hodnotit správnost programů a pomáhá při jejich optimalizaci. Rozlišujeme analýzu statickou, jejímž principem je zkoumání zdrojových kódů bez jejich překladu na program nebo jeho spuštění, a analýzu dynamickou, která je prováděna za běhu programu. V tomto kontextu budu mluvit pouze o zkoumání pomocí speciálních analytických programů, případně jiných technických automatizovaných prostředků. Za analýzu programu je možné považovat i ruční inspekci kódu, ale ta neodstraňuje ten hlavní problém, který je třeba řešit, a tím je chyba vytvořená člověkem, a to ať už při návrhu, nebo implementaci. Na druhé straně vyhodnocení analýzy, tedy simulaci vyšší lidské duševní činnosti, nelze, alespoň zatím, simulovat.

Díky tomu, že nástroje pro statickou analýzu pracují přímo nad zdrojovým kódem, tak v případě, že odhalí nějaký problém, dokáží ho přesně lokalizovat přímo v kódu. Složitost, schopnosti a rozsah práce programů pro statickou analýzu se velmi liší.

Statickou analýzu kódu používá programátor takřka nepřetržitě při své práci, aniž by si to třeba uvědomoval. Každý překlad zdrojových kódů (i do bajtkódu pro interpretované jazyky) spouští statickou analýzu, ta na této úrovni může provádět kontrolu typů nebo stylů kódování. Je to jednoduchý a zároveň velmi efektivní způsob okamžité korekce překlepů, opomenutí nebo nepozornosti programátora. Díky porozumění kódu může IDE umožnit programátorovi navigaci ve zdrojových kódech na základě syntaxe, tedy skok na definici nějakého symbolu, pohyb po celých blocích kódu. Další mimořádně užitečné funkce jsou doplňování kódu nebo například bezpečné přejmenování symbolu, nebo také automatické generování kódu. Nejsložitější nástroje pracující na vysoké úrovni abstrakce jsou nástroje pro kontrolu vlastností a verifikaci programů, a to formálními metodami na principech ověřování modelu nebo logickou dedukcí. Zde je třeba učinit úvahu o vyčíslitelnosti. Přímou redukcí na halting problém (problém zastavení) je možné dokázat, že problém nalezení všech možných chyb v obecném, jakémkoli programu je nerozhodnutelný, tedy neexistuje metoda, s jejíž pomocí by bylo možno strojově rozhodnout, zda v programu ještě nějaké chyby jsou nebo už ne. Neznamená to ale, že je správné vzdát snahu o takovou analýzu, celou řadu nerozhodnutelných problémů lze řešit aproximací s uspokojivou přesností. Další cesta pro formální analýzu je použití specifického jazyka, například funkcionálního.[9]

Dynamická analýza nezkoumá program v celé své komplexitě, jejím principem je monitoring chování programu vždy při jednom spuštění s jedním vstupem. Jako vstup lze zde chápat i úkony provedené uživatelem, to v případě, jedná-li se o program s GUI, nebo komunikace s jinými prvky na síti, nebo jiným periferním zdrojem. Podstatné je to, že při opětovném spuštění a analýze programu mohou být naměřené výsledky zcela jiné, pokud i onen obecný vstup bude jiný. Dynamická analýza je tedy velmi citlivá na vstup a nelze její pomocí zkoumat program na obecné vlastnosti. Její výhodou je, že pro určitý vstup lze chování programu ohodnotit jistě a jednoznačně. Dynamický analyzátor nemá nadhled na celý program, zkoumá přesně vymezená kritéria. Na rozdíl od pokročilých statických analyzátorů není tedy pro implementaci dynamického analyzátoru třeba hledat složité algoritmy nebo modely. Náročnost spočívá spíše v zajištění dostatečných zdrojů pro test. Shromáždění sledovaných hodnot je prováděno v jasně vymezených místech, při určitých událostech, nebo v dobře měřitelných intervalech, v důsledku toho lze hodnotit granularitu sběru dat.[10]

Existuje několik způsobů realizace dynamických analyzátorů. Při instrumentaci kódu dochází ke vkládání příkazů pro sběr míst na vhodná místa. Těmito místy jsou nejčastěji například počátek a konec funkce. Vkládání navíc může být prováděno před kompilací, instrumentován bude tedy zdrojový kód, a to nejen automaticky, ale i ručně programátorem. Další možností je instrumentace binárního souboru, tedy již zkompilovaného programu, nebo je kód instrumentován těsně před spuštěním, kdy je zkoumaný program spouštěn a řízen ve virtuálním prostředí analyzačního nástroje.

Interpretované jazyky jako Java, .NET, Python nebo Ruby poskytují rozhraní, které umožňuje analyzátoru sledovat události emitované interpretovaným programem. Příklad události je vznik nebo zrušení objektu, volání funkce, nebo spuštění vlákna. Existuje například ještě možnost instrumentace interpreteru, taková varianta umožňuje sledovat zkoumaný program na úrovni jednotlivých instrukcí bajtkódu. Při těchto dvou metodách dochází ke zkreslení výsledků měření, neboť akce sběru dat jsou vykonávány vlastně zkoumaným programem, ty stojí čas a také další zdroje.

Tento nedostatek řeší statistická analýza založená na vzorkování. Program je přerušován v pravidelných intervalech a je zkoumán jeho čítač instrukcí. Tato metoda nijak nezasahuje do zkoumaného programu, který tak běží skutečnou rychlostí, a celý systém není zatížen režií průběžného předávání a sběru dat. Bohužel i přerušení negativně ovlivňuje systém, na

kterém probíhá měření, a tím i přesnost měření. Toto lze řešit specializovanými hardwarovými prostředky. Výstupem statistické analýzy nejsou přesné hodnoty, ale aproximace, při celkovém pohledu na program však tento druh analýzy poskytuje přesnější a reálnější výsledky.[11]

Profilování je forma dynamické analýzy zaměřená na měření výkonu programu a hledání míst v programu, která je vhodné optimalizovat. Profilovat program lze na základě různých kritérií, může to být čas a počet volání funkcí, větších částí programu, systémových volání nebo knihoven. Čas a intenzita vykonávání jednotlivých částí programu mohou být vyjádřeny jak v absolutních hodnotách, tak i v poměru k celku. Podle produkovaného výstupu lze profilery zaměřující se na čas a četnost volání funkčních bloků rozdělit do tří skupin:

- Flat profilery – měří dobu zpracování funkcí, neinformují o hierarchii ani četnosti jejich volání.
- Call-graph profilery – ukazují čas i počet zpracování funkcí stejně jako strom jejich volání.
- Input-sensitive profilery – ukazují vztah výkonnostních parametrů ve vztahu k velikosti vstupních dat. Jsou tedy schopny odhalit například nelineární složitost programu.[13]

Kromě časové složitosti lze odhadovat i paměťovou složitost, a to na základě využití fyzického množství paměti programem při jeho běhu. Obecně je možné kromě paměti profilovat i na základě využití dalších systémových zdrojů, jako jsou soubory, roury, sokety nebo vlákna. Profilovat je možné i z hlediska spotřeby energie, nebo i bezpečnosti. Například metoda Program Shepherding monitoruje původ jednotlivých instrukcí programu a snaží se odhalit narušení toku řízení, které by mohlo být způsobené přítomností exploitu.[14]

Profilaci paměti je možné realizovat dvěma způsoby. Jedná se profilaci za účelem optimalizace spotřeby paměti, a potom hledání úniků paměti, tzv. memory leaks. V prvním případě, i když je s pamětí nakládáno formálně správně, to může být neefektivní. Větší problém ale stanoví únik paměti. Odhalování úniků paměti se mírně vymyká obecné definici profilování, částečně by ji bylo možné zařadit do ladění. Únik paměti je důsledek nesprávné práce s pamětí, kdy program alokuje paměť, ale neuvolňuje ji po tom, kdy ji už

nevyužívá. K úniku paměti dochází, pokud je program chybně napsán. Je možné ho spustit a provozovat, ale postupem času, když program vyžaduje další a další paměť, dochází k odstránkovávání paměti ostatních programů do odkládacího prostoru, což může vést ke snížení výkonu systému. Když program vyčerpá všechnu dostupnou paměť, může být násilně ukončen, v horším případě může dojít k havárii celého systému. Programy nejsou běžně vybaveny schopností reagovat na nedostatek paměti jiným způsobem než se ukončit, to může vést ke ztrátě dat nebo jiným závažným následkům.

Existuje mnoho technik a principů jak předcházet nebo omezovat důsledky úniků paměti, a ačkoli jsou mnohé z nich velmi účinné, řeší jen dílčí problémy. Například v současných operačních systémech je každému procesu přiděleno určité množství virtuální paměti, která je vždy po skončení běhu uvolněna jako celek. Vestavěné systémy však už takovou schopnost nemají.

Některé programovací jazyky disponují takzvaným garbage collectorem, což je vlastně automatická správa paměti. Garbage collector se stará o uvolnění objektu z paměti, když už ho není zapotřebí, technicky se jedná o situaci, kdy není ničím referencován. V jazycích jako C/C++ se musí programátor sám postarat o korektní uvolnění paměti, kterou dynamicky alokoval, jinak lze tuto paměť okamžitě považovat za ztracenou. Práce na této nižší úrovni abstrakce umožňuje vybudovat preciznější a výkonnější řešení než s použitím garbage collectoru, ale zároveň je komplikovanější a náchylnější k lidské chybě. I pro C/C++ existují knihovny, které přidávají funkcionalitu srovnatelnou s garbage collectorem, ale její použití je mnohem složitější než u jazyků, které mají správu paměti zabudovanou v sobě přirozeně. Mnoho programátorů Javy, .NETu nebo Pythonu si tak myslí, že díky fargabe collectoru se jich úniky paměti netýkají, bohužel opak je pravdou. V každém konkrétním jazyce existují případy, kdy garbage collector z pohledu pohodlného programátora selhává. Jedná se například o situaci cyklické reference, kdy na sebe dva objekty mají silnou referenci, a tak ani jeden nemůže byt uvolněn a umožnit uvolnění druhého. Použitím nevhodného řešení nebo nesprávným použitím prostředků daného jazyka lze vždy zapříčinit unik paměti.[15]

V dalším textu se zaměřím na profilování paměti, a to konkrétně programů psaných v jazycích C/C++.

2 PROFILOVACÍ NÁSTROJE

V této kapitole jsou kromě Valgrindu popsány další uznávané profilovací nástroje, které by mohly byt použity jako back-end pro můj plugin. Jelikož je cílem této práce vytvořit rozšíření pro IDE CodeLite, které je určeno pouze pro vývoj v C/C++, zaměřil jsem na profilovací nástroje, které jsou určeny právě pro C/C++. Dalším kritériem pro nástroje je jejich vhodná licence.

Valgrind a Dr. Memory splňují všechny teoretické předpoklady. memwatch je bohužel určen především pro C a nehodí se pro C++. O BoundsCheckeru jsem v porovnání s ostatními nástroji nalezl nejméně informací, nicméně až na licenci by jeho použití bylo teoreticky možné. Purify a Insure++ by byly stejně použitelné jako Valgrind nebo Dr. Memory, potíž je pouze v jejich nesvobodnosti.

2.1 Valgrind

V původní verzi se jednalo jen o nástroj pro hledání úniků paměti a přetečení vyrovnávací paměti. Do současnosti se vyvinul v obecné prostředí, které umožňuje vytváření různých nástrojů pro dynamickou analýzu programů primárně psaných v C/C++, ale je schopen diagnostikovat binární soubor programu psaný v kterémkoli jiném jazyce.[1]

Technicky je Valgrind virtuální stroj pracující na principu instrumentace. Zkoumaný program za běhu převede na univerzální strojový kód, ten je posléze instrumentován a výsledek je dynamicky převáděn na strojový kód hostitelského systému a vykonáván. Právě to, že je instrumentován mezikód, umožňuje vytvářet další diagnostické nástroje sdílející stejné rozhraní. Nevýhodou je značné zpomalení běhu zkoumaného programu. Základní zpomalení interpretací je 4-5ti násobné, a dále zpomalení záleží na použitém diagnostickém nástroji. Další nevýhodou je velká spotřeba paměti.

Valgrind je free, je možné ho stáhnout, používat i měnit, vše v souladu s licencí GPL. Valgrind funguje na více platformách, je to Linux, Mac OS X, FreeBSD nebo Android. Záleží však na konkrétní hardwarové architektuře a verzi. Valgrind je stále aktivně vyvíjen, pro zjištění, zda bude fungovat na konkrétním systému na konkrétní architektuře, je třeba se informovat na oficiálních stránkách projektu nebo kontaktovat vývojáře. Valgrind nemá oficiální verzi pro MS Windows, profilace programů pro Windows je však možná s použitím Wine.[20]

Existuje projekt „Valgrind for Windows“, který se pokouší být oním portem pro Windows. Bohužel je jen ve fázi prealfa, což znamená, že je to netestovaná vývojová verze s neustálenou funkcionalitou, dostupná v podobě zdrojových kódů. Vývojáři originálního Valgrindu odhadli, že jeho portování vyžaduje tolik změn, že je stejně rozsáhlé jako vývoj Valgrindu samotného. Bohužel v době vzniku této diplomové práce je to už skoro dva roky, co proběhla poslední aktualizace. Kombinace těchto skutečností nevěstí pro zájemce o Valgrind ve Windows nic dobrého.[21]

Ačkoli Valgrind není přenositelný, je velmi populární. Ze světa Linuxu ho znám jako synonymum pro detekci úniků paměti. Jednak je to tím, že je velmi univerzální, není třeba provádět žádné úpravy zdrojového kódu, aby mohl být nasazen, přičemž funguje pro programy napsané v kterémkoli jazyce. Dalším faktorem je určitě jeho stabilita, spolehlivost a dopracovanost. Ale možná to rozhodující je, že je možné ho nainstalovat zcela zdarma v plné verzi jedním příkazem z oficiálního repozitáře snad každé Linuxové distribuce.

Součástí Valgrindu jsou specializované profilační a diagnostické nástroje, použití Valgrindu demonstruji na nástroji Memcheck. Ostatní sdílejí stejné ovládací rozhraní, ale produkují jiný výstup. Zmíním se o nich jen stručně.

2.1.1 Memcheck

Memcheck je první a hlavní nástroj Valgrindu, detekuje problémy při správě paměti tak, že kontroluje všechna čtení a zápisy do paměti a zachycuje všechna volání malloc, new, free a delete. Plně podporuje programy s vlákny.

Memcheck pracuje s těmito typy chyb:

- Čtení nebo zápis mimo povolený rozsah
- Použití neinicializovaných proměnných
- Použití neinicializovaných proměnných při systémových voláních
- Nesprávné uvolnění: když program provede uvolnění paměti dvakrát
- Použití nesprávné funkce pro uvolnění paměti: např. paměť alokovanou pomocí new nelze uvolnit pomocí delete[]
- Překrytí bloků při kopírování

- Únik paměti

Testovaný program je velmi vhodné zkompilovat s přidanými debugging informacemi (v gcc přepínač -g), díky tomu bude moci Valgrind zobrazit k jednotlivým chybám nejen umístění ve zkompilovaném souboru, ale i jméno souboru v čísle řádku zdrojového kódu. Občas může optimalizace kódu při kompilaci zmást Valgrind, ale zpomalení neoptimalizovaného kódu se ve Valgrindu projeví mnohonásobně, doporučeným kompromisem je mírná úroveň optimalizace (-O).

Memcheck se spouští z příkazové řádky:

```
valgrind --tool=memcheck [valgrind-args] program [program-args]
```

Memcheck je výchozí nástroj, proto ho není nutné explicitně zadávat pomocí `--tool=memcheck`. Po zadání všech přepínačů pro Valgrind jsou dalšími argumenty zkoumaný program a jeho případné argumenty.

Memcheck rozeznává řadu přepínačů, které ovlivňují výpis chyb, ale chování při testu, jejich detailní popis je v manuálu. Jejich použití je ale navázáno na pochopení fungování Valgrindu, Memchecku a programování v C/C++ obecně, což vyžaduje znalosti a zkušenosti.

Já jsem vyzkoušel tyto přepínače pro Memcheck:

- `--track-origins=<yes|no> [default: no]`

  Jestliže Memcheck nalezne neinicializovanou proměnnou, tato volba vynutí, aby se ve výpisu objevilo, kde byla zmíněná proměnná alokována.

- `--leak-check=<no|summary|yes|full> [default: summary]`

  Jak je vidět na konci výpisu (PŘÍLOHA P II: Memcheck – textový protokol), Memcheck ve výchozím nastavení vypíše pouze shrnutí o únicích paměti, až explicitní volba `yes` nebo `full` zapříčiní, že se ve výpisu objeví konkrétní úniky paměti.

Základními přepínači při práci s Valgrindem jsou:

- `--xml=<yes|no> [default: no]`

  Valdrind je unikátní v tom, jak snadno jde pracovat s jeho výstupem. Nejenom že sám přímo generuje výstupní protokol ve formátu XML, ale navíc svůj výstup může

směrovat kromě souboru také do souborového deskriptoru nebo do soketu. Z tohoto hlediska asi není vhodnějšího back-endu. Příklad toho, jak pro konkrétní program (PŘÍLOHA P I: Ukázkový zdrojový kód) vypadá výstup textový a ekvivalentní výstup formátovaný jako XML, je v přílohách PŘÍLOHA P II: Memcheck – textový protokol a PŘÍLOHA P III: Memcheck – Protokol XML. V tomto programu našel Memcheck dvě chyby, první je zápis v hranici alokovaného bloku, druhá chyba je únik paměti způsobený neuvolněním paměti. U první chyby je možné si všimnout, že obsahuje dva stromy volání. V tomto případě je to díky přepínači `--track-origins=yes`. První strom volání odkazuje na místo, kde se chyba projevila, druhý odkazuje na místo alokace paměťového bloku. Chyba totiž může být opravena dvěma způsoby, bud upravením indexu, kam se zapisuje, nebo zvětšením alokovaného bloku. Z XML protokolu je vidět, jak Valgrind chyby strukturuje. Každá chyba kromě typu, titulku a zásobníku (stromu, nebo posloupnosti volání) může obsahovat přídavnou (auxiliary) sekci, která má velmi podobnou strukturu jako celá chyba.

- `--fullpath-after=<string>`

  Valgrind ve výchozí podobě tiskne jenom jména zdrojových souborů bez cest. V projektu, kde se některá jména mohou opakovat, je to matoucí a nepoužitelné. Tímto přepínačem lze zapnout výpis jmen souborů včetně cest, a to tak, že Valgrind vezme cestu k souboru a vypíše jen tu část, která následuje za řetězcem specifikovaným tímto přepínačem. Zvolí-li uživatel pouze `--fullpath-after=` bez žádného řetězce, vypíše se plná cesta. Tato volba je vhodná pro strojové zpracování, protože cesty lze ořezat kdykoli později v jiném programu.

- `--gen-suppressions=<yes|no|all> [default: no]`

  Valgrind je schopen najit opravdu obrovský počet chyb, většinou bohužel v cizích knihovnách, které uživatel nemůže opravit. Valgrind dává možnost tyto chyby potlačit neboli skrýt, tak že se v protokolu neobjeví. Počet potlačených chyb se vypisuje ve shrnutí. K potlačení chyb slouží vzory uložené v textovém souboru. Volba `all` v tomto přepínači znamená, že do výstupního protokolu se ke každé chybě vygeneruje právě takový vzor, který jí potlačí. Volba `yes` znamená, že se Valgrind před každým výpisem chyby zeptá, zda ji má vytisknout.

- `--suppressions=<filename>`

  `[default: $PREFIX/lib/valgrind/default.supp]`

  Tento přepínač říká Valgrindu, který soubor se vzory pro potlačení má použít, takových souborů je možné zadat až sto.

  Suppression soubor je textový soubor obsahující řádkové komentáře uvozené znakem mříží # a potom jednotlivými vzory. Každý vzor je ohraničen složenými závorkami, jak otevírací, tak zavírací závorka musí být na novém řádku. Vzhled potlačovacího pravidla je vidět v příloze (PŘÍLOHA P II: Memcheck – textový protokol), kde jsou dva příklady. Tělo pravidla má následující strukturu:

  - První řádek – jméno: libovolný řetězec, slouží k identifikaci, pro autora
  - Druhý řádek – jméno nástroje a jméno, nebo spíše kód chyby: potlačení zafunguje pro vyjmenované nástroje pro daný typ chyby. Typy jsou shodné s těmi, které jsem vyjmenoval výše.
  - Následující řádek – přídavný parametr: některé typy chyb vyžadují přídavný parametr. Například u „Použití neinicializovaných proměnných při systémových voláních“ je to jméno parametru systémového volání.
  - Ostatní řádky – strom volání funkcí. Nahoře jsou nejvíce zanořené funkce. Volá-li funkce A funkci B a zároveň B volá C, tak C bude nad B a A bude ještě níže. Kromě jmen funkcí je možné zadat i jméno sdíleného objektu, tedy zkompilovaného kódu v souboru. Každý řádek tedy obsahuje slovo `fun` nebo `obj` potom oddělovač dvojtečku a potom jméno. Ve jménech jsou povolené univerzální znaky * a ? pro libovolný počet libovolných znaků nebo pro jeden libovolný znak. Stejně tak není nutné definovat přesně strom volání shora dolů, povoleným textem na řádku jsou i tři tečky „…“, které znamenají libovolný počet libovolných funkcí.

    Při psaní pravidel je třeba dávat pozor na důležitou věc. Jména funkcí C++ je třeba do suppression souboru zadávat v takzvané mangled formě, což je vnitřní reprezentace jména funkce v kompilátoru. Například `_ZN11CodeLiteApp6OnInitEv` je ve skutečnosti `CodeLiteApp::OnInit()`. Valgrind tak při běhu nemusí dělat překlad a

zmenšuje se tak zpomalení běhu testu. Chce-li uživatel v logu vidět zkrácené jména funkcí, musí přidat argument `--demangle=no`. To naštěstí není běžně nutné, protože když Valgrind sám generuje vzory, dělá to vždy ve zkrácené formě.

Další možností pro Memcheck a Valgrind jsou pečlivě popsány v dokumentaci.[6]

2.1.2 Cachegrind

Cachegrind je profiler vyrovnávací paměti, který simuluje cache procesoru. Je schopen identifikovat počet cache misses (netrefení se do vyrovnávací paměti), odkazů do paměti a instrukcí provedených pro každý řádek zdrojového kódu, jakožto i pro shrnutí pro každou funkci, modul, nebo celý program. Cachegrind zpomaluje vykonávání programu 20-100 násobně.

2.1.3 Callgrind

Callgrind je rozšířením Cachegrindu, kdy navíc ukazuje strom volání. Doporučeným nástrojem pro vizualizaci výsledků Cachegrind a Callgrind je KCachegrind.[22]

2.1.4 Massif

Massif je profiler haldy a zásobníku, přičemž ten je zahrnut do profilace, jako by to byla část haldy. Halda je oblast paměti, která je obvykle alokovaná funkcemi jako „malloc“ a slouží jako globální sklad paměti pro program. Zásobník je místo, kde jsou uchovávána lokální data funkcí. Massif v pravidelných intervalech pořizuje snímky haldy a produkuje graf jejího využití, kde je zahrnuta informace o tom, která část programu zabírá nejvíce paměti. Součástí výstupu jsou i textové informace s přesnějšími údaji o tom, kde je paměť alokovaná. Zpomalení programu vlivem profilace pomocí Massifu je asi 20ti násobné.

2.1.5 Helgrind a DRD

Helgrind a DRD jsou nástroje pro hledání souběhů ve vícevláknových programech založených na POSIX vláknech. To je také důvod, proč tyto dva nástroje nelze jednoduše přenést na Windows. Souběh je chyba v programu způsobená tím, že se dvě nebo více vláken snaží přistupovat ke sdílené paměti a jejich pořadí nebo načasování není definováno. Tyto nástroje hledají paměť, ke které je přistupováno z více vláken, ale pro

kterou nelze dohledat konzistentní synchronizační mechanismus, např. mutex. Tyto nástroje vůči sobě stanoví konkurenci a záleží na konkrétní situaci, který je výhodnější.

2.2 Dr. Memory

Dr. Memory je nástroj pro monitorování paměti schopný identifikovat problémy svázané s jejím používáním, kterými jsou např. úniky paměti, přístup mimo správný rozsah paměti, neinicializovanou paměť, použití vadných ukazatelů, chyby při použití GDI (graphic device interface) ve Windows, nebo handlů ve Windows. Pracuje i s mnohovláknovými programy. Je založen na platformě DynamoRIO, což je systém pro dynamickou manipulaci s kódem. Za běhu programu dekóduje jeho instrukce a umožňuje je měnit, jedná se tedy o dynamickou instrumentaci programu v binární podobě.

Dr. Memory se staví do role přímého konkurenta Valgrindu, resp. jeho nástroji memcheck, a poukazuje na svoji větší výkonnost. Z abstraktního pohledu pracuje jako virtuální stroj interpretující zkoumaný program, tedy podobně jako Valgrind, a i pracovní postup s ním je podobný jako s Valgrindem. Dr. Memory je licencován jako open-source. Díky přenositelnosti DynamoRIO je i Dr. Memory přenositelný, funguje na Linuxu, Macu i ve Windows, na 32 i 64 bitových architekturách. Na jeho vývoji se v současnosti stále pracuje, umí diagnostikovat jen 32 bitové programy (i když analýza probíha na 64 bitovém systému). Kvůli tomuto omezení je program pro testovací účely vždy explicitně kompilovat jako 32 bitový. Zatím nedisponuje žádným GUI programem pro prezentaci výsledků analýzy, produkuje pouze textový výpis do souboru, čitelný člověkem.[33][34] Formát chybových hlášení lze upravit mnoha přepínači na příkazové řádce a mělo by tedy být možné strojově načíst protokol o chybách i s použitím regulárních výrazů a použít ho v jiné aplikaci. Stejně jako Valgrind tak i Dr. Memory umožňuje potlačování chyb ve výstupu a i jejich definování řeší stejným způsobem, tedy pomocí suppresion souboru. V takovém textovém souboru je třeba specifikovat chyby, které budou potlačeny. Ty jsou určeny mezi jinými typem a místem vzniku. Stejně jako ve Valgrindu je možné používat ve jménech modulů nebo jménech funkcí univerzální symboly * a ? a je možné vynechat několik funkcí ve stromu volání pomocí „…“. Jen na rozdíl od Valgrindu produkuje Dr. Memory automatické záznamy pro potlačení všech nalezených chyb do zvláštního souboru, nikoli přímo do hlášení o chybách.[31][32]

2.3 memwatch

Memwatch je svobodný software určený k detekci úniků paměti a přístupu mimo vyhrazenou oblast. Funguje na principu instrumentace kódu, do každého zdrojového souboru je třeba přidat jeden hlavičkový soubor. Ten redefinuje metody pro alokaci paměti jako je „malloc“, díky čemuž memwatch získává přehled o alokacích v programu. Memwatch funguje na všech platformách, které podporují ANSI C kompilátor. Je určen hlavně pro programy v C a má jen základní podporu pro C++. Tento je velmi jednoduchý, skládá se vlastně jen z jednoho zdrojového souboru a jednoho hlavičkového souboru. Výstup programu má textovou formu čitelnou člověkem, obsahuje jména souborů a čísla řádku odkazující na zjištěný problém.[23]

2.4 BoundsChecker

BoundsChecker je komplexní nástroj pro detekci chyb spojených s využíváním paměti v programech psaných v C nebo C++. Používá dvě strategie monitoringu – neintrusivní ActiveCheck, který je schopen zachytit chybu při volání funkcí z externích knihoven nebo zdrojů. Druhý způsob, FinalCheck, je mnohem hlubší, je schopen zjistit chyby i při volání funkcí pouze uvnitř zkoumané aplikace. K provedení FinalChecku je nutná instrumentace kódu. BoundsChecker detekuje úniky paměti, přetečení, neinicializované proměnné, uvíznutí v mnohavláknových aplikacích, využití ukazatelů, překrývání a mnohé další. BoundsChecker je proprietární software, je součástí většího softwarového balíku DevPartner, který slouží k vývoji, ladění a diagnostice programů psaných v jazycích C/C++, Java a .Net. Tento balík je primárně určen pro nasazení v MS Visual Studiu a je tedy do něj graficky integrován. Existuje ale i samostatná verze BoundsCheckeru, která by měla pracovat na příkazové řádce. Export dat do xml je možný a tím pádem jejich zpracování v dalších programech. BoundsChecker funguje na operačních systémech MS Windows architektur 32 i 64 bitových a není jinam portovaný.[24][25][26]

2.5 IBM Rational Purify

Purify je ladicí a diagnostický nástroj pro odhalování chyb při práci s pamětí i v mnohavláknových programech. Jedná se například o používání neinicializované paměti, čtení a zápis za hranice pole, za hranice uvolněné paměti, nebo pomocí nulových

ukazatelů. Purify je proprietární software, nabízený ve více verzích pro UNIXy i MS Windows, určený hlavně pro hledání chyb v programech C/C++, má však i podporu pro Javu a .NET. Spolupracuje s dalšími diagnostickými programy rodiny Rational, které jsou schopné monitorovat celkové využití paměti nebo vytvářet grafy volání.

Purify používá k práci instrumentaci kódu. Programátor musí svůj program instrumentovat v okamžiku linkování programu. Purify je volán místo linkeru ale s podobnými parametry. Instrumentace se tedy vztahuje i na kód knihoven, jejichž kód programátor nemá přímo k dispozici. Takto rozšířený program při spuštění sám generuje diagnostické informace. Výchozím způsobem práce s instrumentovaným programem je pomocí GUI programu pro zobrazení diagnostických informací, ten se spustí se spuštěním zkoumaného programu. GUI zobrazuje jednotlivé chyby, k nim strom volání funkcí a pro konkrétní místo s chybou zobrazí okolí kódu, který způsobuje chybu, po dvojkliku se toto místo může zobrazit například ve Visual Studiu. Věřím, že toto chování lze ještě nastavit. Vzhled tohoto programu je velmi podobný programu Valkyrie 3.2.1, který popíšu níže. Instrumentovaný program rozpozná celou řadu argumentů pro příkazovou řádku, program je možné spustit bez GUI a zadat vytvoření binárního souboru se záznamem a teprve později ho načíst. Je možné vytvořit i textový soubor logu, který organizován stejně jako výpis v grafickém nástroji je čitelný i pro člověka. S velkou pravděpodobností ho lze načíst i strojově pomocí regulárních výrazů. Tvar i množství výpisu lze řídit z instrumentovaného programu, Purify totiž poskytuje rozhraní pro takový postup, vyžaduje to ale zásah do zdrojového kódu programu.

Stejně jako Valgrind i Pufiry umožňuje potlačování chyb ve výpisu, a to vlastně stejným způsobem – vytvořením suppresion souboru, který obsahuje seznam chyb, které se neobjeví ve výstupu, každá z nich je definovaná typem a místem vzniku.[27][28]

2.6 Parasoft Insure++

Insure++ je proprietární nástroj na detekci paměťových chyb v programech psaných v C/C++. Je schopen odhalit různé druhy chyb, o kterých jsem psal výše, podporovány jsou i programy s více vlákny. Insure++ pracuje ve dvou režimech. V prvním je patentovaným algoritmem instrumentován zdrojový kód, který je později zkompilován pomocí Insure++, tento způsob analýzy detailní. Kromě toho Insure++ provádí při kompilaci i statickou analýzu. Druhý způsob je méně hloubkový, zato rychlejší, instrumentace je prováděna jen

při linkování již zkompilovaného programu, i to je nutné provést pomocí Insure++ linkeru. Stejně jako Purify je Insure++ přenositelný mezi UNIXy a MS Windows a jeho součástí je grafický nástroj pro zobrazování výstupu měření, který je taky podobný programu Valkyrie 3.2.1.

Insure++ produkuje holý textový výstup čitelný člověkem podobný tomu ve Valgrindu, ten obsahuje štítek s označením chyby, popis a také strom volání. Těch může ale chyba obsahovat více, např. únik paměti obsahuje tři stromy volání: kde byla paměť alokována, kde uvolněna a kde došlo k chybě. I v Insure++ může programátor potlačovat chyby jejich specifikováním v GUI nebo v textovém souboru. Stejně jako ve Valgrindu je možné používat univerzální symboly * a ? pro mnoho znaků nebo jeden libovolný. Hvězdičku je také možno použít ve stromu volání funkcí, ta akceptuje libovolné množství zanoření ve funcích o libovolných názvech.[29][30]

3 IDE

IDE – Integrated development environment nebo interactive development environment, což česky znamená integrované nebo interaktivní vývojové prostředí, je software určený pro psaní jiného software. Jeho základními prvky jsou editor, správce souborů, kompilátor a prostředky pro spuštění vyvíjeného programu a k jeho debugování. [16] Všechny tyto prvky jsou propojeny jak funkčně, tak z pohledu rozhraní, mají poskytovat jednotnou grafickou podobu a ovládání. IDE umožňuje uživateli spravovat nastavení prostředí jako celku a podle toho provádět konfiguraci jednotlivých komponent automaticky.

Vývoj software velice je složitý proces a není tedy divu, že je přáním každého programátora používat nástroje, které mu práci co nejvíce ulehčí a umožní mu být produktivní. O to všechno IDE usiluje. Proti co největší integraci, unifikaci, zjednodušování a automatizaci stojí ale rozdílné požadavky uživatelů – programátorů. Každá specifická oblast vývoje software vyžaduje použití více či méně odlišných stylů práce, postupů nebo nástrojů a to je třeba vynásobit osobními preferencemi jednotlivých lidí, kteří je používají.

Zajímavý rozdíl je například v přístupu k IDE na Unixu a na MS Windows. Unix lze s trochou nadsázky považovat za jedno velké IDE. Za dlouhou dobu své existence bylo vyvinuto nepřeberné množství programů, které jsou stavebními prvky IDE, přičemž propojení a vzájemná spolupráce programů je základní filozofií Unixu. Unix také vzniknul v době, kdy se pracovalo jen v CLI (command line iterface, tedy příkazová řádka), ale výhody CLI přetrvávají dodnes a osobně si myslím, že ještě dlouho budou. Po sečtení těchto skutečností je zřejmé, že programátor v Unixu si může například kolem editorů jako Emacs nebo Vim vytvořit vlastní IDE.

Filozofie Windows je zcela opačná. Hlavní charakteristikou Windows je, že upřednostňují práci v GUI (grafickém uživatelském rozhraní) nad prací na příkazovém řádku a snaží se poskytnout uživateli co neintuitivnější prostředí pro práci. Další vlastností je, že maximálně zjednodušují požadavky na nutnost konfigurace před začátkem práce. IDE pro Windows je tedy silně ohraničený produkt, který se snaží být ihned po instalaci připraven k práci, zapouzdřuje v sobě všechny potřebné komponenty a co nejméně ovlivňuje hostitelský systém.

Existuje mnohem více prostředí, která se specializují na konkrétní jazyk, než těch, která jsou univerzální – podporují plnohodnotný vývoj ve více programovacích jazycích. Přičemž pouze funkce zvýraznění syntaxe v daném jazyce a možnost ručního vytvoření příkazu pro kompilaci nebo spuštění programu rozhodně není žádná podpora nového jazyka. Je mnoho IDE nebo editorů, které se tváří, že jsou univerzální, ale ve skutečnosti jsou primárně vyvíjeny pro konkrétní jazyk a přidání dalšího je jen teorie.

3.1 Návrh uživatelského rozhraní

Stejně jako vznik software jako celku má svůj životní cyklus, tak i k vzniku uživatelského rozhraní by mělo přistupovat komplexně a inženýrsky. Pojem uživatelské rozhraní se nevztahuje jenom k softwaru, je to dotyková plocha mezi uživatelem a strojem, kde spolu musí komunikovat prostředky, které jim dané rozhraní nabízí. Jakýkoli technický prvek, který umožňuje ovládat nějaký přístroj, nebo kterým nějaké zařízení poskytuje informace člověku, lze považovat za uživatelské rozhraní. Já se dále zaměřím jen na grafické uživatelské rozhraní v softwarovém světě. Za dobu existence software bylo pojmenováno nebo vytvořeno několik přístupů nebo metodik pro tvorbu uživatelského rozhraní.[17]

- UI řízené technologií a funkcionalitou systému – v začátcích výpočetní techniky se uživatelé museli přizpůsobovat možnostem počítačů. Bohužel i dnes vzniká množství projektů, které jsou řízeny naivně a kde je spontánně kladen větší důraz na funkcionalitu než na potřeby uživatelů.
- UI orientované na uživatele – tato metoda řeší nedostatky popsané v prvním bodě. Zde je úkolem návrhu uživatelského rozhraní je umožnit uživateli dosáhnout jeho cílů, a to co nejjednodušším a nejefektivnějším způsobem.
- UI zaměřené na business procesy – v tomto případě je rozhraní navrhováno tak, aby byla produktivita uživatelů, zaměstnanců společnosti, co největší ve vztahu k obchodním cílům společnosti.
- UI schopné se učit – zde je uživatel součástí návrhu uživatelského rozhraní, stroj se na základě interakce s uživatelem učí a přizpůsobuje své rozhraní. [18]

Metody, které k uživatelskému rozhraní přistupují skutečně inženýrsky, jsou velmi komplexní, a proto vyžadují značné množství zdrojů. Ve velkých firmách se to určitě vyplatí, u projektů, které jsou vyvíjeny jedním nebo několika málo lidmi, jsou

propracované metody prakticky nerealizovatelné a je potřeba se uchýlit k výrazně skromnějším postupům.

Bez ohledu na použitou metodu je dobré dodržovat zásad pro návrh dobrého uživatelského rozhraní:[19]

- Konzistence: stejné nebo podobné prvky systému by se měly ovládat, chovat a vypadat stejně nebo podobně. Stejné akce by měly dávat stejný výsledek. Funkce ani pozice ovládacích prvků by se neměla měnit.
- Flexibilita: program by měl umožnovat práci různým skupinám uživatelů, které se liší vědomostmi, dovednostmi, zkušenostmi, zvyky nebo osobními preferencemi aktuálními nebo dlouhodobými.
- Zpětná vazba: systém by měl okamžitě reagovat na příkazy uživatele, a to graficky, textově nebo zvukově. Jsou dva druhy vazby: slabá, kdy počítač zobrazí informaci, ale nevyžaduje ověření, že ji uživatel zachytil. Silná zpětná vazba naopak vyžaduje potvrzení uživatele.
- Navigace: posloupnost úkonů k dosažení nějakého výsledku by měla být co nejkratší. Systém by měl ovládat plynule a přímočaře, měl by minimalizovat nutnost pohybů zrakem a rukama uživatele. Při návrhu by měly být předvídány uživatelovy přání a potřeby.
- Shovívavost: program by měl být odolný vůči chybám uživatele. Nelze se spolehnout, že uživatel dodrží předepsaný postup práce a je třeba s tím počítat. Je třeba rozumně chybovým situacím předcházet. Když už k chybě dojde, je třeba uživatele informovat a pro něj srozumitelně poučit.
- Zotavení: systém by měl umožnit vrátit poslední akce zpět nebo ji zrušit. V případě provádění složitější operace by měl systém umožnit celou akci odvolat. Uživatel by nikdy neměl ztratit data technickou nebo svojí vlastní chybou.
- Předvídatelnost: sled událostí při pracovním procesu by měl být pro uživatele přirozeně předvídatelný. Měly by být používány zřetelné a rozpoznatelné vizuální prvky. Mělo by být jasné, co je výsledkem každé akce, a všechny požadavky by měly být splněny jednotně a úplně.

- Přehlednost: rozhraní by mělo být celkově přehledné, tedy včetně vizuálních, funkčních a textových prvků. Uživatel by neměl být nucen si jednotlivé věci pamatovat nebo se s nimi stále znovu seznamovat.

Tento výčet není konečný, resp. existuje mnoho jiných seznamů doporučení, která jsou obsáhlejší nebo jinak členěná. Je třeba zdůraznit, že to jsou opravdu jen doporučení a ne formalizovaná pravidla. Z toho plyne, že je třeba tvořit spíše v celkovém souladu s těmito radami a ne k nim přistupovat mechanicky.

Z osobní zkušenosti si dovolím tvrdit, že kritický vliv na programátorovy požadavky na IDE mají jeho předchozí zkušenosti a návyky z jiných programů, a to jak z hlediska ovládání, tak i vzhledu. To platí hlavně v prvních okamžicích seznamování s novým software a jeho rozhraním, kdy spolu bojují setrvačnost a touha po novinkách. Na druhé straně emočního spektra je situace, kdy uživateli práce v nějakém programu opravdu silně nevyhovuje, ale není ochoten nebo schopen udělat změnu.

Celkově si tedy myslím, že je zcela legitimní, když existuje mnoho různých IDE, a to, že poskytují, při dodržení správných zásad návrhu GUI, více či méně odlišný přístup k řešení určité problematiky.

V další části je popis IDE, reps. jejich pluginů, které realizují integraci Valgrindu do daného IDE.

3.2 IDE s podporou Valgrindu

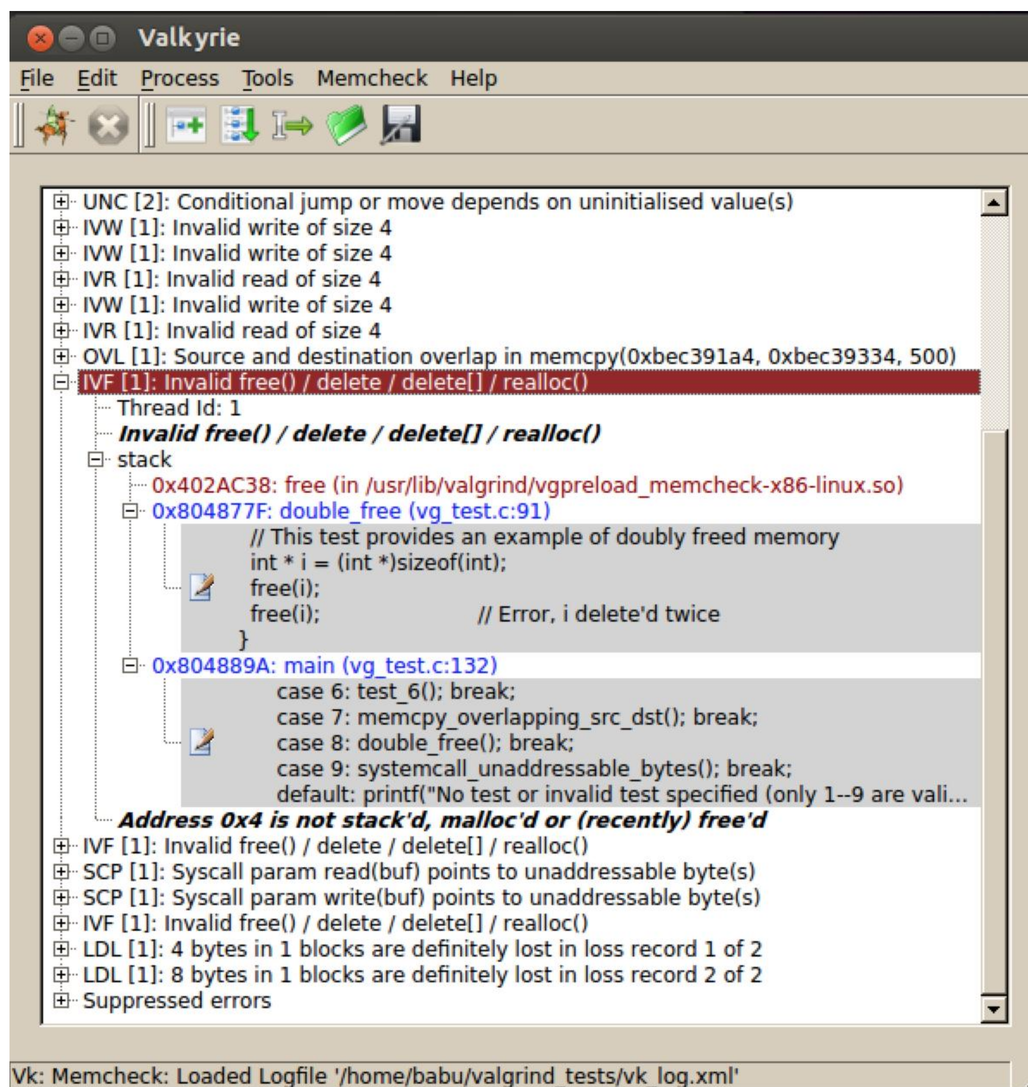
Abych se inspiroval a abych korigoval svoji vizi, vyzkoušel jsem několik programů, jež poskytují grafické rozhraní k Valgrindu. Existují programy, které jsou jenom grafické nástavby, ale pak zde jsou i IDE s vestavěnou podporou Valgrindu. Ty jsem nezkoumal celkově, ale zaměřil jsem se pouze na rozšíření integrující Valgrind.

Kromě programů vyjmenovaných na stránkách Valgrindu jsem našel plugin pro Valgind jenom v Code::Blocks.

3.2.1 Valkyrie

Valkyrie GUI pro Memcheck a Helgrind od vývojářů Valgrindu. Slouží k zobrazení protokolu zmíněných nástrojů. Načítá (případně ukládá) protokol pouze v XML a zobrazuje chyby ve stromové struktuře Obr. 1 Ukázka programu Valkyrie. Uzly první

úrovně jsou chyby, po rozvinutí se objeví další informace a hlavně strom volání. Položky stromu volání obsahují adresu instrukce, jméno funkce a zdrojový soubor. U souboru



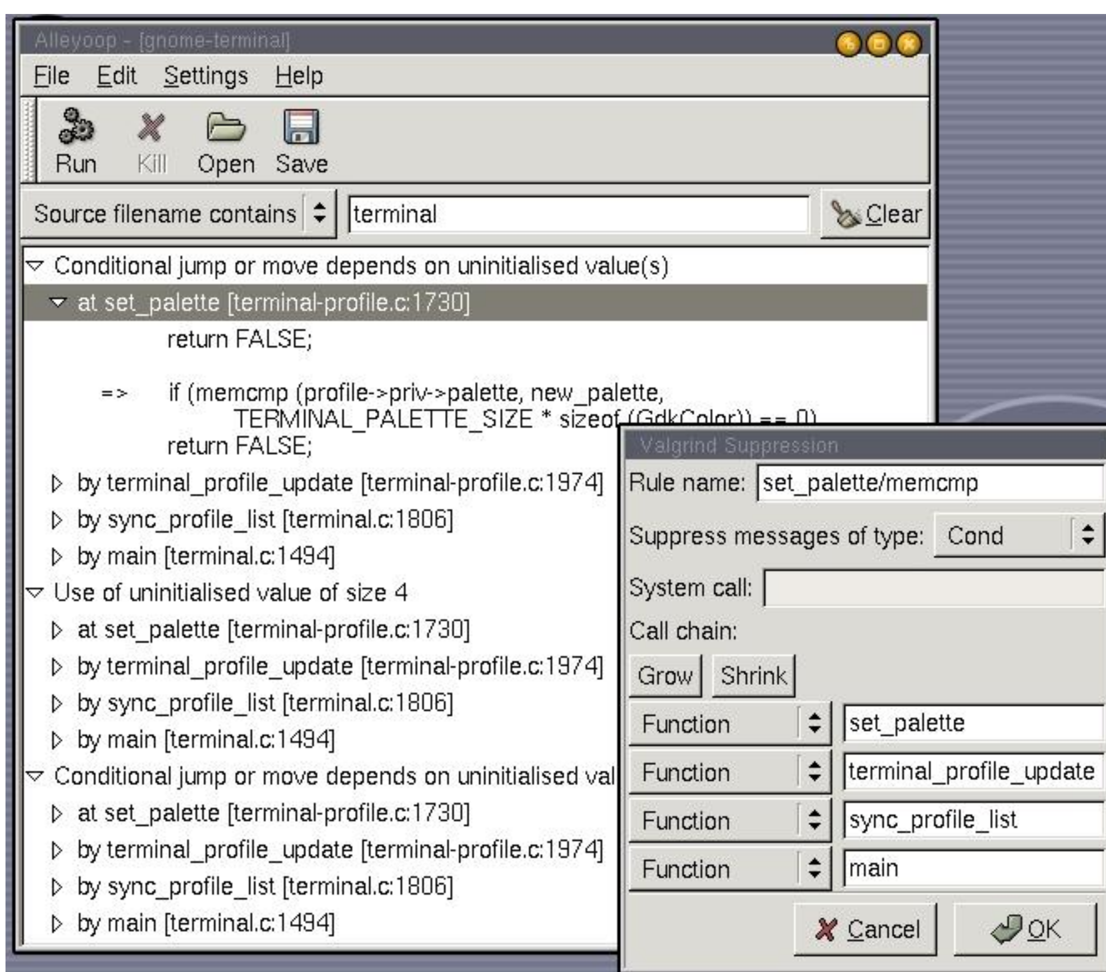
Obr. 1 Ukázka programu Valkyrie[35]

zdrojového kódu je navíc číslo řádku. Červeně jsou označeny položky, k nimž nelze nalézt zdrojový kód, modré položky lze dále rozvinout a objeví se inkriminovaný řádek v souboru s jeho okolím. Na tento kód lze kliknout, po čemž dojde k otevření daného souboru na daném řádku v externím programu. Tím je logicky nejčastěji editor nebo IDE, ve kterém programátor aplikaci vyvíjí. Lze přepínat mezi zobrazením plných cest nebo jenom jmen souborů. Existuje možnost spouštět test Memcheck nebo Helgrind přímo z tohoto GUI, stačí v nastavení vybrat binární program, který bude testován, a zvolit parametry testu pomocí několika přepínačů. Novější verze programu umí spojovat několik logů do jednoho, vývojáři pracují na podpoře nástrojů Cachegrind and Massif.

V nastavení lze vytvořit seznam suppression souborů, které budou použity při testu, jejich vytváření nebo editaci ale aplikace neumožňuje.

3.2.2 Alleyoop

Alleyoop je na první pohled velmi podobný Valkyrie. Ukazuje chyby ve stromové struktuře, umožňuje otevření zdrojového kódu v externím programu a umožňuje spouštět testy a automaticky načítat jejich protokoly. Zvláštní vlastností Alleyoopu, je to, že neumí pracovat s protokolem ve formátu XML, všechny informace tedy načítá jen z textového



Obr. 2 Ukázka programu Alleyoop[37]

protokolu.

Na Obr. 2 Ukázka programu Alleyoop[37] je vidět okno Alleyoopu s otevřeným dialogem pro přidání suppression pravidel. Tento dialog, lze vyvolat pro každou chybu pomocí myši a v podstatě simuluje generování potlačovacích vzorů Valgrindem pomocí přepínače --

`gen-suppressions`. Uživatel dostane k dispozici předvyplněný formulář, kde upravuje jména objektu a funkcí. Výhodou jsou přednastavené typy chyb, bohužel ale nejde vynechávat funkce pomocí „…“. Horší nedostatek je to, že Alleyoop neprovádí mangling (kódování) jmen funkcí pro C++ a nevyužívá vzory generované Valgrindem. Jediná možnost je vypnout demangling, ale v tom případě je protokol výrazně hůře čitelný.

Alleyoop umožňuje filtrovat, tzn. zobrazovat jen některé chybové hlášky pomocí regulárních výrazů.

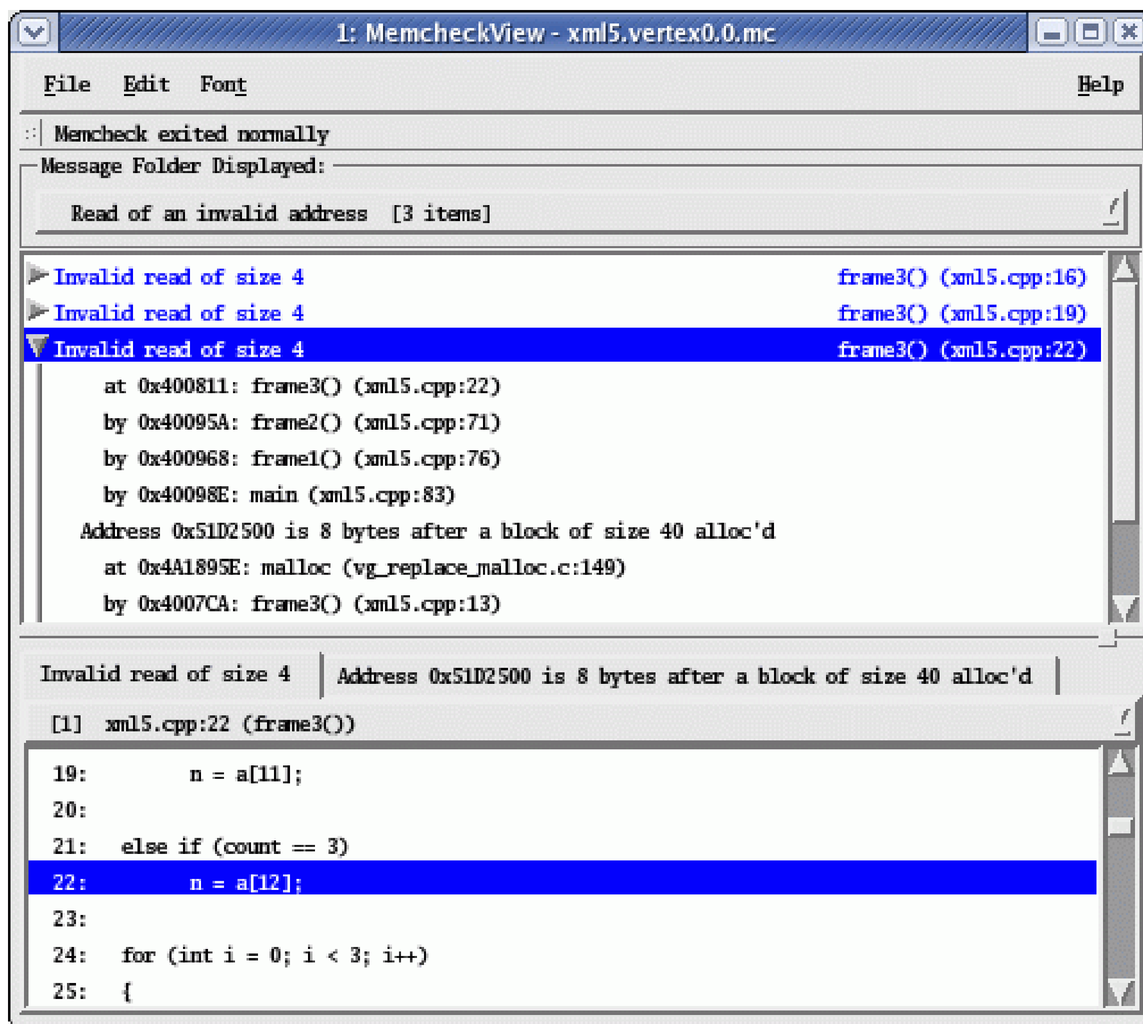
Instaloval jsem Alleyoop ze standardního repozitáře své distribuce, ačkoli jsem se snažil upravit nastavení, nezobrazoval se mi u chyb zdrojový kód. Aplikace vznikla v roce 2003 a asi již není aktivně vyvíjena.[36]

3.2.3 Memcheckview

Memcheckview je program stejného charakteru jako dva předchozí. Obrazovka Memcheckview je horizontálně rozdělena Obr. 3 Ukázka programu MemcheckView. V horní části vidí uživatel chyby a odkazy na zdrojové soubory ve stromové struktuře, je zde dropdown menu, které umožňuje filtrovat zprávy podle typu pomocí. V dolní části je potom náhled zdrojového kódu pro vybranou položku ze stromu. Dolní část je navíc organizována do záložek, obsahuje-li chyba přídavnou část, tak její náhled je umístěn do nové záložky. Memcheckview nemá podporu pro editaci suppression soubory. Náhled zdrojového kódu nejde otevřít v externím programu.

Neobvyklou vlastností Memcheckview je, že obsahuje skripty, které umožňují spouštět Valgrind na paralelních programech napsaných pomocí knihovny MPI.

Program Memcheckview vznikl v roce 2002 a je založen na knihovně QT3. Ačkoli jeho poslední aktualizace byla vydána 31. srpna 2012, tak bohužel není portován na aktuální verzi QT.[38]

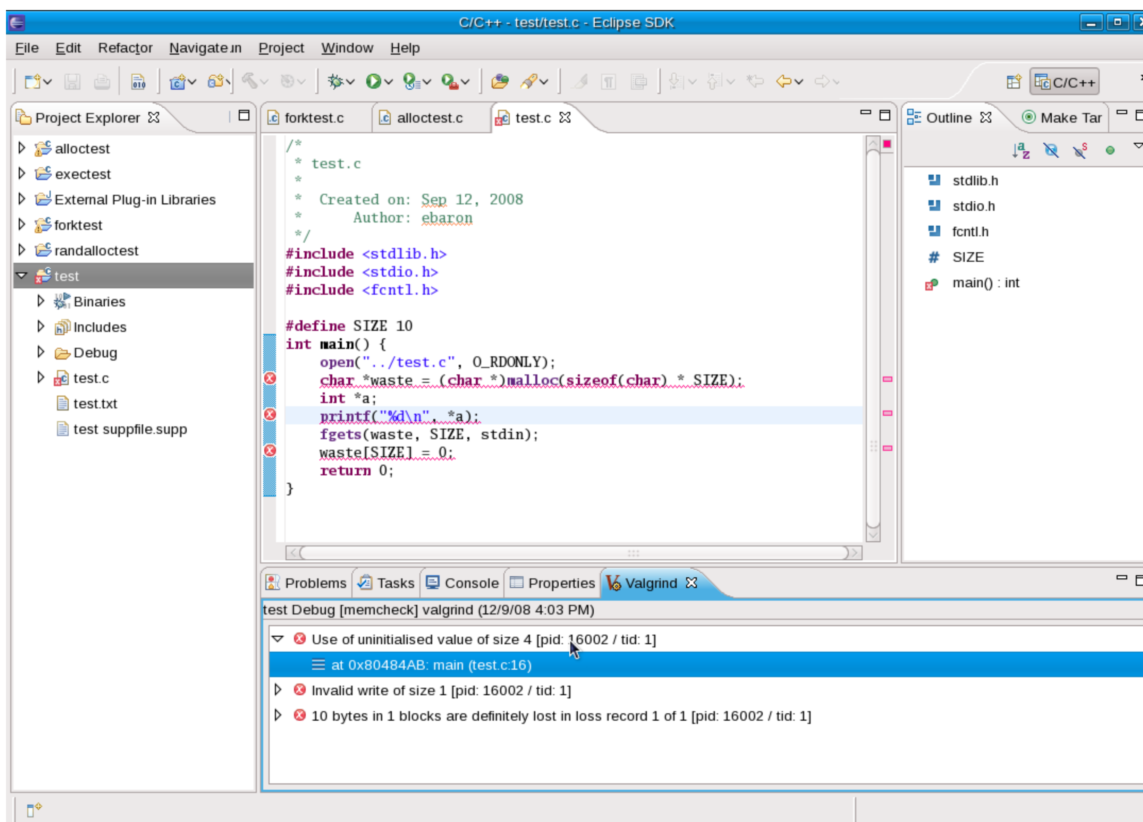


Obr. 3 Ukázka programu MemcheckView[39]

3.2.4 Eclipse

Eclipse je jedno z nejkomplexnějších IDE vůbec, vlastně se jedná o platformu, do které je možné umístit mnoho IDE, které sdílejí styl uživatelského rozhraní. Je to ideální řešení pro ty, kdo pracují ve více jazycích a nemusejí měnit svoje návyky a styl práce. V rámci jednoho pluginu jde do Eclipse přidat celé prostředí pro vývoj aplikací například v Pythonu LaTexu, C++, Javě nebo dalších. Valgrind je v Eclipse dostupný v rámci rozšíření „Linux Tools Project“, což je ve skutečnosti celá sada pluginů, podporovány jsou nástroje Memcheck, Massif a Cachegrind.

Memcheck se v Eclipse spouští univerzální funkcí pro spouštění profilačních nástrojů, je to stejné jako spuštění kompilace. Po skončení analýzy se protokol načte do okna, které je určeno pro výstup pluginů. Toto okno je nejčastěji pod editorem zdrojového kódu, ale



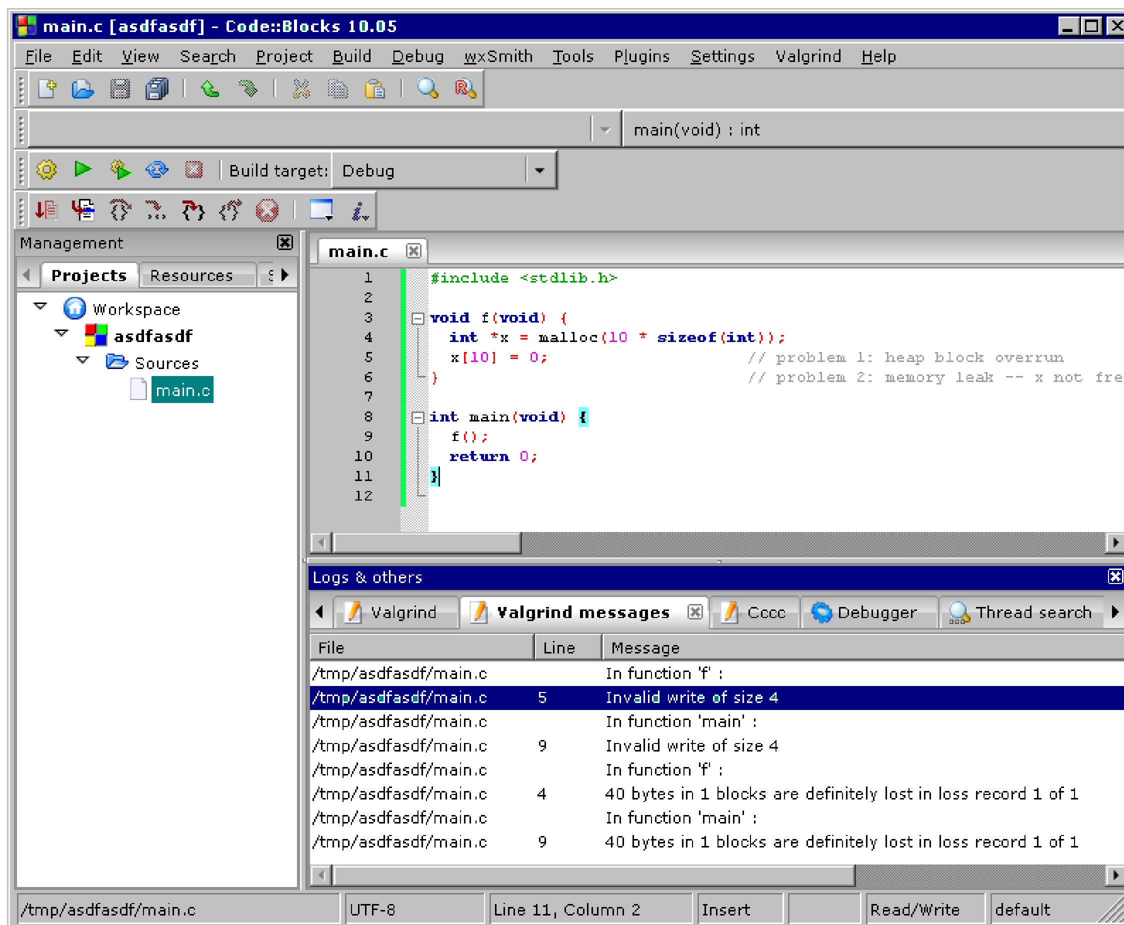
Obr. 4 Ukázka pluginu pro Valgrind v Eclipse IDE

v rámci Eclipsu lze toto okno přesunout i na jiné místo pracovní plochy. Chyby jsou zobrazeny v jednoduché stromové struktuře, podobně jako u předchozích programů, tak jak je organizuje Valgrind. Jednotlivé položky stromu volání reagují na dvojklik otevřením v editoru na příslušném řádku. V editoru jsou navíc chybné řádky označeny symbolem chyby. Obr. 4 Ukázka pluginu pro Valgrind v Eclipse IDE

V nastavení lze měnit parametry pro Valgrind a měnit tak parametry testu, podobně jako u Valkyrie a dalších programů. Tento plugin také přidává do Eclipse schopnost rozpoznat suppresion soubor, což znamená zvýraznění syntaxe, folding (zmenšování syntaktického bloku do jednoho řádku) a automatické doplňování několika klíčových slov.[40]

3.2.5 Code::Blocks

Code::Blocks je přímý konkurent CodeLite na poli IDE zaměřených na jazyky C/C++. Pracovní plocha je organizována jako ve většině programovacích IDE, pod oknem editoru je užší okno se záložkami, kde je možné pracovat s jednotlivými pluginy. V Code::Blocks jsou zde dvě záložky, jedna slouží jako konzole pro výstup Valgrindu z příkazové řádky,



Obr. 5 Ukázka IDE Code::Blocks s pluginem pro Valgrind

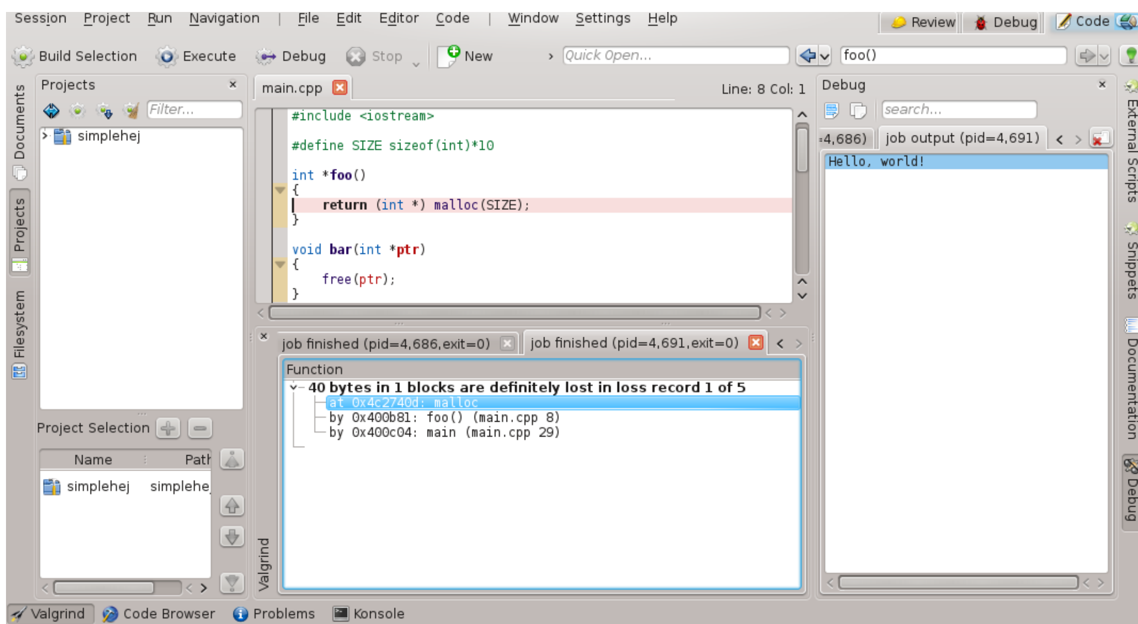
druhá pro prezentaci nalezených chyb. Chyby jsou zde zobrazeny naprosto odlišným způsobem než v ostatních programech. Není zde ukázána žádná stromová struktura. Pro každý soubor otevřeného projektu jsou v tabulce vypsány dva řádky. Na prvním je jméno funkce a na druhém titulek chyby, na obou je pak číslo řádku. Tabulka má tři sloupce, jedná se o jméno souboru, číslo řádku a titulek, kde se zobrazuje jméno funkce nebo jméno chyby. Ve výpisu nejsou žádné záznamy, které programátor nemůže změnit. Protože položky nejsou organizovány do stromu a titulek chyby se stále opakuje, není tak zřejmé, jak spolu záznamy souvisejí. Po dvojkliku se otevře daná lokace v editoru. Obr. 5 Ukázka IDE Code::Blocks s pluginem pro Valgrind

Tento plugin umožňuje spustit test pro právě aktivní projekt, neumožňuje však provést vůbec žádná nastavení testu. Je zde navíc možnost spustit test Cachegrind, jeho výsledek je vypsán do první záložky, tak jak ho Valgrind tiskne na příkazovou řádku, ani zde není možnost nastavení. Plugin pro Valgrind nepřidává do Code::Block žádnou podporu pro práci se suppression soubory.[41]

3.2.6 KDevelop

Plugin pro Valgrind přidával do IDE KDevelop podporu pro nástroje Memcheck, Cachegrind, Callgrind, Helgrind a Massif. Jak je vidět na Obr. 6 Ukázka pluginu pro Valgrind v IDE KDevelop, tak vzhled protokolu z Memchecku je skoro stejný jako v Eclipse. Nastavení skýtá podobné možnosti, takže i schopnosti nebo styl práce se nebudou moc lišit.[42]

Ačkoli se zdá, že do tohoto pluginu autoři vložili značné úsilí, už ho nelze použít. Kvůli chybám a zároveň kvůli tomu, že byl vývoj tohoto pluginu ukončen, byl odebrán z KDevelop.[43] Poslední aktualizace proběhla v lednu 2012. Tato verze je stále ke



Obr. 6 Ukázka pluginu pro Valgrind v IDE KDevelop

stažení, ale její binární podoba bohužel nefunguje v současné verzi KDevelop.

4 CODELITE

4.1 Historie

Na začátku roku 2006 se budoucí autor IDE CodeLite, pan Eran Ifrah, zajímal o možnost rozšíření vývojového prostředí Code::Block o novou, lepší funkci automatického doplňování kódu, včetně navigátoru ve zdrojových kódech, hledání symbolů nebo automatické nápovědy při doplňování. Jeho řešení bylo založeno na parsování zdrojových kódů pomocí ctags a ukládání získaných dat v SQLite databázi.[44]

Na podzim téhož roku pan Ifrah postoupil ve své práci natolik, že mohl představit svůj CodeLite, tehdy ještě „jen" jako jádro, které vytváří virtuální reprezentaci zdrojových kódů a umožňuje tak automatické doplňování a jakékoli další operace se symboly ve zdrojových kódech. Součástí tohoto prototypu byl i jednoduchý editor pojmenovaný LiteEditor, který demonstroval všechny schopnosti CodeLite.[45]

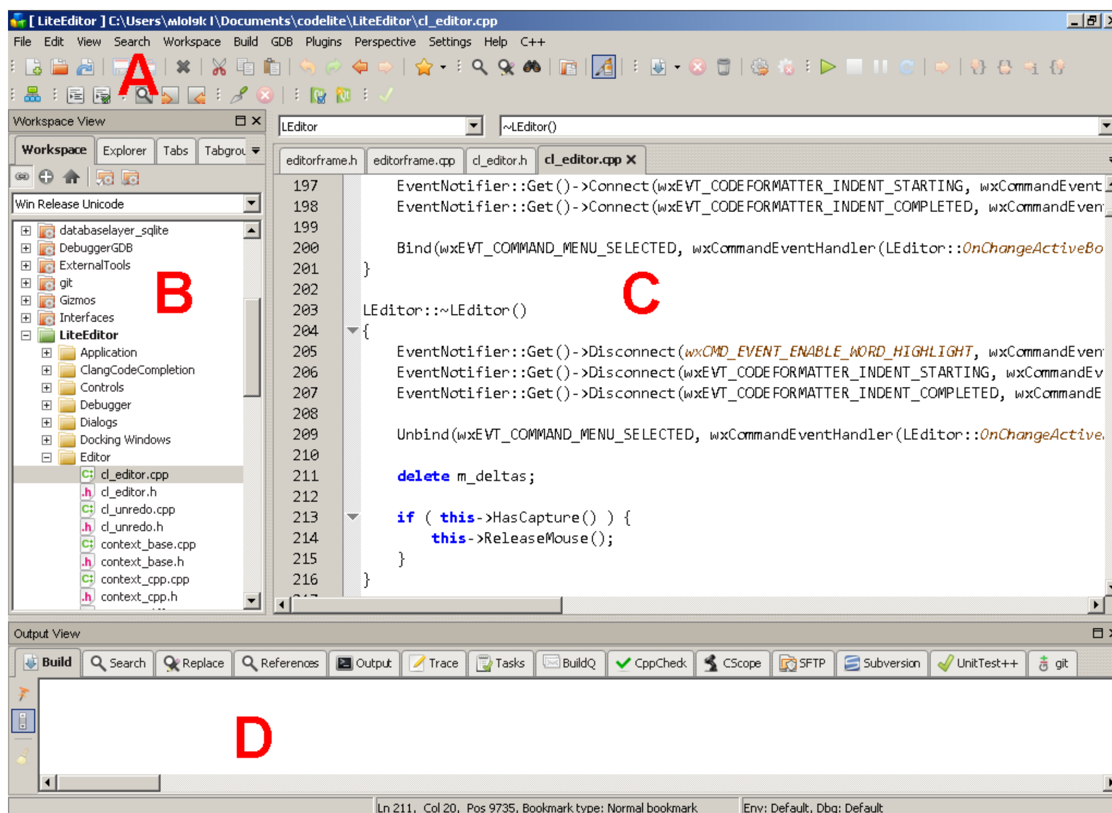
V té době samozřejmě Code::Block už disponovalo všemi funkcemi, které nabízel CodeLite. Možná i proto se nenašel nikdo, kdo by byl ochotný CodeLite portovat jako plugin do Code::Blocks. Koncem roku 2006 diskuze kolem tohoto projektu v rámci Code::Block definitivně utichala a pan Ifrah zvolil jiný směr pro svůj projekt. Pokračoval v práci na CodeLite a ještě více na LiteEditoru, z CodeLite se stalo IDE a LiteEditor jeho součástí. Ačkoli starší verze IDE CodeLite byly průběžně k dispozici už dříve, zmíním významný milník. 1. července 2008 pan Ifrah oznámil vydání verze 1.0 IDE Codelite.[46]

Posledních osm let se na IDE CodeLite takřka denně pracuje a kromě pana Ifraha přispěli a přispívají i další.

4.2 Rozhraní a funkce

CodeLite je vývojové prostředí od počátku dedikované pro programování v C/C++ a tomu jsou přizpůsobeny všechny funkce.[47]

Okno CodeLite lze rozdělit na několik logických sekcí tak, jak jsem to naznačil na obrázku. Jako „A" je označen nástrojový panel, který obsahuje nejčastěji používané funkce, jako tlačítko editaci, hledání, spuštění a zastavení, kompilace, debugování nebo formátování kódu.



Obr. 7 Okno IDE CodeLite verze 5.4 ve Windows (A: toolbar; B: Worspace View; C: editor; D: OutputView)

Sekce „B“ obsahuje několik záložek, například správce workspace, prohlížeč souborového systému nebo Outline (strom symbolu). Jednotlivé záložky z tohoto panelu jde odpojit a přesunout například vpravo od editoru. Ve správci workspace jsou vidět všechny projekty a je možné měnit jejich nastavení nebo přidávat nové soubory nebo spouštět různé akce pro projekty nebo pro jednotlivé soubory. Po dvojkliknutí na jméno souboru je tento otevřen v okně editoru.

Část označení jako „C“ je právě onen editor, zde probíhá samotné editace textu. Je možno mít otevřeno několik souborů najednou, jsou zobrazeny v záložkách. Nad editorem jsou dvě dropdown menu, která slouží pro rychlou navigaci v právě prohlíženém souboru. V pravém je seznam metod pro vlevo vybranou třídu.

Oblast „D“ slouží především jako místo, kam jednotlivé komponenty IDE mohou vypisovat svůj výstup. V neoddělitelných záložkách jsou zde například okna pro výstup kompilátoru nebo spuštěného programu. V dalších záložkách je například okno pro výsledky hledání ve více souborech, statický analyzátor, nebo pro správce verzí. Existuje

několik šikovných klávesových zkratek, které mění vzhled aplikace. Například Ctrl+M skryje všechny panely kromě editoru, nebo lze zasouvat a vysouvat „Output View“ pomocí klávesy F1.

Práce - programování v CodeLite probíhá vždy v rámci nějakého workspace. Workspace obsahuje projekty a projekt je typicky skupina zdrojových kódů, které jsou kompilovány společně, a výstupem této kompilace je spustitelný soubor nebo knihovna. Abstrakce workspace je tu proto, aby související projekty byly spolu svázány a bylo s nimi možno pohodlně pracovat zároveň. Dojde-li k otevření workspace, otevřou se i všechny přidružené projekty. Obdobně funguje zavření workspace. V rámci workspace projekty sdílejí některá obecná nastavení, lze také provést kompilaci celého worskspace zároveň. Indexace souborů pro funkci automatického doplňování kódu také funguje v rámci celého workspace. Pro každý projekt je možné upravit pravidla i celý proces kompilace.

CodeLite je přenositelný mezi Windows, Linuxem a Macem, primárně byl ale vyvíjen ve Windows, což se projevuje v několika jeho vlastnostech. Proces kompilace, debugování a komunikace se spuštěným programem je zabudován do IDE. Ačkoli závisí na externím softwaru, umně to před uživatelem skrývá. Například při spuštění vyvíjeného programu neotevírá ani nezapouzdřuje konzoli, jak by to bylo typické pro Linuxovou aplikaci, nýbrž odchytává výstup programu, prezentuje ho uživateli ve svém okně a zpětně předává uživatelovy příkazy ze svého okna do běžícího programu.

Nastavení parametrů kompilace je také řešeno způsobem, který je běžný ve Windows, tzn. uživatel provádí veškerá nastavení kompilace v rámci GUI pomocí dialogů k tomu speciálně vytvořených. Následně CodeLite z toho nastavení vytváří makefily a spustí kompilátor, který je použije. Makefily pro uživatele nejsou moc zajímavé, protože v nich přímo změny stejně provádět nemůže, resp. může, ale IDE je při první příležitosti opět přepíše na základě svého nastavení. Možnost použití makefile přímo CodeLite samozřejmě také nabízí. V nastavení nějakého projektu lze výchozí sytém vypnout a použít vlastní makefily. Tímto způsobem je řešena kompilace CodeLite v Linuxu a Macu. V současnosti je to konkrétně řešeno pomocí nástroje CMake, ale to je z pohledu tohoto nastavení rozdíl jen v několika řádcích příkazů navíc.

Další vlastností CodeLite je důraz na podporu různých nástrojů pro parsování zdrojových kódů, kompilaci a ladění. Kromě ctags podporuje i clang. Jako kompilátor může být

nastaven VC++, gcc, clang nebo cobra. Základním ladicím programem je dgb a v nadcházející (v okamžiku psaní této práce) verzi 6.0 bude přidána podpora pro lldb debuger.

Kromě mimořádně propracovaného automatického doplňování kódu nebo plně zakomponované debuggeru disponuje CodeLite velkým množstvím různých funkcí pro refactoring. Změna jména proměnné nebo metody funguje přesně ve zvoleném rozsahu a uživatel nemusí mít obavu, že si poškodí nějaké soubory, nebo naopak, že se na nějakou změnu zapomene. Dalšími funkcemi jsou např. automatické generování settru a gettru nebo změna signatury metody podle hlavičkového, resp. zdrojového souboru.

Další funkce jsou realizovány jako pluginy (rozšíření):

- CodeFormatter – upravuje odsazení, zalamování řádku a další atributy formátování zdrojových kódů
- CppChecker – statický analyzátor
- CScope – vyhledávání a navigace ve zdrojových kódech
- CallGraph – vytváří grafickou reprezentaci volání funkcí v projektu
- DatabaseExplorer – umožňuje pracovat s daty a strukturou databází pomocí SQL dotazů a vizualizovat databáze pomocí ERD diagramů.
- ExternalTools – přidává do panelu spouštěče pro externí programy
- CMake – exportuje nastavení kompilátoru do formátu cmake
- Git – umožňuje verzování projektu přímo v Codelite pomoci systému git
- subversion – umožňuje verzování projektu přímo v Codelite pomoci systému svn
- wxCrafter – plně zabudovaný RAD nástroj pro tvorbu programů s GUI založených na wxWidgets
- wxFormBuilder – alternativní RAD nástroj pro tvorbu programů s GUI založených na wxWidgets

4.3 wxWidgets

CodeLite je plně založený na knihovně wxWidgets, a proto všechno, co platí pro wxWidgets, platí také pro CodeLite. Jediná drobná výjimka je, že autoři CodeLite preferují STL kontejnery před těmi z wxWidgets, přičemž kontejnery ve wxWidgets často jen zapouzdřují ty z STL.

WxWidgets je knihovna nejenom základních komponent pro tvorbu grafického uživatelského rozhraní, obsahuje i řadu dalších tříd pro práci například se soubory, regulárními výrazy nebo sockety. Je to tedy plnohodnotná, mnohoúčelová knihovna, pomocí níž lze vytvořit širokou škálu programů bez nutnosti dalších závislostí.[5][8]

WxWidgets jsou přenositelné mezi MS Windows, Linuxem a Macem, přičemž na UNIXech jsou portovány na víc než jednu grafickou subplatformu. Toto je nejvýznamnější charakteristická vlastnost wxWidgets. Na každé platformě využívá nativních grafických prvků daného prostředí. Díky tomu tentýž program bude vypadat více či méně odlišně na základě toho, na jaké platformě je zkompilován a spuštěn. Rozmístění prvků a funkčnost bude přirozeně zachována, ale aplikace bude mít jiný styl, jiný kabát. Protikladem toho přístupu je například toolkit (grafická knihovna) Qt. Její výchozí strategií je, že simuluje všechny grafické prvky, sama je vykresluje, díky čemuž jeden program vypadá na všech platformách stejně a zároveň se ale na každé bude méně nebo více odlišovat od vzhledu okolního prostředí. Výhodou řešení wxWidgets je někdy až dramaticky větší rychlost odezvy aplikace, nevýhodou zas náročnější implementace samotné knihovny a zároveň větší nároky na schopnosti programátora, který ji používá. WxWidgets musí totiž nabízet jakési kompromisní ovládací rozhraní někdy velmi rozdílných objektů. Určité objekty, které jsou podobné ve svém vzhledu a účelu, se mohou velmi lišit v ovládacím rozhraní.

WxWidgets jsou technicky wrapper (obal) jiné nativní (původní, pro danou platformu přirozené) grafické knihovny. Programátor komunikuje s wxWidgets a wxWidgets s další knihovnou. Tento popis se týká především grafických komponent, jsou samozřejmě třídy, které jsou implementovány pomocí standardních prostředků jazyka C++ nezávisle na platformě a nic neobalují. Důsledkem této zprostředkované komunikace se programátor občas dostává do situace, kdy nemá zcela plnou kontrolu nad nativním prvkem. Programátor nemůže reagovat na změnu stavu určité komponenty, protože wxWidgets tento stav nemůže zjistit a předat ho dál. Týká se to například některých událostí myši nebo

klávesnice, které proudí přímo do nativního prvku, a wxWidgets s tím nemůže nic dělat. Programátor si musí být vědom těchto úskalí a už při návrhu volit dostatečně obecná řešení, která je možné ve wxWidgets dobře realizovat, to ovšem vyžaduje znalosti a zkušenost.

Programování pomocí wxWidgets je založené na objektově orientované programování s posíláním zpráv. Programátor navrhne objektovou strukturu aplikace a poté musí definovat chování callbacků (obslužných metod). O hlavní cyklus aplikace se stará wxWidgets a volány příslušné obslužné metody. Programování tedy spočívá v definování například nějaké akce, která bude vyvolána po stisknutí tlačítka. To, co je v tomto systému důležité, je, že GUI „plyne" jen v jednom vlákně, z toho plyne, že události a příslušné obslužné funkce jsou volány synchronně, řetězí se za sebe a jsou zpracovávány postupně. Důsledky toho jsou například následující: každá obslužná funkce blokuje celé GUI, takže je více než vhodné, aby byla co nejrychlejší, a pokud k tomu není vysloveně důvod, aby neprováděla dlouhotrvající výpočty. Dalším důsledkem například je, že některé akce se projeví až po skončení obslužné funkce. Je třeba předat řízení zpět hlavnímu cyklu, aby provedl, co je třeba. Programátor tedy musí konat v souladu s vnitřním mechanismem posílání a zpracovávání zpráv, který je implementovaný ve wxWidgets.

WxWidgets vzniklo v době, kdy C++ neobsahovalo mechanizmus výjimek, proto je i CodeLite psané stylem dopředného testování stavů, a ne ošetřování stavu pro provedení akce.

4.4 Rozšíření pomocí pluginu

Jak už jsem napsal, velká část funkčnosti CodeLite je implementována jako pluginy. CodeLite je proto přizpůsoben, aby mohl být rozšířen, a obsahuje implementační rozšíření pro pluginy. Plugin tedy může v rámci CodeLite provádět to, co mu toto rozhraní umožní. Existují kořenové oblasti, které nejsou pluginům přístupné, je to například kompilační proces, ale i mnohé další.

Dokumentace CodeLite obsahuje jednoduchý návod, jak založit nový prázdný plugin, ostatní informace existují v kódu jako doxykomentáře. Aby programátor zjistil, jaké má možnosti, musí postupně pročítat zdrojové kódy CodeLite. Zjistí tak, jak zmíněné rozhraní

po pluginy vypadá, a tedy kam plugin smí zasahovat a co je implementováné pouze v rámci základní funkčnosti CodeLite.

Nejlepším způsobem jak vytvořit nový prázdný plugin je použít v CodeLite průvodce vytvořením nového pluginu. Je třeba otevřít workspace CodeLite v CodeLite a spustit průvodce. Ten automaticky vytvoří zdrojový kód implementující minimální rozhraní nutného pro plugin, a hlavně provede nutné nastavení workspace, kdy přidá odkazy na nový projekt, kam je třeba, a vyplní parametry pro správnou kompilaci pro prostředí MS Windows. To, aby se plugin kompiloval i v UNIXech, je třeba provést ručně. Ve starších verzích CodeLite se používal klasický skript „./configure“. Poměrně nedávno se začal v CodeLite používat pro kompilaci v UNIXu cmake, takže místo úpravy a přidání potřebných referencí do configure bylo třeba vytvořit cmake nastavení v souboru CMakeLists.txt. Na počátku tohoto roku byla díky panu Ifrahovi do cmake přidána podpora pro CodeLite, díky čemuž cmake umí potřebné nastavení udělat sám. Od verze 6 CodeLite stačí do CMakeLists.txt napsat jediný řádek „CL_PLUGIN(MyPluginName)“.

Programové rozhraní CodeLite pro pluginy je poměrně rozsáhlé. Pluginy mají možnost pracovat nejen s hlavními prvky prostředí, což je vlastně nutnost, CodeLite pluginů sdílí také obecnou funkcionalitu, kterou implementuje, jedná se například o hledání a nahrazování v textovém řetězci.

Zmíním se krátce o některých třídách rozhraní, se kterými jsem pracoval:

- IPlugin – třída, která implementuje toto rozhraní, reprezentuje samotný plugin. Jsou zde metody, které slouží pluginu pro přidání svých položek do hlavního panelu nástrojů a do všech možných popoup menu v celém CodeLite. Chce-li plugin mít svoji záložku v oblasti „WorkspaceView“ nebo v „Output View“, musí to udělat v konstruktoru a použít k tomu rozhraní IManager. Reference na IManager je již v třídě IPlugin.

- IManager – mager je vlastně jakousi branou do CodeLite, kromě dvou výše zmíněných obsahuje reference na celou aplikaci, na workspace, na editor, na ostatní pluginy a další. Obsahuje také některé užitečné funkce, například test, zda CodeLite zrovna provádí kompilaci nebo uložení všech změn v otevřených souborech.

- IEditor – editor je okno se otevřeným souborem, kde uživatel provádí editaci. Toho rozhraní umožňuje měnit soubor otevřený v daném editoru, dále může například

vyvolat okno s automatickým doplňováním kódu nebo označit část textu. Zároveň poskytuje referenci na wxStyledTextCtrl, který CodeLite v jádře používá. To dává programátorovi možnost pracovat s editorem na ještě hlubší úrovni.

- clConfig a clConfigItem – třída clConfig reprezentuje persistentní souborové uložiště pro nastavení. Toto úložiště je ve formátu JSON a obsahuje stromovou strukturu objektů clConfigItem, přičemž ten může obsahovat libovolné množství atributů základních typů jako čísla, pravdivostní hodnoty, řetězce znaků, seznamů řetězců znaků a ještě několika dalších. Objekt třídy clConfigItem musí implementovat serializaci svých atributů z a do třídy JSONElement. Ta je použita při samotném načítání a ukládání dat z a do souboru.
- Plugins events – jeden halvičkový soubor obsahuje seznam všemožných událostí, ke kterým v CodeLite dochází a k jejichž odběru se plugin může přihlásit. Jedná se například o události jako otevření workspace nebo dokončení startování CodeLite.
- Workspace – toto rozhraní umožňuje spravovat aktivní workspace, tzn. otevírat a zavírat workspace, vyhledávat, přidávat nebo mazat projekty nebo zjišťovat jejich nastavení nebo cesty k souborům. Je zde i metoda pro zjištění právě aktivního projektu. V celém workspace je vždy právě jeden takový. CodeLite nabízí v hlavním panelu funkce jako kompilace aktivního projektu, nebo jeho spuštění. Například i vyhledávání lze omezit na právě aktivní projekt.
- FileLogger – pomocí této třídy a souvisejících maker může programátor přidávat do kódu pomocné výpisy, varování, nebo hlášení o chybách, které jsou směřovány do globálního souboru, který sdílí všechny součásti CodeLite. V UNIXu je velmi časté vypisování přímo na standardní výstup nebo standardní chybový výstup. CodeLite i tuto aktivitu zapouzdřuje.
- AsyncExeCmd – tato třída slouží k asynchronnímu spuštění externího programu.

StringFindReplacer – tato třída implementuje co možná nejrychlejší vyhledávání v textovém řetězci, jakým může být například celý soubor. Vyhledávat lze oběma směry, se zohledňováním velikosti znaků nebo bez, podle celých slov (slovo je ohraničeno například bílými znaky) a podle obyčejného řetězce nebo regulárního výrazu.

II. PRAKTICKÁ ČÁST

5 NÁVRH A IMPLEMENTACE PLUGINU

V této kapitole jsem se zaměřil na popis problémů a jejich řešení při relizaci praktické části pluginu, tzn. při návrhu a implementaci. Použil jsem prototypovou metodiku vývoje, kterou jsem však maximálně zjednodušil. Pracoval jsem samostatně a dílčí funkční celky (prototypy) jsem konzultoval především s vedoucím, který měl dostatek času, aby projekt prohlédl a podal mi zpětnou vazbu. Některé otázky, se kterými jsem si nedokázal poradit a považoval jsem je za natolik důležité, jsem také konzultoval s hlavními vývojáři CodeLite (Kromě pana Ifraha se v současné době na vývoji systematicky podílí pan David Hart).

Prototypový postup jsem zvolil hlavně proto, že jsem musel čelit ze svého pohledu značné neviditelnosti. Ačkoli jsem již dříve pracoval s wxWidgets, s objekty, které jsem zkoušel použít pro zobrazování chybového hlášení, jsem neměl zkušenosti žádné. Další neznámá oblast pro mě byla CodeLite a jediný způsob jak se jej naučit bylo číst jeho zdrojové kódy a experimentovat.

Zadání resp. nároky na plugin byly velmi stručné a jasné. Cílem bylo udělat podobný plugin, jako je například v Eclipse, ale navíc přidat podporu pro správu suppression pravidel. Abych tedy mohl navrhnout nějaké řešení, musel jsem nejdříve získat potřebné praktické zkušenosti. Postupoval jsem tedy tak, že jsem určitou dílčí část systému implementoval více způsoby (i třeba jen částečně) a ty horší zahodil.

5.1 Spuštění

Uživatel musí mít možnost spustit diagnostický test přímo z pracovní plochy CodeLite, proto jsem musel tedy najít způsob jak toho dosáhnout. Jako další požadavek jsem si stanovil, že v průběhu testu musí aplikace reagovat stejně, jako by byla spouštěna běžně z CodeLite. Tzn. musí být přesměrován vstup a výstup do a z programu přes CodeLite, což znamená použít k tomu spuštění prostředky CodeLite. Jinými slovy nemohl jsem použít prostředky wxWidgets pro spuštění asynchronního procesu.

CodeLite standardně umožňuje program spustit běžně a v rámci debugeru. Pokusil jsem se využít funkci pro běžné spuštění a upravit je. Spouštění programu v CodeLite je v režii globálního manageru. IManager popsaný výše je manager jen pro Pluginy a neposkytuje stejnou funkčnost. Globální manager je implementován jako singleton a je možné se na něj odkázat pomocí statické metody ManagerST::Get(). Spuštění programu se potom provádí

metodou void ExecuteNoDebug(const wxString &projectName). Program je však spouštěn podle aktuálního nastavení projektu (kam spouštěný program patří) a stávající rozhraní manageru neumožňuje nijaké změny před spuštěním. Abych to vyřešil, vytvořil jsem metodu, která uloží stávající nastavení projektu, změní příkaz pro spuštění (injektuje tam příkaz Valgrindu s argumenty), následně zavolá metodu ExecuteNoDebug a po ukončení běhu programu obnoví původní nastavení. Tento přístup má dvě zásadní nevýhody. Jednak nastavení nemusí být vždy obnoveno, to v případě, že dojde v průběhu spuštění například k pádu celého CodeLite. Druhá nevýhoda tkví v tom, že ManagerST::Get() není obsažen v rozhraní pro pluginy, jinými slovy plugin jej nemá používat. Oba problémy by se daly vyřešit implementací potřebných funkcí přímo v jádře CodeLite.

Pokusil jsem se najít jiné řešení a to jsem nalezl v podobě pluginu ExternalTools. Primárním určením toho pluginu je asynchronní spouštění externích aplikací, například skriptů, které upraví aktuální zdrojový kód, nebo například oblíbeného programu pro správu verzí. V rámci ExternalTools je implementována funkce zachytávání výstupu aplikace a zároveň předávání vstupu.

Postup použití je poměrně jednoduchý, stačí vytvořit objekt reprezentující externí proces, nastavit správné parametry, napojit metodu, která se vykoná po ukončení procesu, a proces spustit. Tuto variantu jsem použil v pluginu.

5.2 Kontejnery pro zobrazení chyb

Největším problémem při realizaci celého pluginu byl výběr vhodné komponenty pro zobrazení chybových hlášení Valgrindu, a to ve dvou situacích, jednak při navigaci po chybách a jejich odstraňování v kódu a při vytváření suppression pravidel.

V první řadě jsem se zaměřil na načtení chyb a jejich zobrazení. Použil jsem nejjednodušší komponentu, a to wxTreeCtrl.

WxTreeCtrl zobrazuje informace ve stromové struktuře, každý řádek obsahuje text a případně podléhající položky mírně odsazené od kraje. Obdobná komponenta je použitá v Eclipse. WxTreeCtrl umožňuje ke každé položce přiřadit tzv. klientská data, to může být teoreticky jakýkoli objekt, který je potomkem třídy wxTreeItemData. Díky těmto přídavným datům jsem mohl ke každému řádku přiřadit celou cestu k souboru a číslo

řádku. Komponenta tedy zobrazovala všechny chyby ve stromové struktuře a po dvojkliknutí provedla v editoru CodeLite navigaci na patřičné místo v souboru.

Dále jsem chtěl umožnit uživateli, aby mohl označit některé chyby a mohl je přidat do suppression seznamu. Pro tento účel se již wxTreeCtrl nehodí, protože nemá žádné možnosti rozšíření.

Existuje kontejner wxTreeListCtrl, který je funkční kombinací wxTreeCtrl and wxListCtrl. Je to komponenta rozdělená do sloupců, stromová struktura je v prvním sloupci, kde navíc může být ještě ikona nebo checkbox. WxTreeListCtrl bohužel nepodporuje virtuální styl, tak jako wxListCtrl, je totiž interně implementován pomocí wxDataViewCtrl.

Podle doporučení vedoucího jsem se tedy soustředil přímo na použití wxDataViewCtrl. Tato komponenta umožňuje obecné zobrazení dat ve stromové nebo sloupcové struktuře. Na rozdíl od dříve popsaných komponent wxDataViewCtrl odděluje prezentační a datovou část své implementace. Sloupec ve wxDataViewCtrl může obsahovat kromě textu, icon i další objekty, například progress bar nebo dropdown menu. Z třídy, která se stará o vykreslování položek ve sloupci určitého typu, může programátor derivovat vlastní třídu a zobrazovat jako položku libovolný objekt. Programátor může ve wxDataViewCtrl implementovat i řazení podle sloupců. Bez ohledu na způsob zobrazení a konkrétní pozici v rámci okna jsou datové položky uloženy ve wxDataViewModel. Když wxDataViewCtrl potřebuje zobrazit nějaké položky, dotazuje se na jejich hodnoty svého modelu. Datová struktura modelu může být interně realizována libovolně, model však musí implementovat virtuální rozhraní wxDataViewModel.

5.3 Vnitřní reprezentace

Již při načítání logu Valgrindu jsem narazil na problém, struktura logu neumožnovala efektivně načítat položky XML a přímo vytvářet položky v wxTreeCtrl. Zároveň jsem chtěl implementovat funkci filtrace chyb, tzn. aby uživatel vyhledal a zobrazil jen určité chyby. Oba problémy jsem vyřešil tak, že jsem vytvořil vnitřní reprezentaci chyb v paměti a až v druhém kroku je přesunul do GUI komponenty. Tento postup je přehlednější a univerzálnější, ale spotřeba paměti je větší. Použil jsem std::list jakožto nejlehčí kontejner z STL. Jedná se jednosměrný spojový seznam bez možnosti přístupu k libovolnému prvku v konstantním čase.

Ve chvíli, kdy jsem musel implementovat model pro wxDataViewModel, uvažoval jsem o sloučení seznamu chyb a modelu. Druhá alternativa byla použití univerzálního modelu, který je generován wxCrafterem.

Teoreticky sice wxDataViewCtrl odděluje data a jejich zobrazení, ale chce-li programátor například změnit barvu písma v určitém řádku, musí k modelu přiimplementovat tu část rozhraní, která informuje wxDataViewCtrl o stylu dat. Model generovaný wxCrafterem tuto část rozhraní neimplementuje.

Model generovaný wxCrafterem vytváří stromovou strukturu svých dat, to je velmi výhodné, protože právě stromovou strukturu jsem potřeboval ve wxDataViewCtrl zobrazit. Kdybych chtěl uvnitř modelu mít svůj seznam chyb, potřeboval bych další datovou strukturu, která by mapovala strom na seznam. WxDataViewCtrl se totiž dotazuje na jednotlivé prvky v modelu pomocí metod GetParent a GetChildren, jedná se tedy o stromovou abstrakci.

Další výrazný problém je situace, kdy by model měl obsahovat více prvků, než je ve skutečnosti zobrazeno. To pro funkci filtrování. Data v modelu se nezobrazují v wxDataViewCtrl na základě prostého faktu, že je model obsahuje. Při změně dat musí model poslat do wxDataViewCtrl přesný seznam prvků, které byly přidány, smazány nebo změněny, a vynutit tak překreslení wxDataViewCtrl. Tyto funkce model z wxCrafter také neobsahuje, nicméně implementace takové funkčnosti se mi zdála obludně složitá zvlášť ve chvíli, kdy jsem se s wxDataViewCtrl a jeho modelem teprve seznamoval.

WxDataViewCtrl je velmi komplexní komponenta, její velkou nevýhodou je velice nízký výkon. Valgrind generuje až tisíce chyb a každá může obsahovat i desítky odkazů na soubory. WxDataViewCtrl si není schopen s takovým množstvím poradit.

Pan Hart mi doporučil oddělit funkčnost procházení chyb pro účely oprav a zobrazení chyb za účelem jejich přidávání do suppresion souboru. Dále mi pro manipulaci s velkým množstvím položek doporučil použití wxListCtrl se stylem wxLC_VIRTUAL, který zároveň vynucuje styl wxLC_REPORT. Styl wxLC_REPORT znamená řádkový seznam s jedním nebo více sloupci. Zajímavá vlastnost je ale až wxLC_VIRTUAL, díky níž načítá wxListCtrl jen několik položek, takových, které je potřeba aktuálně vykreslit, zároveň ale ukazuje posuvník a umožňuje posouvání podle celkového počtu položek, které je

nastaveno. Aktuální položky jsou načteny, až když je potřeba je zobrazit. Nevýhodou je možnost zobrazení jen jednořádkové informace.

Další rada od vedoucího, kterou jsem využil, byla implementace stránkování pro wxDataViewCtrl. Stránkování znamená, že v určitém okamžiku je v wxDataViewCtrl zobrazena jen malá část záznamů, když chce uživatel vidět další, musí provést navigaci na další virtuální stránku se záznamy.

Po několika experimentech a konzultacích jsem zvolil následující řešení:

- Zpracování reportu do seznamu implementovaného pomocí std::list
- Okno pro procházení chyb implementované pomocí wxDataViewCtrl s modelem z wxCrafteru
- Okno pro správu suppressions pravidel implementované pomocí wxListCtrl

5.4 Ikony

Při své práci jsem také narazil na problém s ikonami, resp. na to, jak je v pluginu používat. Dlouho jsem prohledával jiné pluginy i celý CodeLite. Různé části používají různé způsoby, které jsou často i kombinovány, nebyl jsem proto schopen se zorientovat, jak bych ikony správně použil. Až ve výrazně pozdější fázi práce jsem byl schopen jednotlivé způsoby jednoznačně pojmenovat:

1. Rozhraní IManager poskytuje metody GetStdIcons(), která vrací ukazatel na BitmapLoader. BitmapLoader je vlastně art provider pro CodeLite, poskytuje metodu LoadBitmap, která má jediný argument – řetězec znaků, který identifikuje konkrétní ikonu a hlavně zohledňuje při tom aktuálně aktivní grafické schéma CodeLite. Tato metoda vrací ikonu ve formě, kterou lze dále předat nějakému grafickému prvku ve wxWidgets.
2. Všechny okna pro CodeLite by měla být navrhována pomocí wxCrafteru. Obrázky pro všechny prvky ve wxCrafteru lze zadávat pouze jako absolutní cesty k souborům. Ve verzi, tuším, 1.3, což je v současnosti aktuální verze, byla do wxCrafteru přidána možnost používat wxArtProvider, bohužel CodeLite „art provider“ tu není.

3. WxCrafter umožňuje kromě oken spravovat také wxImageListy. Práce s wxImageListem ve wxCrafteru je stejná jako například s oknem. Programátor vytvoří třídu reprezentující konkrétní množinu ikon, v programu podle této třídy vytvoří objekt a pomocí něho může odkazovat jednotlivé ikony.
4. Všechny ikony, které jsou v CodeLite použity, tzn. jsou načteny do paměti, jsou jakožto zdroje zpřístupněny pomocí rozhraní wxXmlResource. Jedná se o ikony z bodu 1 i 3.

Pro svůj plugin jsem tedy zvolil podle mě kompromisní řešení. Ikony, které jsem nevložil přímo do nějakého okna, jsem umístil do wxImageListů a tyto jsem vytvořil jako privátní atributy třídy pluginu. V kódu používám konstrukci `wxXmlResource::Get()->LoadBitmap(<identifikátor ikony>)`, která vrací požadovaný obrázek ikony.

6 PROGRAMOVÉ ROZHRANÍ PLUGINU

Tato kapitola obsahuje popis finální verze pluginu z pohledu vnitřní architektury a principů fungování. Kromě základních požadavků vyplývajících ze zadání jsem do pluginu přidal další drobné funkce a pro jejich realizaci jsem zároveň implementoval potřebné algoritmické struktury. Plugin jsem se snažil strukturovat tak, aby byly zmíněné prvky na sobě co nejméně závislé a umožnovaly tak danou funkčnost v budoucnu rozvinout nebo naopak odstranit.

6.1 Celkový přehled a princip činnosti

V příloze PŘÍLOHA P IV: Diagram tříd je zachycen statický pohled na strukturu pluginu. Kvůli přehlednosti jsou některé prvky v diagramu vynechány.

Ústřední třída MemCheckPlugin, vytváří všechny GUI objekty svého rozhraní, v souvislosti s tím jsou zde funkce pro spouštění testu a vyvolání okna s nastavením. Toto okno není v diagramu zobrazeno, protože není z hlediska algoritmizace ničím zajímavé. Podobně třída MemCheckSettings obsahuje jen všechny patřičné atributy nastavení. Je odvozená od třídy clConfigItem, což umožňuje persistentní úschovu nastavení pomocí mechanismů CodeLite. Obejkt m_settings je sdílen pro čtení v rámci celého pluginu a jednotlivé části z něj čerpají svoji část nastavení. Dojde-li ke změně nastavení, je aktivována metoda ApplySettings, která následně vynutí patřičné změny ve všech ostatních částech pluginu. Metoda IsReady informuje GUI prvky o tom, že právě neprobíhá test a že uživatel může s pluginem pracovat.

MemCheckOutputView je třída, která reprezentuje okno pluginu. Toto okno je v rámci CodeLite umístěno do záložky v panelu „Output View“. Obsahuje panelu nástrojů a navíc je ještě rozděleno na dvě záložky – jednu pro procházení chyb a druhou pro správu suppression pravidel.

Dominantním prvkem první záložky MemCheckOutputView je objekt třídy wxDataViewCtrl, kde se zobrazují chyby ve stromové struktuře. Schopnosti tohoto okna jsou:

- Zobrazovat chyby ve stromové struktuře, ne všechny najednou, ale postupně po určitém počtu, po virtuálních stránkách.

- Provést navigaci
- Procházet chyby po jedné v obou směrech
- Zobrazit aktuální vybranou
- Označit skupinu chyb
- Zkopírovat do clipboardu jeden záznam, chybu, nebo označenou skupinu chyb
- Přidat jednu chybu, nebo skupinu vybraných chyb do suppression souboru.

Druhá záložka slouží ke správě supression souborů. Hlavním prvkem zde wxListCtrl. Zde může uživatel:

- Vybrat z nabídky a otevřít suppresion soubor
- Filtrovat chyby
- Přidávat chyby do suppresion souboru a tam je případně editovat.

6.1.1 Spuštění testu

Třída ValgrindMemcheckProcessor zpracovává XML soubor protokolu vytvořený nástrojem Memcheck a vytváří seznam chyb. Plugin není přímo závislý na třídě ValgrindMemcheckProcessor. Plugin komunikuje s procesorem protokolu pomocí rozhraní IMemCheckProcessor, díky čemuž je zde možnost implementace procesoru pro jiný nástroj, např. Dr. Memory. Rozhraní IMemCheckProcessor je využito hlavně při spuštění testu a při zpracování protokolu, to zachycuje sekvenční diagram v příloze PŘÍLOHA P V: Sekvenční diagram. Řízení probíhá následovně:

1. uživatel vyvolá spuštění testu pro nějaký projekt
2. plugin metodou GetExecutionCommand získá příkazový řádek pro spuštění Valgrindu se všemi argumenty
3. plugin vytvoří asynchronní proces
4. v tomto procesu je spuštěn Valgrind, ve kterém běží testovaný projekt
5. poté, co je test ukončen, se řízení vrací pluginu
6. který vyvolá zpracování protokolu

7. ValgrindMemcheckProtokol prochází XML soubor a pro každý tag <error> vyvolá metodu ProcessError
8. pro každý tag <frame> vyvolá metodu ProcessLocatioon
9. když jsou chyby načteny v paměti, zadá plugin pomoci metody LoadErrors oknu MemCheckOutputView, aby aktualizovao svůj obsah. Třída MemCheckOutputView používá z rozhraní IMemCheckProcessor hlavně metodu GetErrors, která vrací referenci na seznam chyb uložený v paměti.
10. v záložce procházení chyb: metoda ResetItemsView pomocí speciálního iterátoru provede součet chyb, které mohou být podle aktuálního nastavení zobrazeny
11. v záložce procházení chyb: metoda ShowPage zobrazí část chyb ve wxDataViewCtrl
12. v záložce správa suppressions: metoda ResetItemsSupp pomocí speciálního iterátoru provede součet chyb, které mohou být podle aktuálního nastavení zobrazeny
13. v záložce správa suppressions: metoda ApplyFilterSupp zobrazí vybrané chyby v wxListCtrl

6.1.2 Procházení chyb

Navigace k chybě je vykonávána funkcí JumpToLocation, ta jako parametr přijímá položku wxDataViewCtrl. Aby tato metoda mohla bezpečně provést navigaci, potřebuje mít odkaz na MemCheckErrorLocation. Obecně například při přidání do suppression souboru nebo při kopírování chyby do clipboardu musí jednotlivé položky wxDataViewItem referenci na příslušné objekty MemCheckError a MemCheckErrorLocation. Rozhraní modelu umožňuje ke každé položce připojit odkaz na libovolný objekt, který dědí od wxClientData. Problémem je bohužel to, že model generovaný wxCrafterem při vyprázdnění uvolňuje pamět, která je takto referencovaná. Obsah wxDataViewCtrl a tím i jeho modelu se ale v mém pluginu vyprazdňuje často, když uživatel chce zobrazit další stránku chyb. Vzhledem k tomu, že jsem se rozhodl nevytvářet vlastní model, použil jsem pro referencování malou obalovací třídu MemCheckErrorReferrer (resp. MemCheckErrorLocationReferrer), která dědí od wxClientData. Tyto třídy jsou dynamicky alokované na haldě při přidávání položky do wxDataViewCtrl a přijímají jako argument referenci na MemCheckError nebo MemCheckErrorLocation. Třídy Referrer poskytují jedinou metodu Get, která vrací uloženou referenci. Ve chvíli, kdy model maže klientská

data, dochází pouze ke smazání obalovacích objektů a seznam chyb v paměti je nedotčen. Toto řešení není výhodné ani z hlediska časové, ani prostorové složitosti, obecně by bylo lepší model alespoň upravit, nebo ještě lépe vytvořit dedikovaný model.

Plugin umožnuje přecházení po jednotlivých položkách wxDataViewCtrl v obou směrech, tzn. přechod na další nebo následující. Situace je komplikovaná jen tím, že je nutné přecházet na sousední řádek ve vertikálním zobrazení stromu. Přechod na následující řádek tedy může vyvolat opuštění aktuálního stromu chyby a přechod do jiného. Pro implementaci této funkčnosti jsem vytvořil několik pomocných funkcí, které pomáhají navigovat mezi stromy položek wxDataViewCtrl.

Pohyb po položkách jsem uvedl jako příklad, wxDataViewCtrl je komplikovaná třída a odladění aplikace, která ji používá, je náročné.

Podobně jsem čelil například problému s třídou wxListCtrl, která nemá implementovaný mechanizmus pro zobrazování tooltipu (bublinové nápovědy) pro jednotlivé položky. WxWidgets nedisponuje třídou, která by byla schopna nahradit nativní tooltip objektu a zachovat přirozené chování. CodeLite sice implementuje vlastní tooltip pro nápovědu k automatickému doplňování, ale bohužel není dostatečně obecný pro zobrazení informací, které jsem potřeboval. Zvažoval jsem implementaci vlastního řešení, ale rozsah práce byl příliš velký. Nakonec se mi podařilo vytvořit řešení, které používá nativní tooltip a je založené na sledování události pohybu myši po wxListCtrl. Při změne pohybu zjistím, zda je kurzor už na jiné položce, a vynutím vyvolání tooltipu s jiným obsahem.

Při zobrazování chyb v pluginu v grafických kontejnerech, ať už se jedná o stránkování v wxDataViewCtrl, nebo filtraci v wxTextCtrl, je třeba vždy projít seznam chyb uložený v paměti. K procházení STL paměťových struktur jsou často používány iterátory, v případě std::list je to dokonce jediná možnost. Pro procházení seznamu chyb jsem implementoval vlastní iterátor, který podle zadaných parametrů dokáže sám filtrovat některé chyby nebo záznamy stromu volání. Tvorba vlastního iterátoru v C++ je poměrně komplikovaná, v případě tohoto pluginu to ještě složitější, protože seznam chyb je heterogenní stromová struktura. Iterátor, který jsem navrhl, dokáže udržovat kontext průchodu přes celou tuto strukturu. Díky tomu je v pluginu možné skrývat

- Odkazy na soubory, které nepatří do aktuálního workspace

- Duplicitní chyby, které následují těsně po sobě. Nedochází zde k řazení chyb porovnávání a hledání celkové unikátnosti
- Potlačené chyby. Přidá-li uživatel chybu, resp. její vzor, do suppression souboru, ta je okamžitě odstraněna z pohledu. Bez této volby by se potlačení projevilo až po proběhnutí dalšího testu.

Implementace toho iterátoru je mírně složitější, když pomocí vzoru factory vytváří pro iteraci pomocný objekt, který obsahuje pomocné proměnné (udržuje kontext). Výhodou je rozšiřitelnost a oddělení této funkčnosti od ostatního kódu. Kód, který zajišťuje stránkování, je komplikovaný dost sám o sobě.

6.1.3 Potlačování chyb

Oba panely (s wxDataViewCtrl i s wxListCtrl) umožňují vybrané chyby přidat do suppression souboru. Pro efektivní vyhledávání a vybírání chyb k potlačení je určen druhý panel (ten s wxListCtrl). Zde je textové pole, které umožňuje fultextové vyhledávání v chybách buď pomocí prostého řetězce, nebo pomocí regulárního výrazu. Nastavení vybraného filtru provádí funkce ApplyFilterSupp, která aktualizuje obsah wxListCtrl. Funkce vyhledávání jsou součástí programového rozhraní CodeLite. Implementoval jsem zvláštní druh filtrace, který má usnadnit selekci chyb, jež programátor nemůže ovlivnit. Princip je velmi jednoduchý, speciální filtr vybere všechny ty chyby, které neobsahují žádný odkaz do souboru z aktuálně otevřeného workspace. K tomu účelu obsahuje třída MemCheckError metodu hasPath, která přijímá jako argument cestu k současnému workspace.

Vlastní potlačení zajišťuje metoda SuppressErrors, její parametr mode označuje druh výběru chyb k potlačení. Možnosti jsou:

- Aktuální chyba z wxDataViewCtrl, pro tuto volbu je nastaven druhý parametr metody, u ostatních způsobu není druhý parametr použit.
- Označené chyby z wxDataViewCtrl
- Označené chyby z wxListCtrl
- Všechny chyby z exListCtrl

Přidání chyby otevře vybraný suppression soubor v editoru a vzory vybraných chyb do něj přidá. Pro tyto chyby metoda nastaví atribut suppressed v seznamu chyb. Programátor vzory může případně ručně upravit. Funkci, která by tento úkon zrušila a vrátila soubor a plugin do stavu před potlačením chyb, jsem bohužel neimplementoval. Myslím si, že synchronizovat změny textového souboru se seznamem chyb v paměti není efektivně možné. Jestliže uživatel soubor upraví, plugin nemá šanci zpětně poznat, které řádky změnil sám a ke které chybě vzor ze souboru patří.

Je-li aktivní volba skrývání potlačených chyb a zároveň dojde-li k potlačení nějakých chyb, stane se zobrazení v obou oknech neplatné. Je třeba znovu spočítat chyby a provést nové zobrazení. Zobrazení chyb v wxTextCtrl je velmi rychlé, naopak načtení chyb do wxDataViewCtrl velmi pomalé. Z toho důvodu jsem implementoval jednoduchý mechanismus opožděné aktualizace. Pracuje-li programátor v druhém okně, první okno se neaktualizuje, k jeho aktualizaci dojde, až když do něj programátor přepne. Proměnná itemsInvalidView označuje neplatné první okno, itemsInvalidView označuje druhé okno. Aktualizace okna probíhá pomocí metod ResetItemsView a ShowPage, resp. ResetItemsSupp a ApplyFilterSupp.

6.2 Možnost rozšíření

Jak už jsem napsal, plugin je možné dále rozšířit přidáním dalšího procesoru jiného než pro Valgrind-Memcheck. Postup v takovém případě je následující:

1. Procesor musí implementovat virtuální rozhraní IMemCheckProcessor.
2. Chyby musí ukládat do paměťové struktury tvořené z tříd MemCheckError a MemCheckErrorLocation.
3. Je třeba vytvořit třídu, která bude obsahovat atributy s nastavením pro nový procesor.
4. Déle je nutné vytvořit panel v dialogovém okně nastavení a implementovat grafické rozhraní pro změnu zmíněného nastavení.
5. Posledním úkonem je oznámit pluginu, že má k dispozici další procesor. Je třeba přidat několik řádků kódu do metody ApplySettings, které patří třídě MemCheckPlugin.

Jako konkrétní příklad realizace by měla posloužit implementace třídy ValgrindMemcheckProcessor, která se nachází v souboru valgrindprocessor.h/cpp.

Implemetace nastavení pro tento procesor je součástí celkového nastavení v souboru memchecksettings.h/cpp resp. memchecksettingsdialog.h/cpp.

6.2.1 Rozhraní

IMemCheckProcessor obsahuje tři atributy:

- m_settings je reference na objekt nastavení celého pluginu, procesor používá jen tu část, které se ho týká. Toto přístup by mohl být změněn a procesor by mohl mít odkaz jen na svou část nastavení
- m_outputLogFileName obsahuje absolutní cestu k souboru s protokolem, který byl zpracován metodou Process. Dr. Memory generuje pravidla pro potlačení do zvláštního souboru, z toho důvodu by bylo možná vhodné tento atribut přesunout do třídy konkrétního procesoru.
- m_ErrorList je objekt, který obsahuje všechny zpracované chyby. V současné chvíli je implementován jako std::list

IMemCheckProcessor definuje následující metody:

- konstruktor, který přijímá odkaz na objekt nastavení
- GetErrors – vrací referenci na m_ErrorList, tuto metodu není třeba redefinovat
- GetOutputLogFileName – vrací m_outputLogFileName, ani tuto metodu není třeba redefinovat. Plugin nabízí možnost otevření souboru protokolu v editoru a při tom používá tuto funkci.
- GetSuppressionFiles – tato metoda vrací seznam suppression souborů, které se mají použit při aktuálním spuštění. Seznam obsahuje soubory, které jsou pevně definovány v nastavení. Navíc tato metoda musí umět přidat jména souborů závislá na konkrétním stavu aplikace. Například ValgrindMemcheckProcessor spravuje suppression soubor specifický pro workspace, Tento soubor je uložen v privátním adresáři workspace, jímž je v každém workspace adresář „.codelite“.
- GetExecutionCommand – tato metoda sestaví příkaz, který reprezentuje spuštění testu, a který je pluginem spuštění jako asynchronní proces. Jejím jediným argumentem je příkaz, který CodeLite používá při prostém spuštění. Plugin tedy sám dohledá k vybranému projektu jeho příkaz pro spuštění i cestu, na které bude příkaz spuštěn.

Procesor musí originální příkaz metamorfovat v příkaz testovacího nástroje. Musí vhodně nastavit všechny přepínače, zde je vhodné použít metodu GetSuppressionFiles. Důležitým úkonem je specifikace výstupního souboru v rámci příkazu, zároveň je nutné jeho plnou cestu uložit do proměnné m_outputLogFileName pro další použití.

- Process – hlavní metoda, která provádí samotné zpracování souboru s hlášením a převádí ho do interní struktury v paměti. Není-li zadán parametr outputLogFileName, metoda předpokládá, že jejímu volání předcházelo spuštění testu a tedy volání metody GetExecutionCommand, a proto hledá cestu k souboru v proměnné m_outputLogFileName. Je-li jméno zadané parametrem, znamená to načtení dříve uloženého protokolu bez těsně předcházejícího spuštění testu. Tato metoda je časově náročná, a proto by měla buď být neblokující, nebo informovat uživatele prostřednictvím GUI, že je aplikace zaneprázdněna.

6.2.2 Vnitřní reprezentace

Objekt má každá chyba titulek, kde je jméno chyby, typ nebo nějaký obecný člověkem pochopitelný identifikátor chyby. Druhou části chyby je stactkrace, neboli posloupnost volání chyb až do místa, kde došlo k chybě. Každá položka stacktrace, obsahuje jméno funkce, jméno souboru a číslo řádku v souboru.

Valgrind (konkrétně nástroj Memcheck) i další profilery k chybě mohou přidávat jeden nebo více přídavných záznamů. Takový záznam má stejnou strukturu jako chyba sama. Může například nastat situace, kdy chyba má jen titulek a nemá stacktrace, zato obsahuje dvě vnořené chyby, které mají titulek i stacktrace. Součástí kořenové – vrchní chyby by měl být vzor pro generování potlačovacího pravidla. Všechny nástroje bojují s problémem, že nacházejí záplavu chyb, se kterými programátor nemůže nic dělat, a proto zavádějí systém jak takové chování. Vzory pro potlačení vytváří černou listinu chyb, které se neobjeví ve výpisu.

Při zpracovávání protokolu musí procesor do std::list přidávat objekty MemCheckError. Každý MemCheckError obsahuje tyto atributy:

- type – v současné chvíli plugin rozlišuje dva typy, a to základní chybu a vnořenou
- suppressed – tento příznak používá GUI při správě suppressions. Pokud uživatel pomocí pluginu přidá aktuální chybu do suppression souboru, nastaví se tento

příznak na true. Plugin je ve výchozí podobě nastaven tak, že tyto chyby nezobrazuje ve výpisu. Při zpracovávání protokolu musí procesor nastavit tento atribut na false u všech chyb.

- label – titulek, tato položka by měla být člověkem čitelným a pochopitelným údajem, protože se zobrazuje v GUI. Číselný kód chyby by uživatele mátl.
- suppression – vzor pro potlačení v textové podobě. Valgrind nebo např Dr. Memory, nebo i další nástroje mají funkci automatického generování vzorů pro potlačení a je velmi dobré je použít. V opačném případě by procesor tento vzor musel generovat sám z informací o chybě, což je značně náročné.

Každý MemCheckErrorLocation obsahuje atributy:

- func – bývá uvedena snad vždy
- file – v případě, že odkaz směřuje pouze do objektového souboru (program nebyl zkompilován s přídavnými informacemi pro debuging), tak tato položka bývá prázdná
- line – v případě, že je file prázdný, pak je nutné tento parametr nastavit na -1. Taková hodnota pro plugin znamená, že nezobrazí žádné číslo.
- obj – jméno objektového souboru, tzn. souboru se zkompilovaným zdrojovým kódem – knihovna, je-li uveden, bude zobrazen

Místo chyby bývá označeno ještě označeno instrukčním ukazatelem (IP), ale tento údaj jsem do atributů nezahrnul, protože jeho využití je velmi řídké. Přidání tohoto atributu nebo případně dalšího není nijak omezeno. Samozřejmě není nutné upravit plugin tak, aby tento údaj i zobrazoval.

Metody objektů pracují s aktuálními atributy, kdyby došlo k jejich rozšíření, musely by se upravit metody adekvátně upravit. Taková úprava by nebyla složitá, protože metody toString nebo toText vracejí celý obsah chyby a všech pořízených součástí jako řetězec. Funkce toString vrací celý obsah v jednom řádku, toText formátuje text do bloku s odsazením, tato podoba je lépe čitelná pro člověka. toString jso v pluginu použity pro fultextové vyhledávání v chybách. Metoda hasPath vrací true v případě, že chyba obsahuje alespoň jeden odkaz na soubor, který leží na dané cestě.

Na objektech MemCheckError a MemCheckErrorLocation, je silně závislá třída MemCheckIterTools, která slouží k jejich procházení. V případně změn v seznamu chyb je třeba patřičně upravit iterátory v MemCheckIterTools.

6.2.3 Třída nastavení

V současné chvíli plugin obsahuje dvě třídy derivované z clConfigItem, které jsou deklarované v souboru memchecksettings.h. Jedná se o třídy MemCheckSettings a ValgrindSettings, přičemž ta je uskladněna jako atribut třídy MemCheckSettings. V úvodu souboru memchecksettings.h jsou kvůli přehlednosti definovány výchozí hodnoty nastavení, které jsou aplikovány v konstruktorech obou tříd.

Novou třídu nastavení je dobré vytvořit podle již hotového vzoru. Je třeba implementovat všechny funkce podle požadavku tříd clConfig a clConfigItem, s ohledem na princip jejižp oužití v CodeLite. Důležité je definovat unikátní řetězcový identifikátor pro novou třídu nastavení.

Třída s nastavením bude použita jen v implementaci metod z IMemCheckProcessor, ostatní části pluginu se o takové nastavení nezajímají.

6.2.4 Dialog pro nastavení

V případě, že nový procesor bude mít nějaké nastavení, předpokládá se, že bude uživateli umožněno toto nastavení měnit pomocí GUI. K dosažení tohoto cíle je třeba ve wxCrafteru vytvořit nový panel v dialogu nastavení a posléze v souboru memchecksettingsdialog.cpp přidat potřebou implementaci. Minimálně je třeba v konstruktoru provést načtení nastavení do GUI a v metodě OnOK provést uložení z GUI.

6.2.5 Registrace nového procesoru

Do kódu metody ApplySettings je třeba přidat obměnu podmínky

```
if (m_settings->GetEngine() == wxT(CONFIG_ITEM_NAME_VALGRIND))
            m_memcheckProcessor = new ValgrindMemcheckProcessor(GetSettings());
```

která znamená, že jestliže je v hlavním panelu natavení vybrán nějaký procesor, který je identifikovaný unikátním řetězcem znaků, plugin musí vytvořit objekt patřičné třídy.

7 UŽIVATELSKÁ DOKUMENTACE

Plugin MemCheck umožňuje spustit v rámci CodeLite profilaci programu nástrojem Valgrind – Memcheck. Dále plugin přidává panel do oblasti „OutputView", kde jsou po testu zobrazeny chyby z protokolu Valgrindu. Odsud je možné chyby přidávat do suppression souborů. Plugin obsahuje jednoduchý dialog pro změnu nastavení.

7.1 Spuštění testu

V příloze PŘÍLOHA P VI: Plugin – Spuštění testu je zobrazeno, z jakých míst je možné spustit test. Kromě tlačítka pro spuštění obsahuje submenu MemCheck možnost načíst protokol z externího souboru nebo vyvolat dialog s nastavením.

- Červeně označené položky spustí test pro aktivní projekt.
- Modře označená varianta vyvolá popup menu pro libovolný projekt z workspace a umožňuje spustit test pro něj.
- Zeleně označená varianta otevírá popup menu editoru, zde dojde ke spuštění projektu, ke kterému patří otevřený zdrojový soubor.

7.2 Procházení chyb

Po dokončení testu nebo po vybrání dříve uloženého protokolu ze souboru dojde k načtení chyb do pracovního okna pluginu, to je zobrazeno v příloze PŘÍLOHA P VII: Plugin – Procházení chyb.

V levé části okna je svislý panel, který obsahuje spouštěče pro tyto funkce:

- Spuštění testu pro aktivní projekt
- Načtení protokolu ze souboru
- Úplné rozbalení všech položek stromu chyb
- Skok na další chybu
- Skok na předchozí chybu
- Vyvolání dialogu pro nastavení
- Otevření protokolu chyb v editoru

Největší plocha okna je využita seznamem chyb. Každá chyba je zobrazena jako malý strom, kořenový prvek (ikona žlutého trojúhelníku) obsahuje popisek chyby a v další úrovni je seznam odkazů na jednotlivé řádky ve zdrojových souborech (ikona šedého čtverce). Nejhlouběji zanořená volání jsou umístěna nejvýše v seznamu. Chyba může navíc obsahovat přídavné informace (ikona modrého kruhu), takové položky mohou mít kromě titulku svůj vlastní seznam chyb organizovaný stejně jako chyba.

Informace v seznamu jsou organizovány do sloupců. První sloupec označuje vybranou chybu, tzn. chybu, která je zobrazena v editoru nad seznamem chyb. Druhý sloupec umožňuje označovat více chyb pro hromadné úkony. Následují sloupce obsahující už jen údaje o chybách. Jména souborů z workspace jsou uvedeny pouze s lokálními cestami vzhledem k umístění workspace.

Po dvojkliknutí na položku ve stromu dojde jejímu rozvinutí (je-li je to možné). Pokud takto vybraná položka obsahuje údaj ve sloupci „file“, pokusí se plugin tento soubor otevřít v editoru na řádku definovaném ve sloupci „line“.

Pravým kliknutím myši na chybu dojde k vyvolání popup menu, kde může uživatel jednu nebo více chyb potlačit nebo zkopírovat jako text do clipboardu (schránky).

Vpravo od seznamu chyb je navigační panel pro stránkování chyb.

7.3 Potlačování chyb

Ačkoli je možné potlačit chybu už při procházení chyb, ke správě potlačovacích pravidel je především určena druhá záložka v okně pluginu, ta je nadepsána štítkem „.supp“ a nachází se úplně vpravo.

Jak je vidět v příloze PŘÍLOHA P VIII: Plugin – Správa potlačovacích pravidel, okno pro potlačování chyb je vertikálně rozděleno na dvě poloviny.

První řádek levé poloviny obsahuje dropdown menu pro výběr suppression souboru, se kterým se bude pracovat a kam se budou přidávat vzory pro potlačení. Níže jsou ovládací prvky vyhledávání a tlačítka pro přidání chyby do suppression souboru. K dolní hraně okna je zarovnán informační pruh, který obsahuje tři číselné položky: počet všech chyb, počet vyfiltrovaných chyb a mezi nimi ještě počet označených chyb.

Pravá polovina obsahuje seznam chyb, který je možné filtrovat právě pomocí ovladačů vlevo. Rozdíl mezi seznamem chyb v první kartě a zde je ten, že grafická komponenta v prvním okně umožňuje lépe graficky zobrazit chyby, ale je výrazně pomalejší a je schopna pracovat jen s relativně malým množstvím chyb. Proto je také v první kartě implementováno stránkování. Komponenta druhé karty je schopna pojmout všechny chyby zároveň, ale zobrazuje je jen jako jednořádkové položky. Zbytek text celé chyby se zobrazí v tooltipu po najetí kurzorem myši na danou položku.

Vyhledávání nebo spíše filtrace chyb probíhá pomocí fulltextového vyhledávání v chybách. Uživatel má k dispozici volby pro rozlišování velikosti, celých slov, nebo použití regulárních výrazů. Výsledek hledání může být navíc invertován, tzn. budou zobrazeny chyby, které neodpovídají filtru.

V pluginu je implementován ještě pomocný mechanismus filtrace chyb podle toho, na které zdrojové kódy odkazují jednotlivé chyby. Účel této funkce je takový, že má programátorovi usnadnit selekci chyb, které mají zdroj v externích knihovnách a ne v jeho kódu. Tento filtr se vybírá z menu, které se zobrazí po kliknutí pravým tlačítkem myši na ikonu lupy u vyhledávacího řádku. Vybere-li programátor tento filtr, zobrazí se jen ty chyby, které v žádném případě nepramení z jeho zdrojového kódu.

Přidat vzor pro potlačení do souboru je možné buď tlačítky pod nástroji pro vyhledávání nebo dvojkliknutím na chybu v seznamu. V editoru je potom možné všechny vzory libovolně upravovat.

7.4 Nastavení

V příloze PŘÍLOHA P IX: Plugin – Obecné nastavení jsou zobrazeny obecné možnosti pro plugin.

- Výběr back-endu, procesoru, nástroje, který provádí samotný test
- Počet položek v okně procházení chyb. Velká hodnota by výrazně zpomalila načítání chyb do stromu.

Následují tři přepínače, které ovlivňují výpis chyb. První dva ovlivňují chyby jen v první kartě, poslední položka se vztahuje na obě karty s výpisem chyb.

- Je-li zaškrtnut první přepínač, ve stromu chyb se zobrazují jen ty odkazy, které vedou ke zdrojovým souborům, které patří do aktuálního workspace. Při použití této volby by se v ukázkovém seznamu nezobrazily odkazy na funkci malloc, ta je součástí externí knihovny, a ačkoli se v konečném důsledku chyba projeví tam, je způsobena ve funcích, které ji volají.
- Druhý přepínač odstraňuje identicky vypadající záznamy, které následují po sobě. Tato volba je užitečná při použití prvního přepínače. Existují však i zdrojové kódy, resp. programy, kde Valgrind sám vygeneruje výstup, ve kterém některé chyby vypadají stejně (liší se jen instrukčním ukazatelem, který zde není zobrazen).
- Třetí přepínač nachází uplatnění při potlačování chyb. Je-li nějaké chyba potlačena, je přidán její vzor do suppression souboru, je okamžitě odstraněna z výpisu a není tak nutné znovu spouštět test, aby se potlačení projevilo.

Příloha PŘÍLOHA P X: Plugin – Nastavení Valgrindu ukazuje možnosti nastavení Valgrindu, který je spouštěn na pozadí při testu. Implementoval jsem jen nejprostší variantu nastavení, tzn. uživatel má možnost nastavit přepínače v textové podobě, tak jako to musí udělat, když spouští Valgrind z příkazové řádky.

Možnosti nastavení jsou následující:

- Cesta k binárnímu souboru Valgrind
- Cesta k souboru protokolu. Bud se použije výchozí nastavení, kdy se soubor vytváří v privátním adresáři workspace, nebo může uživatel zadat vlastní cestu.
- Argumenty příkazového řádku pro Valgrind. První řádek obsahuje povinné argumenty, které jsou nutné pro fungování pluginu.
- Další argumenty. Zde jsou přednastaveny parametry, které je vhodné používat. Pro jejich vysvětlení je nutné přečíst dokumentaci Valgrindu. Tlačítko vpravo obnovuje výchozí nastavení doporučených parametrů.
- Seznam suppression souborů. Všechny zde vyjmenované soubory budou předány Valgrindu při každém testu. Kromě pevně daného seznamu může uživatel používat jeden suppression soubor, který se bude vždy měnit podle aktuálního workspace. Přidávání a odebírání souborů je možné pomocí pravého tlačítka myši a zobrazeného menu.

ZÁVĚR

Hlavní cíl práce se mi podařilo splnit, programátor může v rámci CodeLite spouštět profilaci svého programu pomocí nástroje Memcheck. Grafické provedení seznamu chyb je podle mého názoru nejkomplexnější z těch, které jsem zkoumal. Výrazným kladem pluginu je jeho integrace do IDE, což umožňuje přímou navigaci na místo problému. Kromě možnosti navigace jsem také implementoval možnost správy potlačovacích pravidel. V rámci svojí rešerše jsem nenalezl jiný program, který by se o to pokoušel.

Nedostatkem pluginu v současné chvíli je jen velmi prosté rozhraní pro nastavení parametrů spouštění Valgrindu. Této části jsem se nevěnoval záměrně, protože nemá žádný vliv na chod nebo vzhled pluginu jako celku. Její implementace je mechanická záležitost, ale vyžaduje určitý čas. Místo toho jsem se věnoval spíše inovativním prvkům nastavení pluginu, jako zvláštní případy filtrace chyb, nebo suppresion soubor, který se automaticky mění v závislosti na workspace. Další věc, která v pluginu chybí, ale měla by být v budoucnu implementována, je lepší podpora suppression souborů v editoru, tzn. minimálně zvýraznění syntaxe, shrnování bloků kódu, nebo dokonce automatické doplňování slov. Toto opět přímo nesouvisí s funkcí pluginu. Zvýraznění syntaxe a folding by mohlo být realizováno dokonce jako rozšíření knihovny wxWidgets tak, aby bylo zpřístupněno i dalším projektům, které používají komponentu wxStyledTextCtrl.

Další prostor pro zlepšení spočívá ve vytvoření dedikovaného modelu pro wxDataViewCtrl, což by umožnilo zvýšit rychlost celého prvku, a to především při načítání chyb. Další přínos by byl v možnosti lepšího grafického provedení seznamu chyb, umožnilo by to používat různé styly písma a barevné zvýraznění některých údajů. Prostor pro vylepšení hlavního grafického kontejneru pluginu, který staví na wxDataViewCtrl, není vůbec omezen. V ideálním případě by mohlo dojít až k vytvoření úplně nové komponenty v rámci knihovny wxWidgets, která by do wxDataViewCtrl přidávala možnost virtualizace, tedy stejného chování, jakým disponuje wxListCtrl. Umožňovalo by to v konečném důsledku sloučit funkcionalitu pluginu jen do jedné karty namísto současných dvou. Rozsah takové práce si ale netroufám vůbec odhadnout. Zároveň si myslím, že je nutné se rozmyslet, jestli se vůbec vyplatí vývoj tak velmi složité a komplexní komponenty, a to ať z globálního hlediska, nebo i jen v rámci uplatnění v CodeLite. Plugin, který jsem vytvořil, je v současné době stále jen prototyp, byť v daném rozsahu plně

funkční. Je potřeba zpětná vazba od uživatelů, aby se ukázalo, kam by měl směřovat další vývoj.

Na základě zkoumání dostupných profilačních nástrojů jsem zjistil, že výstupy, které produkují, jsou velmi podobné, mají podobnou strukturu i sémantiku. Z tohoto důvodu jsem Plugin od počátku koncipoval tak, aby byl rozšiřitelný. Architektura pluginu je tedy dvouvrstvá, což má kromě pozitivního vlivu na přehlednost a udržovatelnost hlavní výhodu v tom, že prezentační vrstva není přímo závislá na použitém back-endu. Zároveň jsem podle svých nejlepších schopností dbal na čistotu kódu a obecně zásady dobrého návrhu a implementace.

Celkově jsem se snažil, aby byl plugin přínosem pro programátorskou komunitu, ale práce na něm znamenala přínos i pro mě samotného. Rozšířil jsem si obzory v oblasti profilace software, naučil jsem se používat Valgrind a poznal jsem pro mě nová IDE. Osobně to vnímám jako znatelné zlepšení profesních schopností.

Poslední a opravdu velmi zajímavou zkušeností pro mě bylo zapojení se do globálního, celosvětového projektu. Jednak jsem si procvičil angličtinu, ale hlavně jsem ztratil ostych a nejistotu, které jsem zprvu měl vůči tak významnému projektu. Zjistil jsem, že ačkoli se ostatní vývojáři nacházejí fyzicky úplně někde jinde, jsou to stejní lidé jako já a že máme společný zájem. Zlepšil jsem si obecně svoje profesní komunikační schopnosti a cítím chuť se v budoucnu zapojit do dalších projektů, které jsou vyvíjeny na globálním internetovém půdorysu.

ZÁVĚR V ANGLIČTINĚ

I have managed to fulfil the main goal of this paper. The programmer can use CodeLite for memory profiling using the Memcheck tool. The graphical representation of the error list is the most complex one of all currently available tools in my opinion. One of the main advantages of the plugin is its IDE integration which allows the programmer to directly navigate to the exact error location. Apart from the navigation functionality I have also implemented manageable suppression rules. I have not found a plugin/program with a similar functionality.

One of the current limitations of the plugin is a very simple UI for adjusting Valgrind settings. I have not implemented this functionality in more detail on purpose as it has no effect on the actual plugin functionality and the overall look and feel. Its implementation would be a straightforward procedure, but I have rather decided to focus on innovative plugin settings such as special error filtration or a suppression file dynamically changing depending on the workspace. Another feature missing in the plugin and planned for a future implementation is a better support of suppresson files directly in the IDE. That means that at least some basic syntax highlighting, content folding or a code suggestion should be available to the user. Again this is not a fundamental functionality for the plugin to work. Syntaxt highlighting and folding could be realized as a wxWidgets library extension so that it is available for all other projects that are using the wxStyledTextCtrl.

An another aspect that could be further worked on would be a creation of a dedicated model for the wxDataViewCtrl which would improve the response time of the whole component mainly when loading error messages. The actual error list could also be enhanced graphically – different fonts and coloured syntax for certain types of records. There are practically no limitations for extending the functionality of the main graphical container of the plugin that is based on wxDataViewCtrl. What would happen in the ideal scenario is a whole new wxWidgets component being implemented that would enhance wxDataViewCtrl with virtualization capabilities – basically the same as wxListCtrl can do. That would in practice allow us to focus the plugin functions in only one tab instead of two as per current functionality. I do not dare estimate the time needed to implement such a feature, though. At the same time I think, that one must carefully consider whether developing such a complex feature would make sense considering both the global implications and potential usability within the CodeLite IDE. The plugin as described in

this paper is still an prototype, fully functional nonetheless. User feedback is really needed at this point to help decide on the right direction for further development of the plugin.

Based on the investigation of other currently available profiler tools I can conclude that outputs my plugin is producing are similar both structurally and semantically to those produced by other tools. It is also a reason why further extensibility was one of the main ideas to keep in mind while developing the plugin. The architecture of the plugin as a two-layer one which not only helps readability and maintainability, but also it ensures that the presentation layer is not directly dependant on the back-end used. I have tried to keep the code as clean as possible and to respect the general guidelines for a good software design and implementation.

I hope I managed to develop the plugin to be an useful asset tool within the developer's community, but working on it mainly benefited myself as well. I have learnt how to use Valgrind and I discovered a new IDE to work in. Personally my professional skills have developed immensely.

Last but not least, the experience of being a part of a global software project was priceless. I have both improved my English and gained much confidence working on a bigger and an important software project. I have found out that even though all the other project developers are physically somewhere else, we are all like-minded people. I have improved my professional communication skills a lot and I am definitely planning to be a part of other global software projects in the future.

SEZNAM POUŽITÉ LITERATURY

[1] SEWARD, Julian, Nicholas NETHERCOTE a Josef WEIDENDORFER. *Valgrind 3.3: Advanced Debugging and Profiling for GNU/Linux Applications*. Bristol: Network theory Ltd, 2008. ISBN 0-9546120-5-1.

[2] PRATA, Stephen. *Mistrovství v C. 3.* aktualiz. vyd. Překlad Boris Sokol. Brno: Computer Press, 1119 s. ISBN 978-80-251-1749-1.

[3] CORMEN, Thomas H. *Introduction to algorithms.* 3rd ed. Překlad Boris Sokol. Cambridge: MIT Press, c2009, xix, 1292 s. ISBN 978-0-262-03384-8.

[4] KANISOVÁ, Hana a Miroslav MÜLLER. *UML srozumitelně.* 2. aktualiz. vyd. Překlad Boris Sokol. Brno: Computer Press, c2009, 176 s. ISBN 80-251-1083-4.

[5] SMART, Julian a Miroslav MÜLLER. *Cross-platform GUI programming with wxWidgets.* 2. aktualiz. vyd. Překlad Boris Sokol. Upper Saddle River: Prentice-Hall, c2006, xxxv, 700 s. ISBN 01-314-7381-6.

[6] Valgrind Documentation. VALGRIND DEVELOPERS. *Valgrind* [online]. 3.9.0. 31.10.2013 [cit. 2014-01-23].
Dostupné z: http://valgrind.org/docs/manual/index.html.

[7] Reference: Standard C++ Library reference. *Cplusplus.com: The C++ Resources Network* [online]. 3.1. c2000-2014 [cit. 2014-01-23]. Dostupné z: http://www.cplusplus.com/reference/

[8] SMART, Julian et al. Documentation. *WxWidgets: Cross-platform GUI Toolkit* [online]. 3.0.0. 11.11.2013 [cit. 2014-01-23]. Dostupné z: http://docs.wxwidgets.org/stable/index.html

[9] NGUYEN, Radek. *Statická analýza kódu* [online]. Praha, 2012 [cit. 2014-04-04]. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/nguyehuy_2011dipl.pdf. Diplomová práce. České vysoké učení technické v Praze: Fakulta elektrotechnická. Vedoucí práce Ing. Radek Mařík, CSc.

[10] GAHURA, Jan. *Dynamická analýza podezřelého kódu* [online]. Praha, 2007 [cit. 2014-04-04]. Dostupné z:
https://dip.felk.cvut.cz/browse/pdfcache/gahurj1_2007dipl.pdf. Diplomová práce.

České vysoké učení technické v Praze: Fakulta elektrotechnická. Vedoucí práce Ing. Radim Ballner

[11] MACHEK, Ondřej. *Profilování software založené na instrumentaci kódu* [online]. 2013 [cit. 2014-04-04]. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Šimon Tóth. Dostupné z: http://is.muni.cz/th/374507/fi_b/.

[12] FAIGL, Jan. Profilování. In: *Metodiky programování* [online]. 2006 [cit. 2014-04-04]. Dostupné z: http://lynx1.felk.cvut.cz/mep/files/slides/lecture2b.pdf

[13] COPPA, Emilio, Camil DEMETRESCU a Irene FINOCCHI. Input-Sensitive Profiling. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. Peking, China, 2012 [cit. 2014-04-04]. Dostupné z: http://wwwusers.di.uniroma1.it/~finocchi/slides/aprof-PLDI.pdf

[14] KIRIANSKY, Vladimir, Derek BRUENING a Saman AMARASINGHE. Secure Execution via Program Shepherding. In: *11th Usenix Security Symposium* [online]. 2002 [cit. 2014-04-05]. Dostupné z: https://www.usenix.org/conference/11th-usenix-security-symposium/secure-execution-program-shepherding

[15] Memory leak. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-04-05]. Dostupné z: http://en.wikipedia.org/wiki/Memory_leak

[16] Integrated development environment. Free On-line Dictionary of Computing [online]. 2002 [cit. 2014-04-06]. Dostupné z: http://foldoc.org/integrated+development+environment

[17] MAČEK, Pavel. *Návrh uživatelského rozhraní aplikace pro správu úkolů* [online]. 2011 [cit. 2014-04-09]. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Jiří Sochor. Dostupné z: http://is.muni.cz/th/172976/fi_m/.

[18] PAPÍK, Richard. Vyhledávání informací II.: Uživatelské rozhraní a vlivy oboru “human-computer interaction”. *Národní knihovna: Knihovnická revue* [online].

Praha: Národní knihovna ČR, 2001, č. 2, s. 81-90 [cit. 2014-04-09]. Dostupné z: http://knihovna.nkp.cz/NKKR0102/0102081.html

[19] DOSTÁL, Martin. Základy tvorby uživatelského rozhraní [online]. Olomouc, 2010, 18. 12. 2010 [cit. 2014-04-09]. Dostupné z: http://dostal.inf.upol.cz/data/0910/URO/uro-18-12-2010.pdf

[20] SEWARD, Julian. THE VALGRIND DEVELOPERS. *Valgrind* [online]. © 2000-2013 [cit. 2014-04-11]. Dostupné z: http://www.valgrind.org/

[21] SCHWARZ, Christoph. *Valgrind for Windows: Valgrind port for Microsoft Windows* [online]. 2011-, 13.07 2012 [cit. 2014-04-11]. Dostupné z: http://sourceforge.net/projects/valgrind4win/

[22] *KCachegrind: Profiling Visualization* [online]. 2002-, 27.9.2005 [cit. 2014-04-11]. Dostupné z:

http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex

[23] LINDH, Johan. *Memwatch* [online]. © 2014 [cit. 2014-04-11]. Dostupné z: http://www.linkdata.se/sourcecode/memwatch/

[24] MICRO FOCUS. *DevPartner* [online]. © 2014 [cit. 2014-04-12]. Dostupné z: http://www.borland.com/products/devpartner/default.aspx

[25] A description of the differences between ActiveCheck and FinalCheck. MICRO FOCUS. *Micro Focus: community* [online]. © 2001 - 2014 [cit. 2014-04-12]. Dostupné z: http://community.microfocus.com/borland/develop/devpartner_-_code_analysis/w/knowledge_base/5259.a-description-of-the-differences-between-activecheck-and-finalcheck.aspx

[26] DevPartner BoundsChecker Web Edition. MICRO FOCUS. *Micro Focus: store* [online]. © 2001 - 2014 [cit. 2014-04-12]. Dostupné z: https://www.microfocus.com/store/devpartner/boundschecker.aspx

[27] IBM. *Rational Purify family: Dynamic software analysis with memory debugging and memory leak protection* [online]. 2014 [cit. 2014-04-12]. Dostupné z: http://www-03.ibm.com/software/products/en/rational-purify-family/

[28] RATIONAL SOFTWARE. *Purify User's Guide* [online]. [cit. 2014-04-12]. Dostupné z: ftp://aix.boulder.ibm.com/software/rational/docs/documentation/manuals/unixsuites /pdf/purify/purify.pdf

[29] PARASOFT. *Insure++: Runtime Analysis and Memory Error Detection for C and C++* [online]. © 1996-2014 [cit. 2014-04-12]. Dostupné z: http://www.parasoft.com/insure

[30] PARASOFT. *Insure++ User's Guide* [online]. © 1996-2000 [cit. 2014-04-12]. Dostupné z: http://vasc.ri.cmu.edu/media/manuals/insure.5.x/users/index.htm

[31] *DynamoRIO: Dynamic Instrumentation Tool Platform* [online]. 2000-2014 [cit. 2014-04-13]. Dostupné z: http://dynamorio.org/home.html

[32] *Dr. Memory: Memory Debugger for Windows and Linux* [online]. 2014 [cit. 2014-04-13]. Dostupné z: http://www.drmemory.org/

[33] *Drmemory: Issue 1128: create simple GUI for launching apps and viewing results* [online]. 2013, 17. 2. 2013 [cit. 2014-04-13]. Dostupné z:

http://code.google.com/p/drmemory/issues/detail?id=1128

[34] *Stack Overflow: Memory leak c++ program* [online]. 2012 [cit. 2014-04-13]. Dostupné z: http://stackoverflow.com/questions/10004785/memory-leak-c-program

[35] NEELAM, Babu. *Valgrind's Memcheck + Valkyrie: Detect user space run time memory errors. TechVolve* [online]. 20. 2. 2014 [cit. 2014-04-16]. Dostupné z: http://techvolve.blogspot.cz/2014/03/valgrind-memcheck-valkyrie-detect-user.html

[36] STEDFAST, Jeffrey. *Alleyoop* [online]. 2003-2011 [cit. 2014-04-17]. Dostupné z: http://alleyoop.sourceforge.net/

[37] CHARNEY, Reg. *Valgrind 2.2.0: Memory Debugging and Profiling. Linux Journal* [online]. © 1994 - 2014, 02. 12. 2004 [cit. 2014-04-17]. Dostupné z: http://www.linuxjournal.com/node/7930/print

[38] GYLLENHAAL, John. *Using Valgrind's Memcheck Tool to Find Memory Errors and Leaks in MPI and Serial Applications on Linux* [online]. 11. 09. 2012 [cit. 2014-04-17]. Dostupné z: https://computing.llnl.gov/code/memcheck/

[39] MemcheckView. GYLLENHAAL, John. *Tool Gear* [online]. 06. 12. 2012 [cit. 2014-04-17]. Dostupné z: http://computation.llnl.gov/casc/tool_gear/memcheckview.html

[40] *Linux Tools Project: Valgrind Support* [online]. 04. 03. 2014 [cit. 2014-04-17]. Dostupné z: http://www.eclipse.org/linuxtools/projectPages/valgrind/

[41] Valgrind plugin. THE CODE::BLOCKS TEAM. *Code::Blocks wiki* [online]. 25. 02. 2012 [cit. 2014-04-17]. Dostupné z: http://wiki.codeblocks.org/index.php?title=Valgrind_plugin

[42] *KDev-Valgrind* [online]. 05. 01. 2012 [cit. 2014-04-17]. Dostupné z: http://eip.epitech.eu/2012/kdevvalgrind/index.html

[43] KDevelop 4.4.0 Beta 1 Released. THE KDEVELOP TEAM. *KDevelop* [online]. 14. 08. 2012 [cit. 2014-04-17]. Dostupné z: http://kdevelop.org/44/kdevelop-440-beta-1-released

[44] Using CTags as main parser. *Code::Blocks: forums* [online]. [2004-2014] [cit. 2014-04-19]. Dostupné z: http://forums.codeblocks.org/index.php/topic,1889

[45] Suggestion: Using ctags & sqlite for code completion. *Code::Blocks: forums* [online]. [2004-2014] [cit. 2014-04-19]. Dostupné z: http://forums.codeblocks.org/index.php/topic,3872

[46] CodeLite IDE v1.0 is vailable for download!. IFRAH, Eran. *Codelite IDE: forums* [online]. [2008-2014] [cit. 2014-04-19]. Dostupné z: http://forums.codelite.org/viewtopic.php?f=9&t=53

[47] IFRAH, Eran. *CodeLite: Documentation* [online]. 11. 01. 2014 [cit. 2014-04-19]. Dostupné z: http://codelite.org/LiteEditor/Documentation

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

IDE Integrated Development Environment – integrované vývojové prostředí.

GUI Graphical User Interface - grafické uživatelské rozhraní.

SEZNAM OBRÁZKŮ

Obr. 1 Ukázka programu Valkyrie[35] 30

Obr. 2 Ukázka programu Alleyoop[37] 31

Obr. 3 Ukázka programu MemcheckView[39] 33

Obr. 4 Ukázka pluginu pro Valgrind v Eclipse IDE 34

Obr. 5 Ukázka IDE Code::Blocks s pluginem pro Valgrind 35

Obr. 6 Ukázka pluginu pro Valgrind v IDE KDevelop 36

Obr. 7 Okno IDE CodeLite verze 5.4 ve Windows (A: toolbar; B: Worspace View; C: editor; D: OutputView) 38

SEZNAM PŘÍLOH

PŘÍLOHA P I: Ukázkový zdrojový kód

PŘÍLOHA P II: Memcheck – textový protokol

PŘÍLOHA P III: Memcheck – Protokol XML

PŘÍLOHA P IV: Diagram tříd

PŘÍLOHA P V: Sekvenční diagram

PŘÍLOHA P VI: Plugin – Spuštění testu

PŘÍLOHA P VII: Plugin – Procházení chyb

PŘÍLOHA P VIII: Plugin – Správa potlačovacích pravidel

PŘÍLOHA P IX: Plugin – Obecné nastavení

PŘÍLOHA P X: Plugin – Nastavení Valgrindu

PŘÍLOHA P I: UKÁZKOVÝ ZDROJOVÝ KÓD

```
 1  #include <stdlib.h>
 2
 3  void f(void) {
 4    int *x = malloc(10 * sizeof(int));
 5    x[10] = 0;         // problem 1: heap block overrun
 6  }                    // problem 2: memory leak -- x not freed
 7
 8  int main(void) {
 9    f();
10    return 0;
11  }
```

PŘÍLOHA P II: MEMCHECK – TEXTOVÝ PROTOKOL

```
==5678== Memcheck, a memory error detector
==5678== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==5678== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==5678== Command: ./example
==5678== Parent PID: 1234
==5678==
==5678== Invalid write of size 4
==5678==    at 0x40052A: f (/tmp/example/main.c:5)
==5678==    by 0x40053A: main (/tmp/example/main.c:9)
==5678==  Address 0x51b9068 is 0 bytes after a block of size 40 alloc'd
==5678==    at 0x4C28BED: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==5678==    by 0x40051D: f (/tmp/example/main.c:4)
==5678==    by 0x40053A: main (/tmp/example/main.c:9)
==5678==
{
   <insert_a_suppression_name_here>
   Memcheck:Addr4
   fun:f
   fun:main
}
==5678==
==5678== HEAP SUMMARY:
==5678==     in use at exit: 40 bytes in 1 blocks
==5678==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==5678==
==5678== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5678==    at 0x4C28BED: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==5678==    by 0x40051D: f (/tmp/example/main.c:4)
==5678==    by 0x40053A: main (/tmp/example/main.c:9)
==5678==
{
   <insert_a_suppression_name_here>
   Memcheck:Leak
   fun:malloc
   fun:f
   fun:main
```

```
}
==5678== LEAK SUMMARY:
==5678==    definitely lost: 40 bytes in 1 blocks
==5678==    indirectly lost: 0 bytes in 0 blocks
==5678==      possibly lost: 0 bytes in 0 blocks
==5678==    still reachable: 0 bytes in 0 blocks
==5678==         suppressed: 0 bytes in 0 blocks
==5678==
==5678== For counts of detected and suppressed errors, rerun with: -v
==5678== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 4 from 4)
```

PŘÍLOHA P III: MEMCHECK – PROTOKOL XML


```
<?xml version="1.0"?>

<valgrindoutput>

<protocolversion>4</protocolversion>
<protocoltool>memcheck</protocoltool>

<preamble>
  <line>Memcheck, a memory error detector</line>
  <line>Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.</line>
  <line>Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright
info</line>
  <line>Command: ./example</line>
</preamble>

<pid>5678</pid>
<ppid>1234</ppid>
<tool>memcheck</tool>

<args>
  <vargv>
    <exe>/usr/bin/valgrind.bin</exe>
    <arg>--suppressions=/usr/lib/valgrind/debian-libc6-dbg.supp</arg>
    <arg>--tool=memcheck</arg>
    <arg>--xml=yes</arg>
    <arg>--fullpath-after=</arg>
    <arg>--gen-suppressions=all</arg>
    <arg>--xml-file=/tmp/.codelite/valgrind.memcheck.log.xml</arg>
    <arg>--suppressions=/tmp/.codelite/valgrind.memcheck.supp</arg>
    <arg>--leak-check=yes</arg>
    <arg>--track-origins=yes</arg>
  </vargv>
  <argv>
    <exe>./example</exe>
  </argv>
</args>

<status>
```

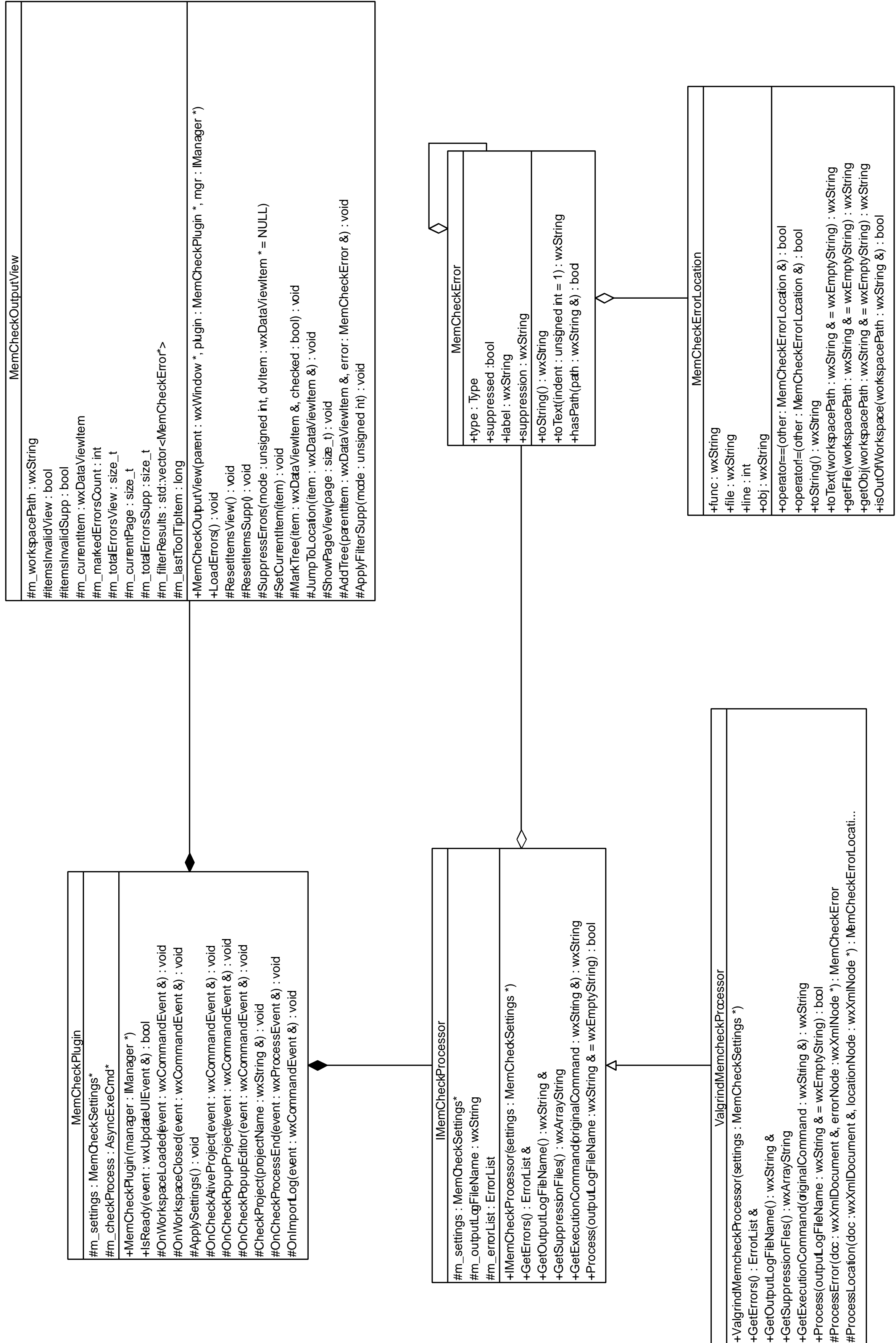
```
  <state>RUNNING</state>
  <time>00:00:00:00.054 </time>
</status>

<error>
  <unique>0x4</unique>
  <tid>1</tid>
  <kind>InvalidWrite</kind>
  <what>Invalid write of size 4</what>
  <stack>
    <frame>
      <ip>0x40052A</ip>
      <obj>/tmp/example/Debug/example</obj>
      <fn>f</fn>
      <dir>/tmp/example</dir>
      <file>main.c</file>
      <line>5</line>
    </frame>
    <frame>
      <ip>0x40053A</ip>
      <obj>/tmp/example/Debug/example</obj>
      <fn>main</fn>
      <dir>/tmp/example</dir>
      <file>main.c</file>
      <line>9</line>
    </frame>
  </stack>
  <auxwhat>Address 0x51b9068 is 0 bytes after a block of size 40
alloc'd</auxwhat>
  <stack>
    <frame>
      <ip>0x4C28BED</ip>
      <obj>/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so</obj>
      <fn>malloc</fn>
    </frame>
    <frame>
      <ip>0x40051D</ip>
      <obj>/tmp/example/Debug/example</obj>
      <fn>f</fn>
      <dir>/tmp/example</dir>
```

```
        <file>main.c</file>
        <line>4</line>
      </frame>
      <frame>
        <ip>0x40053A</ip>
        <obj>/tmp/example/Debug/example</obj>
        <fn>main</fn>
        <dir>/tmp/example</dir>
        <file>main.c</file>
        <line>9</line>
      </frame>
    </stack>
    <suppression>
      <sname>insert_a_suppression_name_here</sname>
      <skind>Memcheck:Addr4</skind>
      <sframe> <fun>f</fun> </sframe>
      <sframe> <fun>main</fun> </sframe>
      <rawtext>
<![CDATA[
{
   <insert_a_suppression_name_here>
   Memcheck:Addr4
   fun:f
   fun:main
}
]]>
      </rawtext>
    </suppression>
</error>


<status>
  <state>FINISHED</state>
  <time>00:00:00:00.418 </time>
</status>

<error>
  <unique>0x5</unique>
  <tid>1</tid>
```

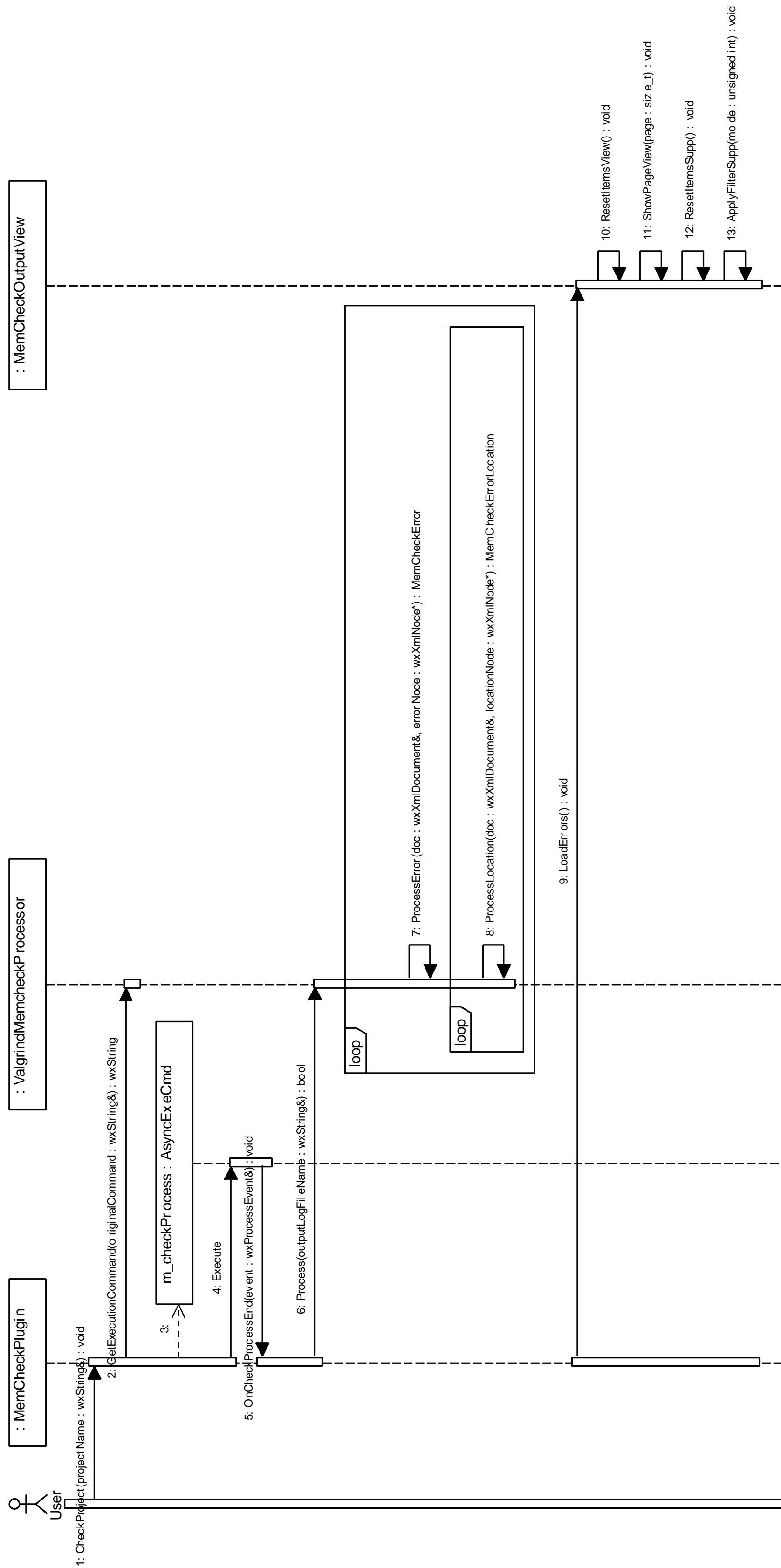
```
  <kind>Leak_DefinitelyLost</kind>
  <xwhat>
    <text>40 bytes in 1 blocks are definitely lost in loss record 1 of 1</text>
    <leakedbytes>40</leakedbytes>
    <leakedblocks>1</leakedblocks>
  </xwhat>
  <stack>
    <frame>
      <ip>0x4C28BED</ip>
      <obj>/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so</obj>
      <fn>malloc</fn>
    </frame>
    <frame>
      <ip>0x40051D</ip>
      <obj>/tmp/example/Debug/example</obj>
      <fn>f</fn>
      <dir>/tmp/example</dir>
      <file>main.c</file>
      <line>4</line>
    </frame>
    <frame>
      <ip>0x40053A</ip>
      <obj>/tmp/example/Debug/example</obj>
      <fn>main</fn>
      <dir>/tmp/example</dir>
      <file>main.c</file>
      <line>9</line>
    </frame>
  </stack>
  <suppression>
    <sname>insert_a_suppression_name_here</sname>
    <skind>Memcheck:Leak</skind>
    <sframe> <fun>malloc</fun> </sframe>
    <sframe> <fun>f</fun> </sframe>
    <sframe> <fun>main</fun> </sframe>
    <rawtext>
<![CDATA[
{
   <insert_a_suppression_name_here>
```

```
   Memcheck:Leak
   fun:malloc
   fun:f
   fun:main
}
]]>
    </rawtext>
  </suppression>
</error>

<errorcounts>
  <pair>
    <count>1</count>
    <unique>0x4</unique>
  </pair>
</errorcounts>

<suppcounts>
  <pair>
    <count>4</count>
    <name>dl-hack3-cond-1</name>
  </pair>
</suppcounts>

</valgrindoutput>
```

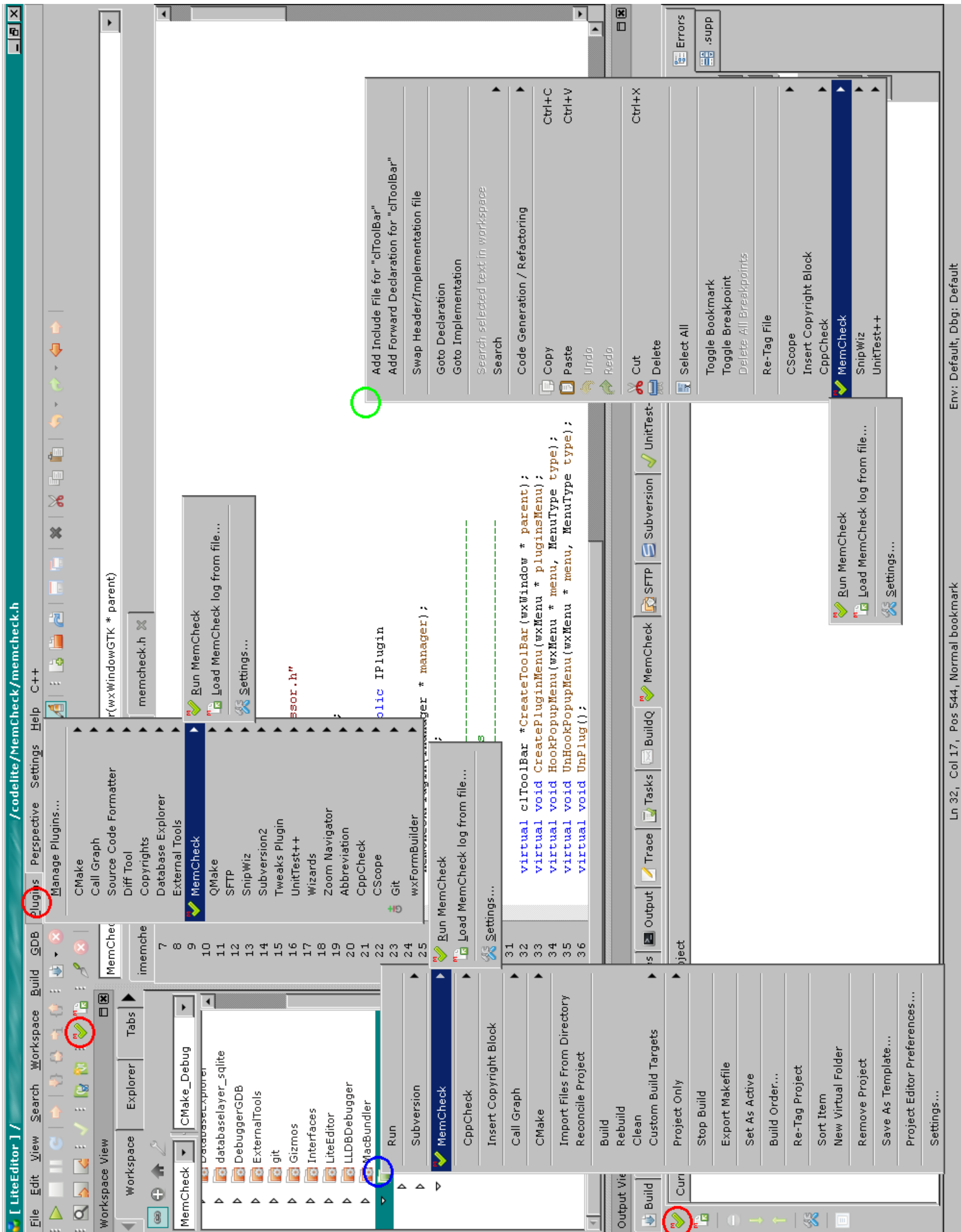
PŘÍLOHA P IV: DIAGRAM TŘÍD



PŘÍLOHA P V: SEKVENČNÍ DIAGRAM



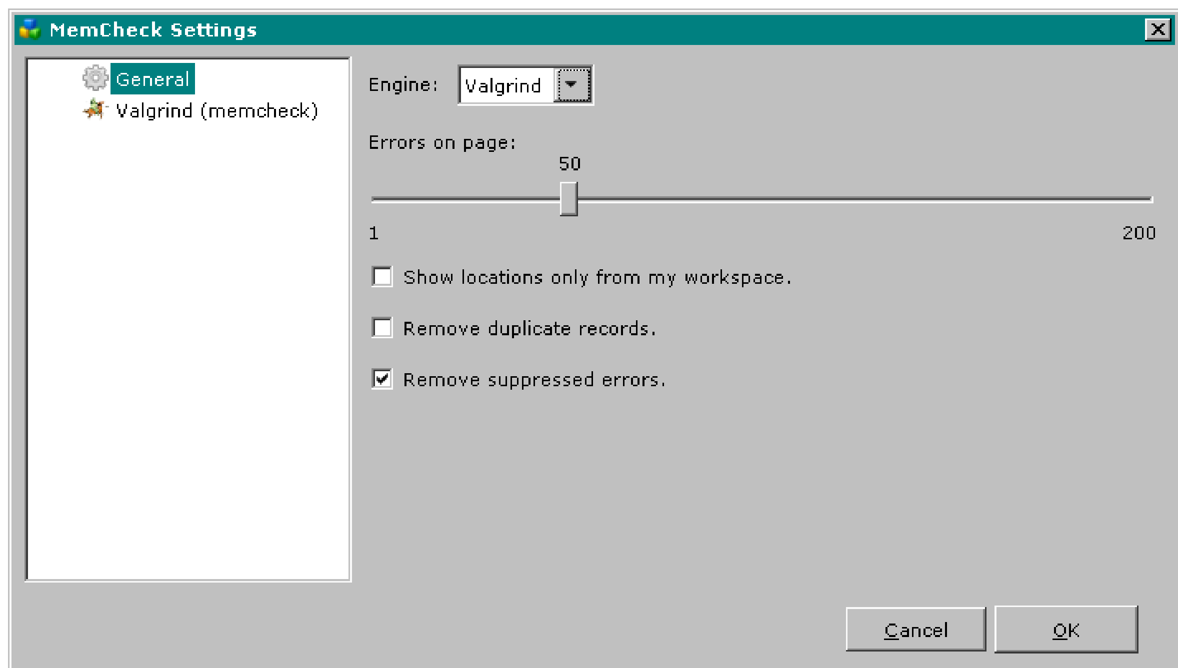
PŘÍLOHA P VI: PLUGIN – SPUŠTĚNÍ TESTU



PŘÍLOHA P VII: PLUGIN – PROCHÁZENÍ CHYB

PŘÍLOHA P VIII: PLUGIN – SPRÁVA POTLAČOVACÍCH PRAVIDEL

PŘÍLOHA P IX: PLUGIN – OBECNÉ NASTAVENÍ



PŘÍLOHA P X: PLUGIN – NASTAVENÍ VALGRINDU