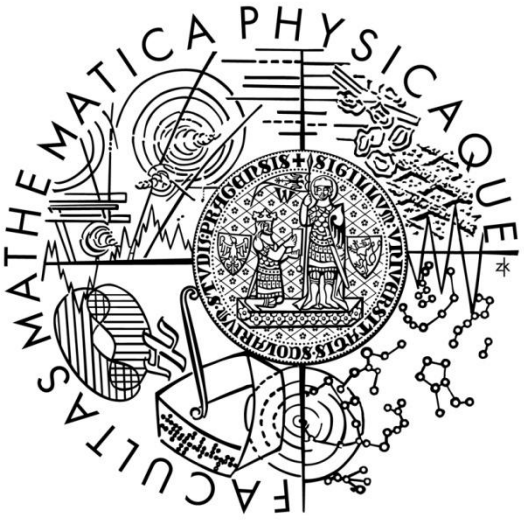
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Přemysl Kouřil

Multiplatformní buildovací systém

Ústav formální a aplikované lingvistiky

Vedoucí diplomové práce: RNDr. Miroslav Spousta

Studijní program: Informatika
Studijní obor: Distribuované systémy

Praha rok 2012

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.
Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne: Přem ysl Kouřil

Abstrakt

Název práce: *Multiplatformní buildovací systém*
Autor: *Přemysl Kouřil*
Katedra: *Ústav formální a aplikované lingvistiky*
Vedoucí diplomové práce: *RNDr. Miroslav Spousta*
E-mail vedoucího práce: *miroslav.spousta@mff.cuni.cz*
Abstrakt: *Build systém je důležitou součástí softwarových projektů a téměř všechny procesy spjaté s vývojem software se více či méně dotýkají build systému. Složitost build systému roste úměrně se složitostí softwarového projektu. Cílem práce je vytvořit návrh build systému vhodného pro nasazení ve vysoce multiplatformních projektech a přizpůsobený specifickým potřebám softwaru vyvíjeného v komerčním prostředí organizací. Práce nejprve definuje kontext a poskytne náhled do problematiky a poté analyzuje teoretické aspekty klíčových problémů. Součástí práce je i stručná analýza nástrojů pro řízení překladu. Na základě srovnání dostupných technologií a analýzy klíčových problémů je představen návrh build systému pro danou třídu softwarových projektů. Pro jádro build systému je použit nástroj SCons. Práce by měla vývojáři build systému poskytnout dostatečný aparát na to, aby byl schopen pro specifický softwarový projekt implementovat build systém, splňující všechny hlavní atributy, určující kvalitní build systém.*

Klíčová slova: *Build systém, SCons, multiplatformní*

Abstract

Title: *Multiplatform build system*
Author: *Přemysl Kouřil*
Department: *Institute of Formal and Applied Linguistics*
Supervisor: *RNDr. Miroslav Spousta*
Supervisor's e-mail address: *miroslav.spousta@mff.cuni.cz*
Abstract: *Build system is an important part of software projects and almost all processes involved in software development are more or less connected to a build system. A complexity of a build system increases accrodingly to a complexity of a software project. The goal of this thesis is to introduce a proposal for a build system suitable for use in highly multiplatform projects and adapted to a specific needs of software developed in enterprise environment. This thesis first defines context and provides an overview on the topic and then analyzes theoretical aspects of key problems. Brief analysis of build tools is included in this thesis. Based on a comparison of available technologies and analysis of key problems a build system proposal is introduced for the specific class of software projects. SCons tool is used as a core of proposed build system. Thesis shall provide a developer with an apparatus strong enough so that developer is able to implement a build system that satisfies all key attributes which determine good build system*

Keywords: *Build system, SCons, multiplatform*

Obsah

Úvod 8
1 Cíle 9
2 Pozadí 11
2.1 Build systém a jeho role v komerčním software 11
2.1.1 Překlad 11
2.1.2 Nepřetržitá integrace 12
2.1.3 Integrovaná vývojová prostředí (IDE) 12
2.1.4 Správa konfigurací softwaru 13
2.1.5 Packaging 13
2.1.6 Deployment 14
2.1.7 Release Management 14
2.1.8 Správa variant 15
2.2 Přehled nástrojů pro řízení překladu 15
2.2.1 Make & Autotools 16
2.2.2 CMake 17
2.2.3 Jam 18
2.2.4 Ant 18
2.2.5 SCons 19
2.3 Make, GNU Make a GNU Autotools 20
2.3.1 Princip fungování Make 20
2.3.2 GNU Autotools 20
2.4 SCons 23
2.4.1 Cíle 24
2.4.2 Architektura 24
3 Analýza 28
3.1 Softwarový projekt jako AOG 28
3.2 Definování SP jako AOG – Build systém a build skripty 30
3.3 Varianty 31

3.4 Přístup Make vs. Přístup SCons ... 33
3.4.1 Popis struktury AOG ... 33
3.4.2 Dynamická povaha build systému ... 35
3.4.3 Deklarativní vs. imperativní ... 37
4 Návrh ... 40
4.1 Výběr vhodného nástroje ... 40
4.2 Vývojové prostředí ... 41
4.2.1 Struktura vývojového prostředí ... 42
4.2.2 Formalizace vývojového prostředí ... 46
4.3 Princip vzdáleného překladu ... 47
4.4 Paralelizace překladu ... 48
4.5 Struktura build systému a build skriptů ... 49
4.5.1 Zdrojové uzly – hlavní ukazatel struktury rozložení build skriptů ... 49
4.5.2 Vnitřní uzly ... 51
4.6 Varianty ... 53
4.6.1 Přesun rozhraní ... 53
4.6.2 Implementace v doméně nástroje SCons ... 56
4.7 Separování cílů od zdrojů ... 60
4.7.1 Separování pomocí VPATH ... 60
4.7.2 Separování pomocí prefixování ... 62
4.8 Procesy mimo závislostní systém ... 63
4.8.1 Externí závislosti ... 63
4.8.2 Integrace s nástroji pro správu revizí ... 65
5 Nasazení ... 67
5.1 Motivace ... 67
5.2 Přechod ... 68
5.3 Přijetí ... 68
6 Závěr ... 69
Seznam použité literatury ... 70
Seznam použitých zkratek ... 71
Příloha A ... 72
Platformy podporované produktem NSM ... 72

Úvod

Životní cyklus softwaru sestává z několika fází, z nichž jednou z nejvýznamnějších je fáze vývoje. Stejně tak vývoj softwaru sestává z fází zahrnující např. Analýzu, programování, překlad, ladění a testování (ne nutně v tomto pořadí).

Podíváme-li se blíže na fázi překladu – která má za úkol transformovat sadu zdrojových entit do sady produkčních entit zjistíme, že se jedná o netriviální úkol, který v minulosti motivoval vývojáře k tomu, aby vytvořili nástroje, které překlad řídí. Jedním z nejznámějších nástrojů s dlouholetou historií je Make a jeho různé implementace (GNU Make), které patří i dnes k nejpoužívanějším nástrojům k řízení překladu. I přes existenci těchto nástrojů je často překlad tak složitou záležitostí, že sama jeho realizace v doméně těchto nástrojů je tak rozsáhlá, že vzniká ve své podstatě autonomní systém, který má své specifické chování, algoritmy a variace vzhledem k softwaru a prostředí, pro jehož účely byl vytvořen a optimalizován – vzniká build systém.

S rostoucí komplexností softwarových projektů se začaly klást rostoucí požadavky i na systém řízení překladu. Význam těchto systémů přesahuje čistě překlad. Je to věc, se kterou vývojář přichází denně do styku, a tyto systémy z velké části definují to, jakým způsobem se bude software vyvíjet. Reflektují strukturu projektu, definují procesy týkající se jak samotného vývoje, tak správy a údržby softwaru. Uvědomíme-li si, že v rámci těchto systému se definují například instalační cesty či generování ladících informací mají v neposlední řadě vliv na to, jakým způsobem se bude software ladit a testovat.

Vzhledem k tomu, že softwarové projekty se velmi liší svými požadavky na build systém, je důležité, aby byl build systém navržen s ohledem na tyto specifika tak, aby byl maximálně použitelný, udržovatelný a bezpečný. Podíváme-li se do sféry svobodného softwaru respektive do projektu GNU nalezneme zde v současnosti nejpoužívanější build systém dnešní doby na UNIX/Linuxových platformách – GNU Build System (známý také pod názvem Autotools). GNU build systém byl vytvořen s ohledem na velikou různorodost, která panuje v UNIXovém prostředí, kde jsou mezi různými operačními systémy, jako jsou Linux, Solaris, AIX atd. Odlišnosti v prostředí, které znesnadňují přenositelnost kódu (odlišnosti v kompilátorech, hlavičkových souborech, implementaci knihoven či jádra atd.)

V komerčním prostředí často panuje poněkud odlišná situace a klade se zde důraz na odlišné věci. Přenositelnost zdrojového kódu, respektive build systému není tak důležitá jako u svobodného a open source softwaru, protože v komerčním prostředí se často distribuují binární podoby softwaru a ne jeho zdrojové kódy. Naopak se však kladou vyšší nároky na škálovatelnost, jednoduchost a transparentnost umožňující snadné integrace a disintegrace s jinými softwarovými projekty, které jsou běžné v komerčním prostředí.

1 Cíle

Cílem této práce je navrhnout build systém vhodný pro nasazení v komerčním softwarovém projektu. Základní vlastnosti softwarového projektu, pro který je build systém určen, jsou následující:

- **Multi-platformnost** - softwarový projekt podporuje platformy MS Windows, GNU/Linux, Sun Solaris.
- **Dynamičnost** - softwarový projekt do sebe může v průběhu své existence integrovat nebo byt integrován s jiným softwarovým projektem či produktem (např. při akvizici konkurenční společnosti jsou fragmenty konkurenčního projektu integrovány do stávajícího). Nelze také předpokládat, že složení týmu zůstane po celou dobu života softwarového projektu neměnné. Naopak je potřeba počítat s tím, že může docházet k častým a často i velice rychlým změnám ve složení týmu, podle toho jak jednotliví zaměstnanci odcházejí či přicházejí do projektu či do firmy případně podle toho jak se mění jejich role v rámci týmu.
- **Různorodost technologií** - Softwarový projekt je heterogenní co se týče použitých technologií. Jsou použity různé programovací jazyky, externí závislosti a různé komponenty.

Vzhledem k výše uvedeným atributům by měl mít Build systém následující vlastnosti:

- **Jednotnost** - Build systém by měl být jednotný napříč podporovanými platformami. To znamená vyvarovat se pokud možno vyvíjení a udržovaní více nezávislých build systémů pro různé platformy či jazyky (případně komponenty). Snahou také je, aby build systém definoval jednotný strom závislostí napříč platformami.
- **Jednoduchost** - Systém má být co nejjednodušší, snadný na pochopení a na používání. Je pravděpodobné, že pozice správce build systému se bude často měnit a je proto potřeba, aby se při předávání správy nad build systémem byl nový správce schopen rychle zorientovat a pochopit interní principy build systému. Stejně má také být jednoduchý pro používání, aby se nově příchozí vývojáři rychle sžili s build systémem.
- **Konzistence** - Systém by měl být konzistentní a neredundantní. Uživatel by měl specifikovat pouze "zajímavé" informace/data. Nemělo by docházet například k nechtěným propagacím proměnných nebo naopak specifikování redundantních proměnných či dat.
- **Správnost a bezpečnost** - Systém by měl do důsledku udržovat závislosti mezi entitami. Nesmí vznikat díry ve stomě závislosti. Systém budou často používat

vývojáři, kteří nemají detailní přehled o interním fungování build systému a proto musí byt těžké vůbec takovéto díry vytvořit.

- **Modulárnost a snadná rozšiřitelnost** - Systém má byt snadno rozšiřitelný tak, aby umožňoval snadné přidávání či odebíraní komponent a má být dostatečně flexibilní, aby se do systému daly integrovat respektive importovat cizí projekty postavené na zcela odlišném build systému.

2 Pozadí

2.1 Build systém a jeho role v komerčním software

Tato část se snaží popsat, jak build systém interaguje s ostatními elementy, figurujícími v softwarovém projektu a obecně ve vývoji softwaru v komerčním prostředí. To umožní vytvořit si lepší představu o rolích build systému v procesu vývoje software a umožní zvýraznit jeho důležitou úlohu.

S pomocí automatizovaných nástrojů pro řízení překladu můžou vývojáři definovat jednotlivé kroky v procesu konstrukce výsledného softwaru a tyto kroky spolehlivě reprodukovat v různých prostředích a za různých podmínek. Typický proces konstrukce software v komerčním prostředí pokrývá všechny aspekty jak produkčního tak vývojového stadia daného softwaru, které se ovšem můžou lišit (například v databázové aplikaci jednotliví vývojáři můžou potřebovat inicializovat databázi nějakými vzorkovými testovacími daty, zatímco v produkčním prostředí takový krok není potřeba).

Postupem času, tím jak se nároky a složitost softwarového projektu zvětšují, věci jako testování či generace dokumentace si najdou svoji cestu do build systému. Většina vývojářů a obzvláště těch, kteří mají na starosti vývoj a udržovaní build systému, zjistí, že začínají s jednoduchým build systémem, který se postupně vyvine do složitého systému obstarávajícího daleko víc úkolů než obyčejný překlad. Od testování po dokumentaci, dobře navržený build systém se stává reflexí vývojových procesů v softwarovém projektu.

Konzistentní build systém figuruje jako společný jmenovatel pro většinu rolí definovaných v rámci vývojového týmu a to od vývojářů přes testery a dokumentátory až po manažery a po lidi, mající na starosti deployment aplikací.

Hlavní role, které build systém zastává v celém procesu vývoje komerčního software, jsou popsány v následujících bodech. Velká část z nich je platná i pro Open Source software, ovšem Open source software má jistá specifika, která značně odlišují proces jeho vývoje od vývoje komerčního software. Ty odlišnosti, které pak mají vliv na podobu build systému budou zdůrazněny v patřičných sekcích.

2.1.1 Překlad

Build systém se stará o transformaci zdrojových artefaktů na jiné artefakty. To zahrnuje kompilaci zdrojových kódů, generování hlavičkových souborů, transformaci textur z jednoho formátu do jiného atp. Těchto transformací je v softwarovém projektu obrovské množství a navíc mezi nimi existují závislosti, proto má smysl tyto transformace formalizovat a automatizovat, což je hlavním úkolem build systému. Velké softwarové projekty obsahují obrovské množství modulů, programů, knihoven atd. Vývojář často

pracuje pouze na jedné izolované části tohoto rozsáhlého projektu a proto by build systém měl umožňovat přeložit pouze izolovanou část projektu. Protože překlad je výpočetně náročná operace, build systém umožňuje provádět inkrementální překlad, což znamená, že při spuštění překladu se přeloží pouze ty části, které byly změněny od posledního spuštění. Pokročilými featurami build systému, může byt distribuovaný překlad, kdy překlad se distribuuje mezi několik výpočetních strojů, paralelní překlad, kdy nezávislé části jsou překládány paralelně či caching, kdy přeložené mezisoubory jsou uloženy v paměti pro použití v příštím spuštění.

2.1.2 Nepřetržitá integrace

Mezi build systémem a balíkem testů (např. Unit testů) je úzké spojení. Vývojové postupy jako např. Extremní Programovaní silně závisí na důkladném testovaní a na robustním balíku testů, které jsou v ideálním případě aplikovány při každé změně ve zdrojovém kódu. V případě, že není definován způsob, jak jednotlivé vývojáře přinutit spouštět tyto testy, můžou být principy metodologií jako je např. test driven programming porušovány. To může způsobit selhaní cílů, které si tyto metodologie kladou za úkol splnit. Build systém řeší tento problém, protože poskytuje ideální způsob, jak prosadit používaní testů a jak aplikaci testů učinit opakovatelnou a automatizovanou. Tyto myšlenky byly posunuty na novou úroveň s příchodem integračních serverů, což jsou servery, které pravidelně a automaticky kontrolují zdrojový kód (nezávisle na kterémkoliv vývojáři) a detekují nové změny ve zdrojovém kódu. Při změně v kódu je software automaticky přeložen a množina testů aplikována, přičemž pokud testy selžou, vývojáři jsou o problému automaticky upozorněni. Pokud uspějí je možné zdrojový kód označit a považovat ho jako nový release daného softwaru.

2.1.3 Integrovaná vývojová prostředí (IDE)

Za účelem zvýšení produktivity vývojářů zkombinovali dodavatelé nástroje pro překlad, editory, frameworky pro unit testy, debuggery, prohlížeče dokumentace a build systémy do jednotného vývojového prostředí (IDE – Integrated development environment). V současné době jsou tyto IDE extremně populární a každý, kdo vyvíjel v jazycích jako C++ nebo Java, se s nimi setkal. Přestože detaily konstrukce a konfiguračních skriptů jsou skryty za uživatelsky přívětivým uživatelským rozhraním, jednotlivá IDE většinou mají svoji vlastní implementaci build systému. Často je takový build systém silně integrovaný s kompilátorem a jinými nástroji jako např. s debuggery. To umožňuje aplikovat jazykově-specifické optimalizace nebo usnadnit definici jednotlivých kroků, potřebných pro překlad. Důsledkem však často je, že se software stane závislým na tomto konkrétním IDE. Obecná praxe ukazuje, že tyto nástroje jsou vhodné pro dostatečně lokalizovaný vývoj, nehodí se ale

pro rozsáhlejší projekty, kde dochází k míchání různých technologií a různých přístupů a jsou nepraktické pro “buildy”, se kterým autoři IDE nepočítali.

2.1.4 Správa konfigurací softwaru

Správa konfigurací softwaru (SCM – Software Configuration management) se stará o kontrolu nad evolucí komplexních systému. Tento úkol sahá daleko za hranice obecného build systému, tak jak bude popsán v následující části, protože build systém obecně uvažuje pouze jednu verzi daného softwaru (tu, kterou pravě konstruuje v dané běhové instanci). V reálném prostředí mnoho vývojářů současně přispívá do zdrojových kódů daného projektu a nějaký druh správy verzí či konfigurací musí být použit pro správné zvládnutí této situace. Tato správa verzí udržuje historii všech změn v souborech zdrojových kódů (kterým se zde říká revize), umožňuje vytváření mnoha nezávislých větví v software, poskytuje prostředky a nástroje pro slévání různých větví a pro porovnávání změn, udržuje meta data tykající se toho, kdo změny udělal, jaký byl jejich účel atd. Jako důsledek, vývojáři často potřebují míchat dohromady různé části kódu patřící do různých revizí či větví, protože např. aktuální verze zdrojového kódu přestala být stabilní. Na druhé straně, výsledky procesu konstrukce software, který zajišťuje build systém tzn. release verze binárních souborů, knihoven atd. musí být uloženy v systému správy konfigurací také pro účely pozdější údržby či opravy závad.

2.1.5 Packaging

Packaging softwaru sestává z vytváření instalačních balíčků, instalátorů, instalace zkompilovaných binárních souborů atp. Jeho účelem je zabalení zdrojových a zkompilovaných entit do jistého druhu abstrakce (balíčku), která poskytuje jasně danou a srozumitelnou formu pro deklarování identity, konfiguračního rozhraní, závislostí, verzí atd. Instalace takového balíčku či instalátoru pak zahrnuje vyřešení všech závislostí na jiných balíčcích (případně spuštění instalace požadovaných balíčků), aplikaci konfigurace, specifické pro cílovou instalační platformu, případně překlad entit, které jsou v balíčku distribuovány jako zdrojové kódy. Z teoretického pohledu na build systém jakožto na závislostní systém není rozdíl mezi vytvářením těchto instalačních balíčků a např. kompilací zdrojových kódů. Jako cíl zde figuruje balíček místo např. spustitelného souboru a místo souborů se zdrojovými kódy jako prerekvizitami zde figurují např. binární soubory.

2.1.6 Deployment

Deployment je proces uvedení softwaru do prostředí ve kterém je provozován. Může se jednat od prostého zkopírování souborů na cílový stroj přes instalace balíčků či instalátorů až po sofistikované manuální instalace, kde deployment zajišťuje speciální tým lidí a řídí ho s ohledem na potřeby zákazníka (SAP). I zde je vazba mezi tímto procesem a build systémem. Často může být proces samotného deploymentu definován v rámci build systému. Jako příklad lze uvést situaci, kdy firma vyvíjí software, ale uživatelům je zprostředkován jako služba a ne jako softwarová distribuce. Typicky se jedná o různé serverové služby, webové servery apod.
V takovém případě nemá smysl definovat rozhraní mezi vývojářem a uživatelem v podobě balíčku či instalátoru, místo toho je aplikace přímo umístěna na interní produkční stroj, který typicky bývá jen jeden (nebereme li v úvahu zálohy), pomocí deploymentu definovaném v build systému. Jiným příkladem by mohla být situace, kdy v rámci softwarového projektu jsou definovány testovací stroje, na kterých se zkouší správná funkčnost software v pseudo-reálném prostředí, přičemž vývojáři na takovýchto strojích testují nové komponenty či změny ve zdrojovém kódu. Bylo by poněkud neefektivní při každé změně v kódu procházet zdlouhavou instalační procedurou, která je definována pro deployment na straně uživatele. Místo toho je možné definovat zjednodušený deployment (např. na úrovni kopírovaní jednotlivých souborů) přímo v build systému, což umožní vývojářům okamžitou propagaci změny, kterou provedli až na samotný testovací stroj.

2.1.7 Release Management

Deployment softwaru je úzce spjat s release managementem, který má za úkol nakonfigurovat a také spravovat produkt dostatečně flexibilně na to, aby bylo (a to i zpětně) možné zjistit, jaké komponenty, zdrojové moduly či verze byly zabaleny do instalovaného nebo již nainstalovaného balíčku a minimalizovat tak například potřebu častých upgradů softwaru. Release management v podstatě definuje rozhraní mezi producentem softwaru a jeho konzumentem tzn., podle jakého schémata se bude software distribuovat, jak budou prováděny inkrementální updaty, jakým způsobem bude uživatel software přidávat do produktu nové featury, jakým způsobem bude poskytována podpora apod. Release management v komerčním prostředí je značně odlišný od release managementu v Open source. V open source se často software distribuuje jako zdrojová distribuce (balíček zdrojových kódů) a release management tak končí vydáním nové verze této zdrojové distribuce, přičemž přidávání či odebírání nových featur je řízeno na úrovni překladu zdrojových kódů této zdrojové distribuce. V komerčním prostředí je situace daleko složitější. Existuje mnoho různých strategií jak řídit release management a kolem tohoto tématu je vybudována rozsáhlá teorie. Ať už je to tedy release management v Open Source

nebo v komerční sféře v obou případech je potřeba poměrně silná spolupráce s build systémem.

2.1.8 Správa variant

Jak deployment, tak release management silně závisí na možnosti dostatečně flexibilně specifikovat různé konfigurace (varianty) na úrovni konstrukce softwaru tak, aby bylo možné dostatečně robustně, a efektivně zvládnout obrovskou variabilitu, která je často nedílnou součástí velkých softwarových projektů. Jak bylo řečeno v předcházejícím bodě v Open Source se toto často řeší na úrovni kompilace zdrojových kódů, respektive na úrovni konfigurace zdrojové distribuce (např. parametry pro GNU configure). V komerční sféře je situace daleko složitější. Software se často nedistribuuje jako balíček zdrojových kódů, což implikuje nemožnost přenést na stranu uživatele všechny možné existující verze, které můžou ze zdrojových kódů softwaru vzniknout (v Open Source se k uživateli dostává "zdroj" všech možných existujících variant v podobě samotného zdrojového kódu a uživatel si může sám vygenerovat takovou variantu, která uspokojí jeho potřeby). Distribuována je tedy jen konkrétní varianta a velká pozornost se musí věnovat tomu, aby se k uživateli dostala varianta, která dokáže uspokojit všechny jeho potřeby. To implikuje potřebu vytvářet a spravovat různé varianty daného softwaru a vytvářet oficiální produktové řady tak, aby byly uspokojeny potřeby všech tříd uživatelů. Zajímavé je, že z pohledu build systému jde, co se týče komerčního software o značné zjednodušení – není potřeba, aby build systém počítal s extremní variabilitou různých prostředí, na kterých by měl build systém fungovat tak jako je tomu třeba v Open Source a co bývá často zdrojem velkých problémů (portabilita build systému). Na druhou stranu to však klade zvýšené nároky na management softwarového projektu, protože je potřeba vytvořit dobře definované produktové řady tak, aby byly uspokojeny potřeby všech uživatelů.

2.2 Přehled nástrojů pro řízení překladu

Výběr vhodného nástroje pro řízení překladu je první a základní krok při implementaci build systému. Výběr ovlivňuje výslednou podobu systému i to, jak se bude systém používat. Je proto důležité znát dostupné alternativy tak, aby je bylo při návrhu build systému možno porovnat a vybrat nejvhodnější.

Následující část poskytuje stručný přehled nástrojů pro řízení překladu, které se v současnosti používají v praxi. Zvláštní pozornost je věnovaná nástroji GNU Make, respektive GNU build systému. Je to nástroj, který se v historii vývoje software objevil jako první a v současnosti je stále nejrozšířenější. Dále nástroji SCons, protože tento nástroj bude použit pro návrh build systému, který prezentuje tato práce.

Kromě níže uvedených nástrojů existuje značné množství dalších, které jsou buď derivátem jednoho ze zmíněných, nebo jsou ve stádiu vývoje a pro firemní použití nejsou zatím dostatečně stabilní nebo se jejich používání v praxi příliš nerozšířilo.

2.2.1 Make & Autotools

GNU Make je nejstarší a dnes nejrozšířenější nástroj pro řízení překladu. Velkou částí ovlivnil podobu všech nástrojů, které vznikly poté a které jsou zmíněny v této kapitole. Z tohoto důvodu je pro jeho popis vyhrazena vlastní kapitola.

2.2.2 CMake

CMake je relativně rozšířený nástroj orientovaný na multiplatformní software, který funguje jako generátor nativního build systému. Po spuštění CMake vygeneruje nativní Visual Studio soubory, nativní makefile skripty pro GNU make, nmake skripty atp. a to podle přání uživatele a v závislosti na cílové platformě. Konfigurační subsystém je daleko jednotnější a funguje lépe než konfigurační systém poskytovaný nástroji GNU Autotools. CMake definuje svůj vlastní skriptovací jazyk, který je zpracovatelný na všech platformách, které CMake podporuje. To je nevýhodou, protože vývojář build systému, potažmo jeho uživatel je nucen učit se další "domain specific" jazyk, ale to je rozhodně přijatelnější než 3 až 4 "jazyky" potřebné pro zvládnutí GNU make a GNU Autotools.

Z výše uvedeného popisu vyplývá, že CMake neimplementuje ani nerealizuje samotný build proces, ale deleguje tuto činnost na nativní nástroj. To umožňuje vývojářům, kteří jsou zvyklí na konkrétní vývojové prostředí na konkrétní platformě dále používat toto prostředí bez potřeby seznamovat se s jiným nástrojem. Na druhou stranu takový přístup vnáší do build systému další úroveň abstrakce navíc (první je unifikovaná multiplatformní definice v doméně CMake a druhá je definice v rámci nativního nástroje). Tím navíc zůstává proces samotného překladu neunifikovaný napříč platformami, protože na každé platformě se používá nativní build systém. Vývojáři, kteří pracují na více než jedné platformě musí používat více než jeden build systém. Toto je některými vývojáři považováno jako výhoda a ve skutečnosti je to tato vlastnost, která odlišuje CMake od ostatních nástrojů pro řízení překladu.

Výhody:

- Multi-platformnost
- Automatická analýza závislostí zdrojových kódů pro C/C++
- Podpora pro spolupráci s MS Visual Studio
- Podpora pro cross-kompilaci
- Rychlost
- Využívaní nativních nástrojů pro řízení překladu
- Snadná integrace se zaběhnutými nástroji pro řízení překladu, a to i těmi, které jsou součástí IDE

Nevýhody:

- Závislost na nativních nástrojích pro řízení překladu
- Občas nemožnost snadno definovat potřebnou funkcionalitu z důvodu možnosti využití pouze průniku funkcionalit nativních nástrojů.
- Nízká podpora pro cross-kompilace (vychází částečně z povahy tohoto nástroje)

Jak bylo řečeno, CMake je rozšířený a jako projekty využívající tento nástroj pro implementaci svého build systému lze uvést např. projekt KDE nebo mySQL.

2.2.3 Jam

Jam je nástroj pro řízení překladu, který implementuje svůj vlastní, v jistém smyslu imperativní jazyk (skripty a definice cílů zůstávají pořád deklarativní). Jam funguje na podobném principu jako make v tom smyslu, že jeho fungování je založeno na spouštění příkazů, definovaných v pravidlech a iniciovaných průchodem jednotlivými závislostmi. Jam je minimalistický a definuje jen velmi malou, ale rozšiřitelnou sadu funkcí. I pro nejzákladnější úkoly, jako je vytvoření sdílené knihovny, je potřeba dodefinovat vlastní pravidla a při složitějších pravidlech nebo při složitějších požadavcích na build systém je možné narazit na problémy spojené s omezenou vyjadřovací schopností skriptovacího jazyka, použitého nástrojem Jam. Definice velké části základních úkonů, se kterými se ve vývoji software setkáváme, je však relativně snadná, deklarativní a výsledné build skripty jsou přehledné.

Za dobu existence Jamu se z tohoto nástroje odštěpilo množství klonů, které více či méně řeší nedostatky původního nástroje. Nejznámějšími klony jsou KJam, FTJam a Boost Jam.

Výhody:

- Minimalistický design
- Velmi rychlý
- Základní a pokročilejší úkony jdou definovat velmi snadno a přehledně
- Dobře definované rozhraní rozšiřitelnosti
- Multi-platformnost

Nevýhody:

- Vlastní jazyk, který je potřeba se naučit
- Expresivita jazyka a trochu nepřehledná syntaxe
- Mnoho klonů – rozštěpuje uživatelskou základnu

2.2.4 Ant

Ant je nástroj, který vnikl v rámci projektu Apache. Tento nastroj je implementovaný v Jave a jako build skripty využívá XML soubory a byl vytvořen s ohledem na potřeby projektů, psaných v Jave. Z podstaty XML je jazyk build skriptů stejně

jako u nástroje GNU make deklarativní a i základní princip, na kterém nástroj pracuje, je podobný jako u make.

Přestože je Ant zaměřený na Javu, umožňuje vytvářet tzv. custom tasky (definují se jako jednoduché java třídy), které umožňují upravit si Ant i pro potřeby jiných než Java projektů. Na internetu je běžně dostupná sada již hotových custom tasků pro často využívané úkony jako např. task pro generování c/c++ objektů.

Ant je velmi rozšířený a vznikla kolem něj velká komunita uživatelů i vývojářů a stal se základnou pro mnoho odvozených nástrojů, jako je např. MS Build – nástroj integrovaný do vývojového prostředí MS Visual Studio.

Diskutabilní částí tohoto nástroje je volba XML jako "jazyka", ve kterém se píší build skripty. Jedním problémem je, že XML je značně verbózní, což build skriptům ubírá na přehlednosti. Druhým a závažnějším problémem je, že XML je vhodný pro popis struktury a je čistě deklarativní. V reálném projektu je však často potřeba chovat se procedurálně a konstrukce, které by byly v normálním jazyku přirozené, se musí v XML složitě emulovat. Typickým příkladem je task <if> definovaný v jednom z přídavných balíčků pro Ant. Uživatel je pak často nucen přecházet mezi dvěma přístupy. Čistě deskriptivním při popisu cílů v build skriptech a čistě procedurálním při definovaní/programovaní tasků.

Principielně je Ant podobný utilitě Make a stejně jako tato utilita spouští jednotlivé tasky při průchodu pravidly (tasky) definujícími cíle a jejich závislosti.

Výhody:

- Rozšířenost, velká uživatelská základna
- Vynikající použitelnost pro java projekty
- Relativní standardizovanost API
- Častá integrace do vývojových prostředí (Eclipse, NetBeans)
- Dobrá použitelnost pro základní deskriptivní úkony

Nevýhody:

- Podpora jiných než Java projektů nebyla uvažována v začátcích, při designování tohoto nástroje
- Omezená expresivnost XML, verbózní syntaxe

2.2.5 SCons

SCons je nástroj implementovaný v jazyce Python a svojí filosofií a principy je značně odlišný od principů definovaných nástrojem make a přejatých ostatními nástroji zmíněnými výše. Popisu tohoto nástroje bude vyhrazena zvláštní kapitola.

2.3 Make, GNU Make a GNU Autotools

Make je standard v oblasti nástrojů pro řízení překladu a všechny ostatní nástroje jsou vůči němu srovnávány. Make je v základním principu nástroj, který kontroluje závislosti a může spouštět příkazy shellu na základě stavu, kdy jeden soubor je starší než jiný. Implementace tohoto nástroje existují na téměř každé platformě, avšak zdaleka nejrozšířenější je GNU Make.

Většina implementací obsahuje sadu defaultních pravidel, které umožňují snadný překlad pro základní typy projektů, jako jsou C,C++, FORTRAN,lex atd. Navíc můžou být specifikována pravidla, která rozšíří podporu pro jiné typy projektů, avšak tyto pravidla musí být často parametrizována v závislosti na platformě.

2.3.1 Princip fungování Make

Make funguje na principu zpětného průchodu stromem závislostí. Pro nějaký konkrétní cíl je každá jeho závislost vyšetřena rekurzivně až do takové hloubky, kdy cíl na této dané úrovni už nemá žádné další závislosti nebo všechny jeho závislosti už byly vyšetřeny. Jestliže cíl na dané úrovni neexistuje nebo je nějakou heuristickou metodou, jako jsou časové známky či kontrolní součty dokázáno, že některá ze závislostí daného cíle je novější, než tento cíl, pak nástroj spustí sérii úkolů na vygenerování tohoto cíle. Jestliže není definováno žádné pravidlo pro daný cíl, Make se tento cíl pokusí najít ve filesystému, pokud i to selže, systém vygeneruje chybu. Informace o tom, zda byl cíl rekonstruován nebo ne, je propagována rekurzivně nahoru ve stromu závislostí až k uživatelem definovanému cíli, který je buď také rekonstruován, prohlášen jako aktuální nebo v případě výskytu chyby je proces přerušen.

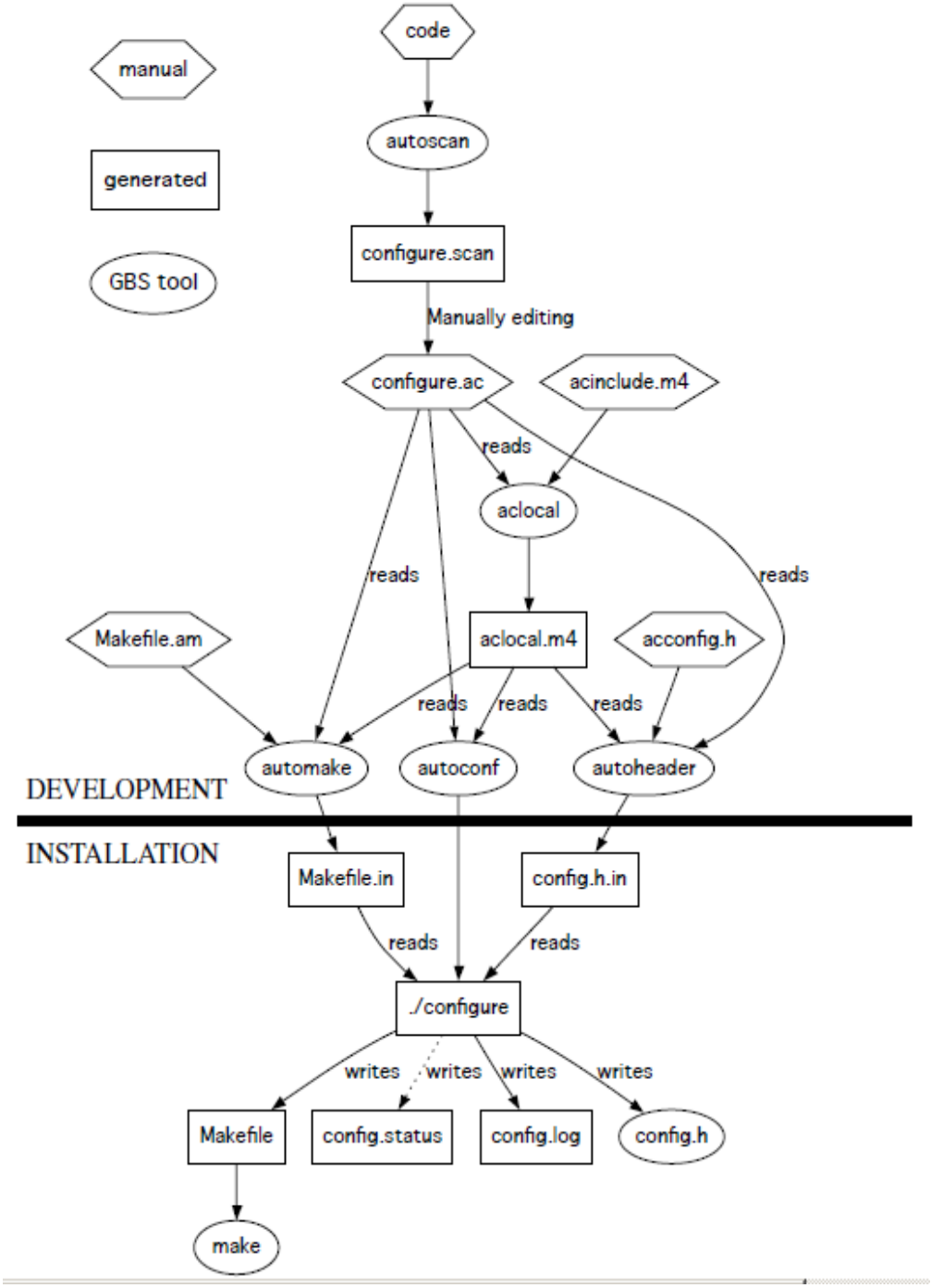
2.3.2 GNU Autotools

GNU Autotools rozšiřují GNU Make o rozsáhlejší knihovnu defaultních pravidel a navíc přidávají schopnost rozsáhlého testování externích závislostí a parametrizaci build skriptů v závislosti na tomto testování. GNU Autotools (resp. Autoconf) generují ‘configure‘ skript, který je přibalen do zdrojové distribuce softwarového projektu, takže uživatel, který překládá software, nemusí mít autotools nainstalován. Hlavními úkoly sady Autotools pak jsou:

- Zajistit, aby kód byl přenositelný skrz všechny platformy a operační systémy
- Usnadnit vývojářům a uživatelům psaní a používaní build skriptů tím, že vynucují GNU Coding standard

- Poskytnout platformově nezávislou instalaci ze zdrojové distribuce

Obrázek 1 poskytuje schematický náhled na hlavní komponenty systému Autotools a roli GNU make.



Obrázek 1

2.4 SCons

SCons je nástroj pro řízení překladu implementovaný v jazyku Python a používající Python pro své build skripty. Původně byl SCons inspirován podobným nástrojem Cons, který pro svoje build skripty využíval jazyk Perl. Na rozdíl od nástrojů GNU se SCons snaží o to, aby byl "self-contained" tzn., aby co nejméně závisel na dalších nástrojích a existujících externích utilitách.

Použití Pythonu, jako deskriptivního jazyka pro build skripty, je jednou z nejzajímavějších vlastností tohoto nástroje. Tento přístup je v přímém kontrastu s přístupem většiny ostatních nástrojů pro řízení překladu, které typicky definují nový jazyk pro popis build skriptu či pro popis konfigurace. Tento přístup také umožňuje sjednocení syntaxe pro definici build skriptů a syntaxe pro definici nových rozšíření množiny defaultních úkonů. Znamená to, že Python je použit, jak pro build skripty, tak pro rozšíření. To je rozdíl oproti většině ostatních nástrojů jako např. ANT, kde build skripty jsou popisovány pomocí XML a rozšíření pomocí Javy nebo nástroje Make, kde build skripty jsou psány ve speciálním jazyce, definovaném nástrojem Make a definice příkazů jsou psány v Shellu. Paradoxně použití Pythonu jako deskriptivního jazyka může znamenat zjednodušení i pro neprogramátory, protože Python byl navržen tak, aby jeho syntaxe byla jasná a čistá, a aby jeho základní konstrukce byly použitelné i pro neprogramátory (např. administrátory). Naopak jazyky, navržené nástroji pro řízení překladu speciálně pro své vlastní účely, bývají často nepřehledné, nedomyšlené nebo postrádající dostatečnou vyjadřovací schopnost či postrádající čistotu skutečného programovacího jazyka.

SCons je značně rozšířený a využívá se pro tvorbu build systémů pro rozsáhlé softwarové projekty. Jako příklad lze uvést webový prohlížeč Google Chrome od společnosti Google nebo virtualizační software VMware od stejnojmenné společnosti.

Výhody:

- Build skripty jsou psány v jazyce Python
- Veliká spolehlivost ve smyslu korektnosti buildu
- Zabudovaná automatická analýza implicitních závislostí
- Zabudovaná podpora pro velké množství různých typů jazyků a objektů od C až po Tex. Dobře definované rozhraní pro další rozšiřování.
- Zabudovaná podpora pro integraci s různými verzovacími nástroji (CVS, BitKeeper atd.)
- Zabudovaná podpora pro MS Visual Studio
- MD5 signatury pro podporu inkrementálních buildů
- Paralelní zpracování nezávislých větví
- Integrovaný konfigurační framework
- Vysoce multiplatformní

- Velká uživatelská základna
- Vynikající dokumentace

Nevýhody:

- Pro psaní pokročilejších funkcí je potřeba detailnější znalost jazyka Python. Občas je potřeba pro implementaci potřebné funkcionality psát vetší množství Python kódu.
- V porovnání s ostatními nástroji je SCons velmi pomalý, obzvláště u inkrementálních buildů (možno zapnout různé optimalizace, ale i s nimi je znatelně pomalejší než jiné nástroje)
- Sjednocení konfigurační a build vrstvy. Konfigurace je prováděna v rámci samotného buildu. Je možno manuálně dodefinovat toto oddělení.

2.4.1 Cíle

Vývojáři nástroje SCons definovali několik stěžejních bodů, které mají vystihnout hlavní ideje, stojící za vznikem tohoto nástroje, a které ho profilují ve velkém množství nástrojů, které jsou v této sféře k dispozici:

- **Korektnost** – SCons garantuje korektnost build procesu i kdyby to mělo znamenat částečné obětování výkonnosti. SCons se snaží garantovat korektnost bez ohledu na to, v jakém prostředí je software sestavován, jaké nástroje se používají, či jakým způsobem byl software napsán.
- **Výkon** – Jakmile je zajištěna korektnost, SCons se snaží sestavit software tak rychle, jak jen to je možné, bez zbytečných kroků a za pomocí všech dostupných technik optimalizace.
- **Pohodlí** – SCons se snaží ulehčit uživatelům co nejvíce věcí tím, že kde je to možné, tam poskytuje defaultně prostředky, které by si jinak musel uživatel vyvíjet sám.

2.4.2 Architektura

Klíčem k odkrytí všech vlastností nástroje SCons je pochopení jeho architektury a principů, které stojí v pozadí.

Hlavní funkcionalitou nástroje SCons je definování grafu závislostí AOG pomocí API, které SCons definuje pomocí jazyka Python. SCons definuje akce, které transformují zdrojové uzly do cílových uzlů. Jakmile je AOG zkonstruován, SCons engine začne procházet tento graf a ověří, zda jsou uzly aktuální za použití kontrolních součtů

jednotlivých souborů (v současnosti se používá MD5). Jestliže zjistí, že uzel není aktuální a je nutné jej znovu sestavit, engine přidá task na provedení akce definované pro sestavení neaktuálního cíle do fronty čekajících tasků. Scheduler pak spouští tasky ve správném pořadí a mimo jiné umožňuje spouštět nezávislé tasky paralelně.

SCons také definuje tzv. Scannery, které extrahují z množiny zdrojových entit implicitní závislosti. SCons defaultně obsahuje množinu předdefinovaných scannerů pro běžné objekty, jako např. scanner, který extrahuje implicitní závislosti na hlavičkových souborech ze zdrojových souborů jazyka C.

Jinou důležitou vlastností je zapouzdření celkového stavu prostředí (všechny faktory, které mají vliv na překlad daného cílového souboru), které je definováno pro daný cílový objekt do objektu jazyka Python. To pak umožňuje jeho kopírování, modifikaci, dědění atd.

2.4.2.1 Node subsystém

Uzly (třída Node) ve smyslu uzlů v doméně nástroje SCons představují interní reprezentaci stromu závislostí AOG a mapují tyto interní reprezentace na externí entity (např. soubory), které tyto reprezentace zastupují. Kromě souborů může Node reprezentovat např. adresáře nebo proměnné python.

Hlavním smyslem třídy Node je separovat správu závislostí od skutečných objektů, které instance třídy Node zastupují (soubory, adresáře, hodnoty). Řízení závislostí mezi jednotlivými entitami je implementováno v třídě Node. Specifické třídy reprezentující specifické objekty dědí z této třídy.

2.4.2.2 Builder subsytém

Buildery (třída Builder) představují jednak mechanismus, kterým se vlastní instance třídy Node vytvářejí a jednak mechanismus jakým se nastroj SCons rozšiřuje o nové typy cílových entit. Buildery slouží jako prostředek k vytvoření požadovaného cílového objektu.

Buildery definují, několik dalších entit, které dohromady tvoří framework pro práci se závislostmi. Třída Actions slouží k definování samotné akce, která je potřeba pro vygenerování daného cílového objektu v rámci daného builderu. Scannery slouží ke generování implicitních závislostí. Tzv. Emitery slouží k dynamické manipulaci se seznamy zdrojových a cílových entit pro daný builder.

2.4.2.3 Environment subsystém

Třída Environment v sobě zapouzdřuje celkový stav prostředí (všechny faktory, které mají vliv na překlad daného cílového souboru). SCons přiřazuje instance této třídy jednotlivým builderům. To definuje v jakém prostředí se cílový objekt daného builderu

sestaví. Environmenty se používají např. ke generování různých variant nějaké cílové entity, což je realizováno např. tím, že dva různé environmenty obsahují jiné hodnoty pro nějakou proměnnou prostředí (např. CCFLAGS=-g pro debug prostředí).

Tím, že je Environment implementován jako třída jazyka Python, je možné provádět standardní operace nad třídami jako např. klonování pro účely vytvoření "mírně" odlišného prostředí.

2.4.2.4 Signatures subsystém

Pro potřeby inkrementálního překladu je nezbytné, aby SCons implementoval mechanismus, na základě kterého bude možno určit, že nějaká entita je neaktuální.

SCons toto řeší implementací signatur počítaných na základě obsahu daného objektu. Instanci třídy Node je přidělen MD5 součet, který je v průběhu zpracování uložen do databáze společně s časovou známkou a velikostí souboru. MD5 hash je také vypočítán i z instance třídy Action, která definuje příkaz, který sestavuje cílovou entitu. Výsledkem je to, že kdykoliv se změní soubor, příkaz nebo i prostředí, uzly, které jsou závislé na této entitě, budou aktualizovány.

Mechanismus rozhodování toho, co je potřeba aktualizovat, je implementován ve třídě Decider. SCons defaultně poskytuje dva Decidery, jeden využívá MD5 hashe a druhý používá standardní časové známky tak, jak jsou známé z utility GNU make, které jsou o třídu rychlejší než počítání MD5 součtů (ovšem na úkor spolehlivosti).

2.4.2.5 Scanner

Scannery jsou zodpovědné za extrahování implicitních závislostí ze zdrojových souborů či entit. Pokud je danému souboru respektive jeho reprezentaci objektem Node asociován scanner, pak tento scanner zparsuje (nebo provede jinou činnost) soubor a vrátí seznam entit, na kterých tento soubor závisí. Navíc je možné, aby se tímto mechanismem scannery volaly rekurzivně.

2.4.2.6 Subsystém externích nástrojů

SCons defaultně disponuje poměrně rozsáhlou základnou znalostí, specifických pro danou platformu či pro danou sadu nástrojů jako jsou například kompilátory. Objekty třídy Tools slouží jako nosné třídy pro specifikaci těchto nástrojů. Obsahují základní informace o tom jak daný nástroj používat, jak jej spustit, kde se nachází, jaké jsou platformově specifické variace některých parametrů atd.

Tento subsystém značně zvyšuje uživatelský komfort, protože od uživatele přebírá zodpovědnost za definici způsobu používání běžně rozšířených a používaných nástrojů.

2.4.2.7 Scheduler

Subsystém, který obstarává spouštění jednotlivých tasků, se jmenuje Taskmaster. Tento subsystém zajišťuje, že operace spojené s jednotlivými objekty typu Node jsou vyhodnocovány ve správném pořadí a umožňuje, aby nezávislé větve stromu závislostí mohly byt vyhodnocovány paralelně.

Taskmaster jako takový běží v jednom threadu, ale jednotlivé tasky jsou spouštěny ve zvláštních threadech, tak aby bylo možno využít paralelismus.

2.4.2.8 Konfigurační subsystém

SCons poskytuje tzv. konfigurační kontext, který slouží k podobným účelům jako nástroj GNU Autoconf. Tento subsystém poskytuje základní sadu testů jako např. ověření přítomnosti hlavičkového souboru, ověření přítomnosti knihovny či specifického nainstalovaného balíčku. Tyto testy umožňují adaptaci build systému na konkrétní prostředí.

SCons navíc poskytuje dobře definované rozhraní pro vytváření vlastních testů, pomocí kterého může uživatel rozšířit sadu defaultních testů poskytovaných automaticky nástrojem SCons.

Detailní popis způsobu používání a vnitřního fungování nástroje SCons je mimo rámec této práce. Pro získání všech detailů proto tato práce odkazuje na manuál nástroje SCons, který je dostupný na stránkách projektu a dále předpokládá znalost tohoto nástroje.

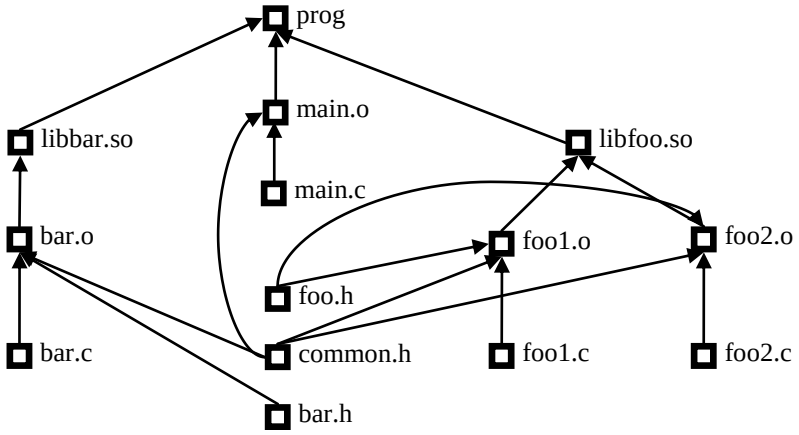
3 Analýza

Aby bylo možné navrhnout vhodný build systém a veškeré operace prováděné v rámci BS (Build Systému) byly dobře definované, je nezbytné identifikovat vlastnosti a pravidla, která platí v softwarových projektech. Tato kapitola analyzuje obecné rysy softwarových projektů a snaží se formalizovat aspekty mající vliv na návrh BS. Kapitola se snaží popisovat věci co nejobecněji tak, aby při navrhování reálného systému umožnila správný náhled a umožnila pokrýt obrovské množství složitých situací, které můžou v praxi nastat.

3.1 Softwarový projekt jako AOG

Z pohledu teorie grafů se dá na softwarový projekt nahlížet jako na orientovaný acyklický graf (AOG), kde jednotlivé uzly reprezentují entity a hrany reprezentují závislosti mezi nimi. Z tohoto úhlu pohledu je závislostí myšlena "sémantická" skutečnost vyjadřující vlastnost, že existence závislé entity je podmíněna existencí entity, na které je závislá. Obrázek 2 ukazuje typický graf závislostí pro malý softwarový projekt.

Pro správné chápání zbylého textu je důležité uvědomit si, že to co bude dále označováno jako AOG je statický závislostní graf. Dal by se přirovnat ke grafu, který by se získal zpětnou rekonstrukcí příkazů získaných z logu procesu překladu software.



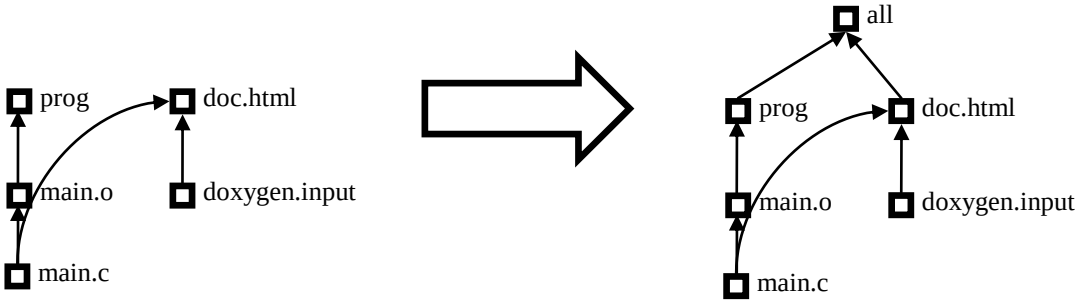
Obrázek 2

Entity (uzly v grafu) nemusí reprezentovat jen soubory, které mají svou reprezentaci ve filesystému, ale můžou reprezentovat například vykonání nějakého příkazu, nebo může jít o virtuální entity (phony targets v GBS), které můžou například sloužit k navázání jiných

závislostí na sebe tak, aby se skupina závislostí "sdružila" do jedné a bylo zaručeno, že všechny závislosti v této skupině budou vyřešeny společně.

Z obecné podstaty SP (Softwarových projektů) je výsledkem libovolného SP *software* čili sada počítačových programů, procedur, dokumentace atd., která provádí nějaký úkol na výpočetním stroji. Z úhlu pohledu, který je relevantní pro konstrukci BS je tedy možné říct, že SP je závislostní systém, kde hlavním požadovaným výsledkem je získaní finální entity, která je v AOG tohoto SP reprezentována uzlem, který netvoří závislost pro jiný uzel. V závislosti na rozsahu softwarového projektu respektive na rozsahu našeho "záběru" v rámci SP může jít v praxi například o ISO soubor, který se vypálí na DVD a distribuuje zákazníkovi nebo samoinstalační balíček pro různé instalační wizardy, spustitelný soubor, knihovna atp. Je dobré podotknout, že pro spoustu SP, a to obzvláště v OpenSource sféře, je z pohledu BS výsledkem sada binárních souborů (knihoven, spustitelných souborů atd.), tzn. že packaging se provádí nezávisle na build systému.

Takovýto finální objekt kromě jiného vykazuje tu vlastnost, že on sám není závislostí pro žádný jiný objekt v grafu. Pro teoretické účely se dá říct, že v grafu SP bude právě jeden takovýto objekt, pokud totiž strom obsahuje více takovýchto uzlů (např. když dokumentace tvoří samostatný balíček, který se nedistribuuje společně s programem) je možné jej uměle rozšířit tak, aby obsahoval právě jeden, viz obrázek 3.



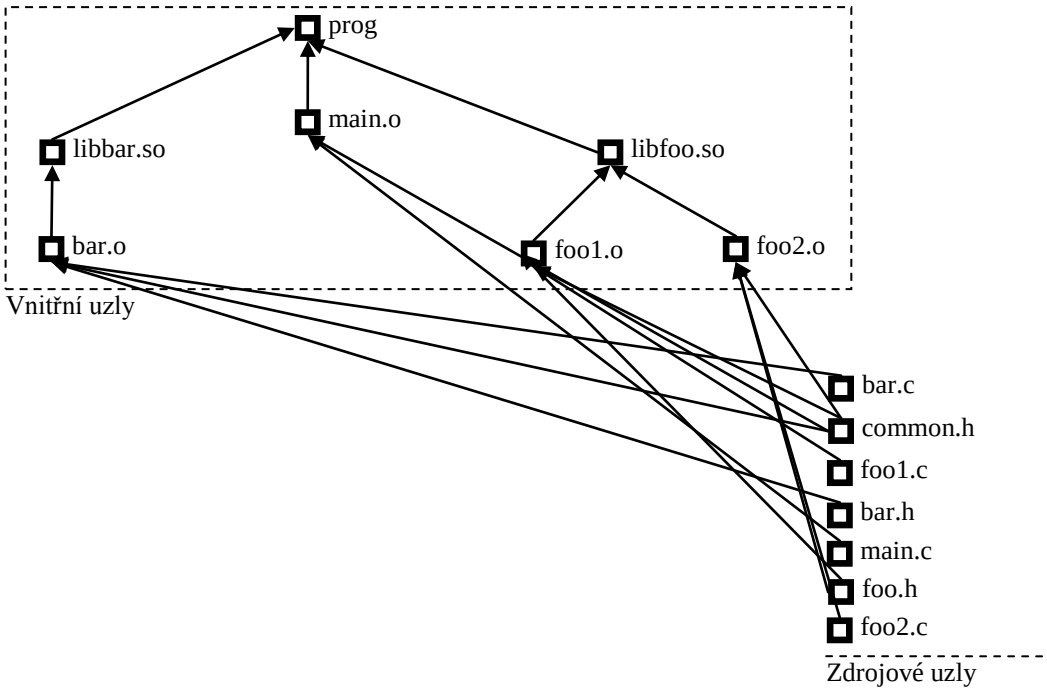
Obrázek 3

Důležitým atributem entit v AOG je to, zda v grafu figurují jako listy nebo jako vnitřní uzly. Obecně se dá říct, že listy jsou buď člověkem vytvořené a spravované soubory - například zdrojové kódy, textury, dokumentace, třídy objektů atp. (dále v textu se bude tento typ uzlů nazývat zdrojový uzel) nebo se jedná o externí závislosti projektu - nástroje pro překlad, hlavičkové soubory, knihovny či komponenty, SDK atd. Obecně se externí závislostí myslí libovolný objekt, který je potřebný pro SP, ale nespadá pod správu SP.

Listy, které nejsou externími závislostmi, mají tu důležitou vlastnost, že jsou to entity, které jsou manuálně spravovány člověkem. Toto pravidlo platí bez výjimky – kdyby

nebyly pod manuální správou člověka, znamenalo by to, že jsou modifikovány či vytvářeny nějakým nástrojem automaticky tzn., nebyly by listy.

Softwarové projekty se uchovávají v repositářích či jiných úložištích v nesestaveném stavu. V takovém počátečním stavu je software sadou člověkem vytvářených souborů. V ideálním případě je tato sada právě rovna množině všech listů (vyjma externích závislostí) v AOG softwarového projektu. Provedením posloupnosti různých operací se výše zmíněná sada počátečních objektů postupně transformuje na cílový finální objekt (bylo by možné říct, že se transformuje do sady cílových objektů, ale v rámci zjednodušení lze vždy AOG rozšířit tak, aby cílový objekt byl jen jeden, jak bylo vysvětleno výše). Všechny objekty, které vzniknou během této transformace, pak figurují jako vnitřní uzly v AOG. Jsou to všechny mezistupně mezi sadou počátečních objektů a finálním objektem tzn. objektové soubory, knihovny, soubory tříd apod.



Obrázek 4

3.2 Definování SP jako AOG – Build systém a build skripty

Vzhledem k definicím uvedeným v předchozí kapitole, je v čistě formálním smyslu build systém program, který definuje AOG a realizuje průchod tímto grafem až ke konečnému cílovému uzlu. Rozsah problémů, které musí BS řešit je natolik široký, že by bylo naprosto nemyslitelné, pro každý softwarový projekt vytvářet specializované programy v obecných jazycích jako je C nebo JAVA, Python atd. Navíc při pohledu napříč

softwarovými projekty a jejich BS se dá vypozorovat, že přestože spousta věcí se v jednotlivých build systémech liší, je zde množina podobných věcí a funkcionalit, které jsou přítomny ve všech BS nebo se v nich přinejmenším vyskytují poměrně často. Z těchto důvodů vznikly pomocné nástroje jako je GNU make implementující základní funkcionality, které jsou potřebné pro všechny BS a definují doménu (a případně jazyk) v níž BS implementován.

Běžně tedy implementace BS pro daný SP sestává z výběru nástroje pro řízení překladu a implementace BS v doméně tohoto nástroje. Nástroj pro řízení překladu realizuje průchod grafem AOG a druhý úkol BS - definice AOG je realizována pomocí definičních souborů vytvářených uživatelem v rámci domény NRP. Tyto soubory budou dále v textu označovány jako build skripty - je to tedy soubor, který definuje hrany a uzly v AOG.

U build skriptů je, stejně jako kdekoliv jinde v programování, dobré uplatnit princip divide et impera. Jak bude princip aplikován si lze představit na hypotetickém monolitickém build skriptu, který obsahuje kompletní definici AOG, který bude umístěn v kořeni adresářové struktury buildovacího prostoru. Tento kořen se bude dále v textu označovat jako build root a je definován jako taková cesta, že všechny perzistentní uzly vyjma externích závislostí a všechny build skripty se nacházejí v podstromu, jehož kořenem je tato cesta.

Mít kompletní definici AOG v jednom monolitickém build skriptu není nejlepší řešení. Znamenalo by to mít skript, obsahující u velkých projektů řádově tisíce řádků kódu, kde všechny definice uzlů a závislostí by byly referencovány vzhledem k build rootu, což by často znamenalo referencování dlouhých složitých cest v adresářové struktuře. Znamenalo by to také mnohem složitější údržbu. Kdyby například bylo potřeba odstranit nějaký modul (např. dále nevyužívanou knihovnu) ze softwarového projektu, musel by se monolitický build skript upravit a odstranit z něj definici odstraňovaného modulu, což je daleko složitější a více náchylné k uživatelským chybám, než prosté smazání build skriptu definujícího odstraňovanou knihovnu (v případě, kdy definice modulu je v samostatném build skriptu). Stejně jako je tomu u jakékoliv jiné organizace dat, všechny nevýhody centralizované monolitické architektury se projeví s rostoucí komplexností softwarového projektu.

3.3 Varianty

Problematika variant je netriviální záležitostí, která se značnou měrou podílí na komplexitě celého build systému, je proto důležité problém přesně vymezit a analyzovat.

Nejprve je dobré pochopit důkladně pojem varianty. Obecně se dá říct, a to i mimo rámec build systému a programování vůbec, že je-li A objekt (typ, výrobek, produkt, model, …) mající určité atributy, které determinují jeho chování a vlastnosti, pak varianta objektu A je objekt který se od objektu A odlišuje v některých atributech, ale většina atributů zůstává shodných s A.

Tato definice je trochu vágní v tom smyslu, že není určena jasná hranice mezi tím, kdy se objekt od svého originálního objektu odlišuje jen trochu a je možné ho považovat za

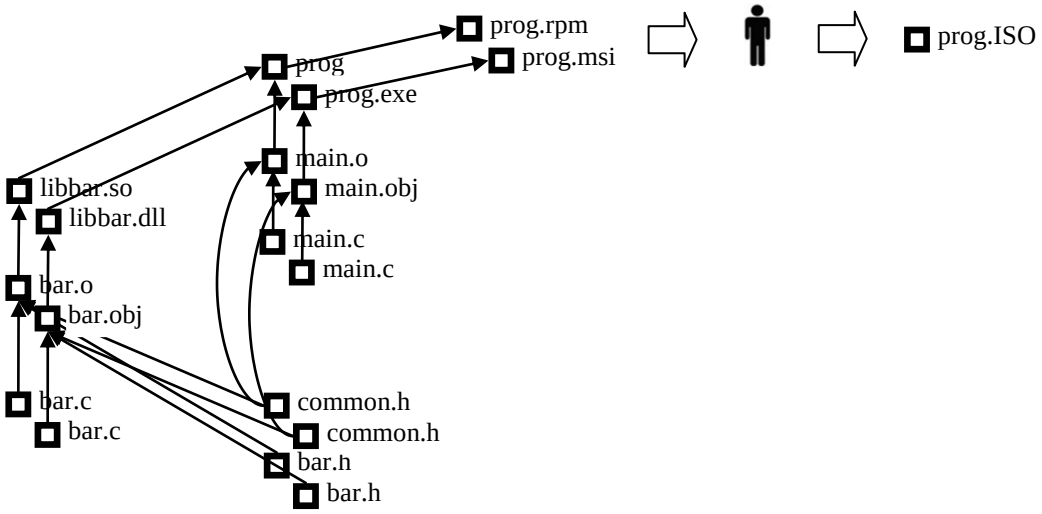
variantu a kdy už je rozdílnost těchto objektů natolik velká, že by jeden neměl být považován jako varianta druhého ale jako nezávislý objekt. V tomto smyslu je na osobním subjektivním rozhodnutí, kde je v daném kontextu hranice mezi tím, kdy je výhodné objekt považovat za variantu a kdy už by bylo lepší považovat ho za samostatný objekt.

Smysl variant je v tom, že není potřeba vytvářet speciální definice či procesy pro zacházení s variantou (varianta auta, které má oproti základní verzi navíc klimatizaci podléhá stejnému obchodnímu modelu, distribuční síti a marketingu jako základní verze).

Jak bylo řečeno v kapitole 3.2, graf AOG je definován pomocí build skriptů psaných v doméně nástroje pro řízení překladu. Můžeme předpokládat, že jazyk této domény je dostatečně silný na to, aby dokázal pracovat s proměnnými veličinami, které nabývají hodnot na základě uživatelského vstupu či prostředí. V opačném případě by takovýto BS byl v kterémkoliv jiném než triviálním SP prakticky nepoužitelný. Máme-li tedy build skripty psány v takovémto jazyce, bude se BS chovat v závislosti na vstupních parametrech, na tom, v jakém prostředí je spuštěn apod. Pro účely následujícího textu bude finální podoba AOG po zpracování všech build skriptů v závislosti na množině elementů, které mají vliv na toto zpracovávání (vstupní parametry, hardware, prostředí atd.) nazývaná Varianta.

Mějme nyní jako příklad software, který je podporován na platformě Solaris a Linux. Na Linuxu se distribuuje pomocí rpm balíčků a na Windows jako balíček msi. Tyto balíčky se však zákazníkovi distribuují jako jeden DVD image, který se použije i pro Windows i pro Linux. BS systém pro tento projekt je navržen tak, že po spuštění vygeneruje na dané platformě balíčky pro tuto platformu. Tzn., na Linuxu zkompiluje zdrojové kódy a vytvoří rpm balíček, na Windows zkompiluje kód a vytvoří msi. Pro vytvoření DVD obrazu je tedy nutné spustit build na Linuxu, poté na Windows, a poté vzít výsledné balíčky a vytvořit z nich DVD ISO soubor. Jinými slovy to znamená, že vytvoříme dvě varianty programu pomocí spuštění dvou iterací celého build procesu.

Nevýhody tohoto systému jsou vidět na první pohled. To, co skutečně chceme, je získat DVD obraz (finální uzel) z množiny zdrojových uzlů. Podíváme-li se však na tento ukázkový systém blíže, vidíme, že graf AOG na Linuxu je naprosto oddělený od grafu AOG na Windows a přechod od jednotlivých balíčků rpm a msi k DVD obrazu není v AOG pro změnu vůbec a musí se dělat manuálně (tento proces může být automatizován nějakým "vnějším" skriptem, ale to principielně na situaci nic nemění, protože graf závislostí zůstává porušený).



Obrázek 5 - varianty

sám obraz DVD byl součástí AOG a tudíž byl výsledkem realizace buildovacího procesu? Návrh build systému se v kapitole 4.6 pokusí nalézt řešení na tento problém stejně jako na celou problematiku variant.

3.4 Přístup Make vs. Přístup SCons

Cílem této kapitoly je analyzovat a porovnat dva odlišné přístupy k samotnému řízení překladu; tradiční přístup představený původním nástrojem make a později převzatý většinou ostatních nástrojů pro řízení překladu a přístup představený nástrojem SCons.

V kapitole 2.3 byl popsán princip, na jakém je založeno fungování nástroje make. V build skriptech (makefilech) se deklarují pravidla popisující, jak sestavit danou cílovou entitu. Make s cílem vytvořit požadovanou entitu rekurzivně prochází tyto pravidla na základě závislostí a postupně vytváří jednotlivé entity spouštěním příkazů definovaných v pravidlech. Na druhou stranu v doméně nástroje SCons se v build skriptech voláním API psaného v Pythonu postupně definuje virtuální podoba výsledného stromu. Až poté, co je sestaven kompletní strom závislostí, začne SCons engine tento strom procházet a konstruovat výsledné entity voláním patřičných příkazů.

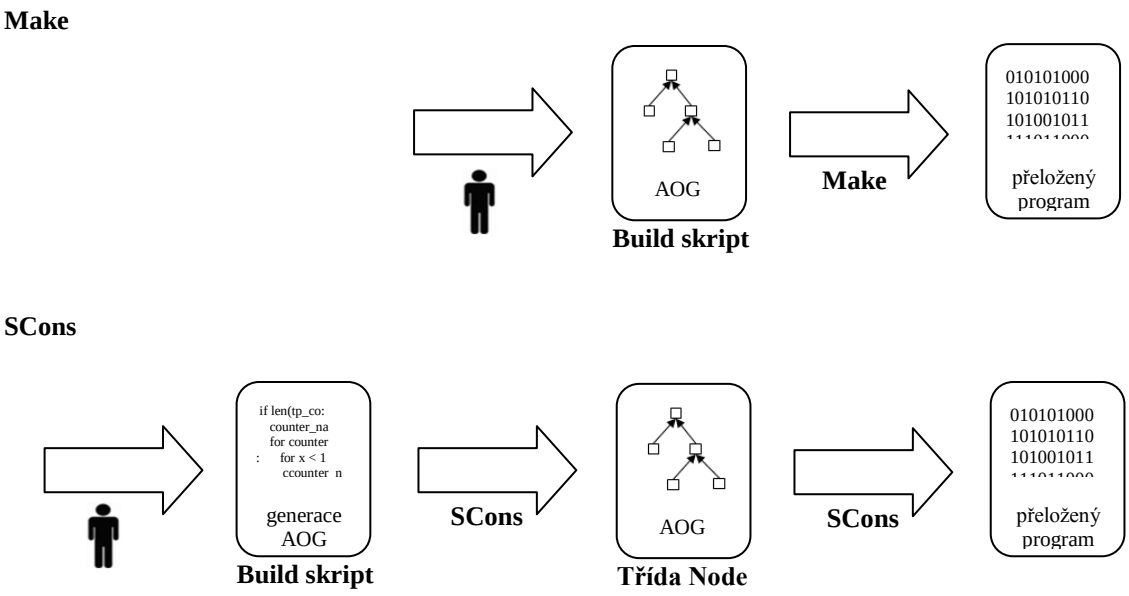
3.4.1 Popis struktury AOG

Strom AOG, jak byl popsán v kapitole 3.1 je statickým popisem závislostní struktury softwarového projektu a jako takový nemá nic společného se samotným procesem překladu a dynamickou povahou s ním spjatou.

Z tohoto pohledu je rozdíl v přístupu Make a SCons následující. Make (a všechny ostatní nástroje, které převzaly základní principy z nástroje Make) ve svých build skriptech římo popisuje podobu AOG ve smyslu definovaném kapitolou 3.1– jednotlivá pravidla v build skriptech figurují jako definice jednotlivých uzlů ve stromu AOG a seznamy prerekvizit definují hrany mezi uzly. Make engine pak jednoduše staticky vezme pravidla definovaná v build skriptech a okamžitě začne s překladem na základě těchto pravidel. Znamená to tedy, že u nástroje Make vytváří graf AOG uživatel manuálně tím, že definuje jednotlivá pravidla v build skriptech.

SCons naopak ve svých skriptech strukturu AOG vůbec nepopisuje. Build skripty nástroje SCons ve skutečnosti popisují samotnou konstrukci stromu AOG a vlastní AOG existuje jako interní datová struktura, vytvořená z instance třídy Node. To na rozdíl od nástroje Make znamená, že strukturu stromu AOG nepopisuje ani nevytváří uživatel, ale je vytvářena automaticky nástrojem SCons, přičemž uživatel v build skriptech definuje způsob, jakým se má tato struktura vytvářet.

Souhrnně se dá říct, že make v build skriptech popisuje strukturu AOG, zatímco SCons popisuje proces vytváření struktury AOG. Obrázek 6 přibližuje tyto dva odlišné přístupy.



Obrázek 6 - Make vs. SCons

3.4.2 Dynamická povaha build systému

Vzhledem ke skutečnosti, že v reálném použití je proces překladu ovlivňován množstvím různých faktorů (uživatelské vstupy, prostředí, proměnné, platforma atd.), není možné dopředu znát přesnou podobu AOG v takovém smyslu, v jakém byl popsán v kapitole 3.1.

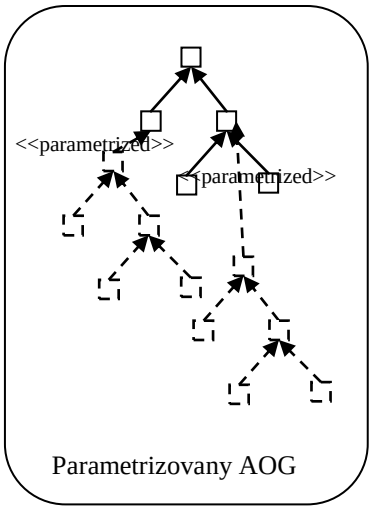
Make toto řeší komplexním systémem parametrizování pravidel, preprocessingem a makry. Je třeba si uvědomit (jak bylo ukázáno v předchozí kapitole), že build skripty nástroje Make figurují jako popis struktury AOG, a je to proto samotný popis AOG, co je zde parametrizováno a preprocesováno.

Tyto mechanismy znepřehledňují build skripty a celou deklarativní povahu nástroje. Kromě toho, že to přináší další vrstvu abstrakce navíc, tak se pomocí těchto mechanismů často řeší kontrola toku programu (např. podmíněné větve, GNU make implementuje speciální makra pro podmíněnou transformaci, např. *ifeq*).

Z těchto důvodů trpí make stejným problémem, kterým trpí velká spousta deklarativních jazyků, kde jazyk je navržen v čistě deklarativním duchu, ale postupně, jak se potřeby uživatelů rozšiřují, jsou do jazyka přidávány procedurální elementy a výsledná směsice těchto přístupů pak působí nekonzistentně.

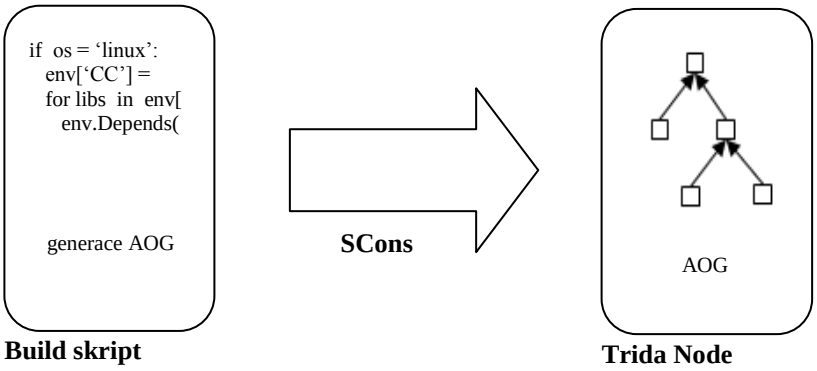
Přístup nástroje SCons je daleko přirozenější. Tím, že je samotný popis struktury AOG oddělený od generování tohoto popisu, není nutné konfrontovat čistě statickou povahu struktury AOG s procedurálními prvky jako u nástroje Make. Veškeré rozhodování je přesunuto do úrovně generování popisu struktury AOG, které je implementováno v build skriptech, využívajících jazyk Python, který jako imperativní jazyk poskytuje dostatečně silné nástroje k přirozenému zvládnutí dynamických a procedurálních prvků build systému.

Make



Build script

SCons



Build skript **Trida Node**

Obrázek 7 - parametrizace Make vs. SCons

3.4.3 Deklarativní vs. imperativní

Obecně se dá říct, že za ideálních podmínek je deklarativní přístup vždy lepší než imperativní. Deklarativní je přirozenější pro uživatele, není potřeba vůbec uvažovat o řízení toku a o dalších detailech procedurálního zpracování programu. Uživatel je pak nucen implementovat jen "to zajímavé". Ke všemu platí, že imperativní přístup vyžaduje jistou dávku programátorského způsobu myšlení, který není pro neprogramátory přirozený, naopak deklarativní přístup může být srozumitelný i pro neprogramátory.

V oblasti build systémů se bohužel ukazuje, že problém, který build systémy řeší, je příliš komplexní, než aby byl jednoduše popsatelný deklarativním jazykem, bez toho, aby tento jazyk nebyl kompromitován procedurálními vlastnostmi.

Build skripty u nástroje Make jsou deklarativní. S deklarativností build skriptů nástroje SCons už je to komplikovanější. Vzhledem k následující citaci z uživatelské příručky k nástroji SCons, považují autoři tohoto nástroje build skripty jako deklarativní

„In programming parlance, the SConstruct file is declarative, meaning you tell SCons what you want done and let it figure out the order in which to do it, rather than strictly imperative, where you specify explicitly the order in which to do things."(Knight and Roach, 2004)

Je pravda, že pořadí jednotlivých volaní v build skriptech neovlivní to, v jakém pořadí se budou volat jednotlivé příkazy pro konstrukci cílových objektů. To je ale z toho důvodu, že build skripty nástroje SCons nemají na starosti řízení překladu tak, jako je tomu u Make, ale spíš mají na starosti generování popisu stromu AOG tak, jak bylo popsáno výše. Samotný překlad jde mimo build skripty a je řízen zcela jiným subsystémem (Taskmaster subsystém). Z tohoto důvodu není rozumné, považovat tyto build skripty za deklarativní čistě na základě skutečnosti, že volání jednotlivých funkcí neovlivňuje pořadí, v jakém jsou jednotlivé entity přeloženy.

Uživatel nástroje SCons nepřichází se subsystémem, který řídí samotný překlad a spouští jednotlivé příkazy vůbec do styku a to ani skrze interaktivní přístup, ani skrze psaní build skriptů. To, s čím uživatel přichází do styku je generování struktury AOG. Toto, jak bude ukázáno v následujícím textu, je skutečně spíše imperativní. Vzhledem k tomu, že toto generování je to, co uživatel implementuje v jednotlivých build skriptech, jsou build skripty spíše imperativní než deklarativní, což je v přímém rozporu s tím co tvrdí uživatelská příručka nástroje SCons.

Aby build skripty nástroje SCons mohly být považovány za deklarativní, musely by umožňovat definování jednotlivých částí stromu AOG zcela nezávisle na pořadí, v jakém jsou volány jednotlivé API funkce, které jednotlivé části AOG definují. Stačí se však podívat do manuálu nástroje SCons na základní příklad použití pro překlad C knihovny že tomu tak u nástroje SCons není.

obj = env.Object('foo.c')
env.Library('prog', obj)

Výše uvedená ukázka je standardní python kód a ted obj je proměnná jazyka python a musí být nejprve vytvořena na řádce 1, než může být použita na řádce 2 ve volání builderu Library. S trochou úsilí by šel kód do deklarativní podoby přepsat následovně.

env.Library('prog','foo.o')
env.Object('foo.c')

V tomto případe je sice pravda, že takovýto build skript je deklarativní a bude fungovat, problémem však je, že vnitřní uzel *foo* je na řádku 1 specifikován staticky pomocí cesty v adresářové struktuře. To není správný přístup, jak bude vysvětleno v kapitole 4.5.2, protože uživatel musí hlídat veškeré implementační detaily jako např. jaká koncovka souboru či jaký prefix se implicitně používá na dané platformě, musí znát přesnou cestu k souboru atd.

Samotný manuál nástroje SCons potvrzuje, že používání proměnných jazyka python je preferovaný způsob.

Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(['hello.o', 'goodbye.o'])

„The problem with listing the names as strings is that our SConstruct file is no longer portable across operating systems. It won't, for example, work on Windows because the object files there would be named hello.obj and goodbye.obj, not hello.o and goodbye.o.A better solution is to assign the lists of targets returned by the calls to the Object builder to variables, which we can then concatenate in our call to the Program builder:" (Knight and Roach, 2004)

hello_list = Object('hello.c', CCFLAGS='-DHELLO')
goodbye_list = Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(hello_list + goodbye_list)"

Ještě více je tento problém patrný, pokud jsou jednotlivé související entity definovány v různých build skriptech. SCons poskytuje globální pohled na celý AOG (tzn. všechny entity ve všech build skriptech jsou definovány v rámci stejného "scope", narozdíl od Make, kde scope závislostí je vždy jen v rámci jednoho souboru build skriptu) a aby splnil podmínky deklarativnosti musel by poskytnout prostředek, jak deklarativně "propojit" tyto související entity nacházející se v různých build skriptech.

Imperativní charakter build skriptů nástroje SCons je tedy zřejmý a skutečně by se mělo k tomuto nástroji přistupovat s vědomím této skutečnosti. Neuváženou snahou

aplikovat deklarativní principy v build skriptech dochází často k znepřehlednění a zesložitění celého systému.

4 Návrh

Tato kapitola popisuje návrh build systému, určeného pro komerční softwarový projekt. Je vyvíjen interně uvnitř softwarové společnosti, jeho zdrojové kódy jsou uzavřené, je distribuován v binární podobě a podporuje několik platforem (distribuuje se několik produktových řad). Návrh bude popisován v kontextu, definovaném předchozími kapitolami. Prezentován bude formou nalezení řešení pro stěžejní problémy, na kterých stojí podoba build systému v reálné praxi a formou vybrání a obhajoby konkrétních řešení z množiny alternativních řešení.

4.1 Výběr vhodného nástroje

Kapitola 2.2 přinesla stručný přehled nejpoužívanějších nástrojů pro řízení překladu včetně stručných seznamů jejich výhod a nevýhod. V kapitole 3.4 byly analyzovány některé aspekty nástrojů pro řízení překladu. Analýza se věnovala především srovnání dvou základních přístupů k řešení problému závislostí.

Na základě těchto údajů bude pro build systém navrhovaný touto prací použit nástroj SCons. Ten vým charakterem umožňuje přirozeněji popsat všechny složité situace, které můžou v rozsáhlých softwarových projektech nastat. Použití jazyka Python je výhodou, protože ruší potřebu učit se specifický jazyk čistě pro potřeby popisu překladu.

Z definice třídy projektů, pro které je build systém navrhován je multi-platformnost naprosto stěžejním bodem, který musí být brán v úvahu při výběru nástroje pro řízení překladu. Podpora multi-platformnosti je u nástroje SCons na velmi vysoké úrovni – SCons pracuje na všech platformách, na kterých je podporovaný interpret jazyka Python. Podpora jazyka sahá od systému Windows až po značně minoritní UNIXové systémy jako např. Tru64 (DEC). Navíc má SCons zabudovanou schopnost automatické detekce standardních vývojových nástrojů na daných platformách a ví, jak tyto nástroje správně používat. (Tzn., jak je najít, jak je spouštět, jaké parametry jim poskytovat v závislosti na úkonu, který uživatel požaduje. Tato schopnost např. zahrnuje nástroje vývojového prostředí MS Visual Studio na platformě MS Windows, Sun Studio na platformě Solaris, XL C/C++ Enterprise na platformě AIX atd. a nástroje projektu GNU na všech jeho podporovaných platformách.

To šetří spousty času, jak uživatelům, tak především vývojářům build systému, protože lidé v komerčním prostředí často přicházejí (a opouštějí) k softwarovému projektu bez detailní znalosti některé z vývojových platforem, a přesto jsou nuceni na ní vyvíjet. Je proto vhodné, tam kde je to možné, přenechat platformově specifické rozhodování na inteligentním nástroji - samozřejmě s možností delegovat veškeré rozhodování zpět na uživatele tak, aby zkušený vývojář měl možnost změnit nevyhovující defaultní chování či

rozhodování. SCons všechny tyto požadavky splňuje a je proto ideálním nástrojem pro komerční multiplatformní projekt.

Protože s komerčním vývojem přichází velká zodpovědnost nejenom za lidi ale i za zisky a celkovou úspěšnost a životaschopnost projektu, jsou veškeré technologie použité v softwarovém projektu důkladně vybírány tak, aby splňovaly několik základních kriterií.

Technologie, použitá pro softwarový projekt, musí být dostatečně stabilní, aby její používání nevyvolalo množství technických problémů a aby časté změny v rozhraní této technologie nezpůsobily nekompatibility a nezkomplikovaly vlastní vývoj software.

Dále by technologie měla být rozšířená a adoptovaná jinými společnostmi, jednotlivci a softwarovými projekty. Dostatečná uživatelská základna slouží jako důkaz toho, že je technologie použitelná v praxi a zajišťuje, že je životaschopná a rychle nezanikne. To by při adoptování takové technologie negativně zasáhlo vývoj produktu. Široká uživatelská základna s sebou také přináší širokou základnu znalostí týkajících se technologie. To má nezanedbatelný vliv na použitelnost a efektivitu práce s touto technologií v praxi.

Je také důležité, aby současní vývojáři technologie poskytovali dostatečnou podporu, řešili problémy, které se v dané technologii objeví, vydávali záplaty, přidávali nové vlastnosti atd.

SCons splňuje i tyto požadavky. V roce 2008 byla po dlouhém období, kdy byl SCons označován jako beta verze s označením menším než 1, vydána stabilní verze 1.0 a do dnešní doby se vývoj dostal k verzi 1.2. Mnoho velkých komerčních projektů používá SCons jako základ pro svůj build systém. Jmenovitě se jedná např. o Google Chrome.

4.2 Vývojové prostředí

Jedním z hlavních rozdílů mezi Open Source softwarem a komerčně vyvíjeným softwarem je způsob, jakým tyto dvě odlišné sféry přistupují k distribuci software. Open source ze své podstaty umožňuje komukoliv nahlížet do zdrojových kódů a většinou poskytuje prostředky, jak tyto zdrojové kódy přeložit.

V praxi to znamená, že Open Source software je ve většině případů distribuován jako balík zdrojových kodů. Společně s build skripty a konfiguračními skripty je může uživatel použít k sestavení výsledného software na svém vlastním počítači (pokud tento počítač splňuje všechny prerekvizity stanovené softwarovým projektem).

To znamená, že Build systém musí být schopen korektně přeložit software na obrovském množství různých platforem a systémových konfiguracích. Systémy, na kterých se poté software překládá, se můžou lišit např. ve verzích či přítomností knihoven a programů nebo v jiných konfiguracích, jako je architektrura procesoru, bitovost, přítomnost nástrojů atd. S tím vším se musí vypořádat konfigurační vrstva build systému. Toto bývá nejnáročnějším a současně nejslabším místem build systému (a to právě z důvodu obrovské variability různých konfigurací).

Typický zástupce komerčně vyvíjeného softwarového projektu je však distribuován v přeložené podobě a zdrojové kódy se nedistribuují. Není proto potřeba vynakládat velké úsilí na to, aby konfigurační vrstva zajišťovala, že software půjde přeložit na systémech mimo vývojové prostředí společnosti. To ušetří spousty námahy a celý problém konfigurací se dá vyřešit deklarováním, že build systém musí zajišťovat přeložitelnost pouze pro vývojové stroje, definované v rámci softwarového projektu. Vývojové prostředí je tedy statické, proto je nezbytné strukturu tohoto prostředí formalizovat a dobře definovat. Formalizace konfigurační vrstvy build systému je jediným pilířem, o který se vývojář může opřít. Build skripty samotné se o konfiguraci prostředí nestarají.

4.2.1 Struktura vývojového prostředí

Vývojové prostředí pro navrhovaný build systém se skládá z několika tříd strojů, které dohromady utváří strukturu optimalizovanou pro vývoj požadovaného typu softwarových projektů.

4.2.1.1 Source server

Centrálním prvkem struktury vývojového prostředí je source server. Source server je stroj, který slouží jako centrální úložiště zdrojových kódů, ale hlavně source server řídí překlad. Source server **je jediný stroj, který provádí výpočet stromu závislostí, řídí spouštění všech příkazů. Je to jediný stroj, s kterým interaguje uživatel při překladu softwaru. Tento stroj při spuštění přrekladu vytváří a v paměti udžuje kompletní strom závislostí všech platforem.** Source server zná síťové adresy build strojů, a kdykoliv potřebuje provést platformově závislý překlad, požádá build server pro danou platformu o překlad konkrétního artefaktu (např. ho požádá o překlad souboru foo.c na foo.o). Každý vývojář zde má svůj pracovní adresář, kam si může uložit snapshot požadované verze zdrojových kódů, nejčastěji získané z repositáře systému pro správu verzí (CVS, SVN, atd.). Tyto adresáře jsou exportovány např. pomocí NFS nebo protokolu SMB, takže můžou být připojeny k jednotlivým build strojům. Díky tomu jsou zdrojové kódy sdílené mezi všemi buildovacími stroji a změny provedené v těchto zdrojových kódech jsou konzistentní napříč všemi buildovacími stroji.

4.2.1.2 Build server

Buildovací stroje jsou systémy sloužící pro realizaci samotného překladu. Typicky jsou dedikovány čistě pro potřeby spouštění buildů a generování výsledných souborů. **Buildovací stroje neví nic o závislostech ani build systému a jejich jediným úkolem je vykonávat platformově specifické příkazy, spouštěné vzdáleně source serverem.**

Významnou definující podmínkou pro navrhovaný build systém je skutečnost, že daný buildovací stroj může být pro danou platformu **jediným dostupným strojem v rámci dané organizace**, ať už z důvodu finančního nebo z důvodu problematického shánění dané platformy. Jinými slovy scénář, kdy každý vývojář dělá kompletní cyklus vývoje na své vývojové stanici není realizovatelný. Jako příklad lze uvést např. platformu tru64, která se v dnešní době velmi problematicky shání, případně AIX, kde i základní verze systému architektury PowerPC je značně finančně nákladná.

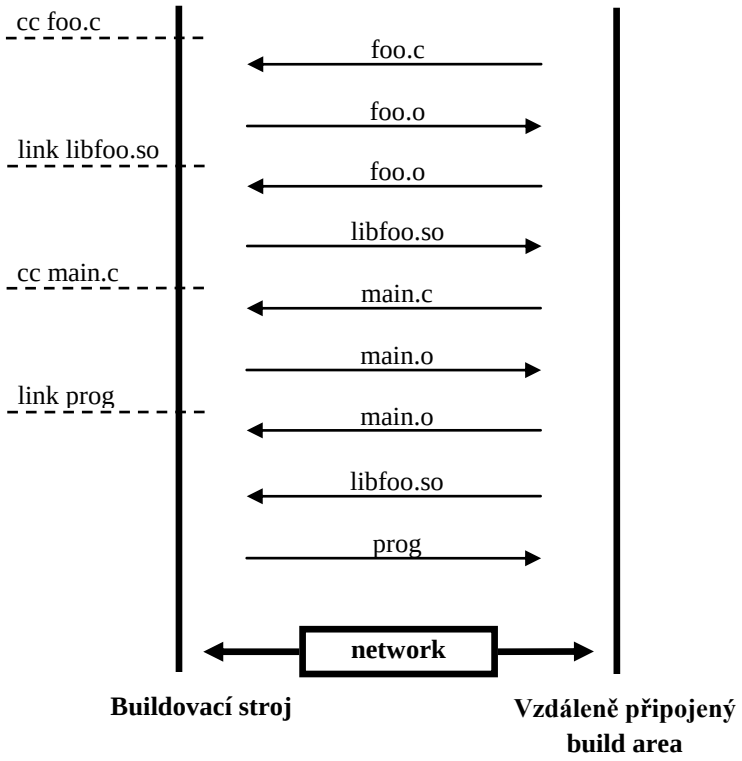
Z těchto důvodů je buildovací stroj sdílen mezi všemi vývojáři a případné porušení jeho integrity by mohlo ohrozit celý vývojový tým. Musí být tedy zajištěno, aby buildovací stroje byly stabilní, minimalizovat případné výpadky a především minimalizovat zásahy do konfigurace systému, protože ty skrývají největší nebezpečí toho, že by se mohla narušit integrita daného stroje. Důsledkem je, že buildovací stroj nemůže sloužit jako systém pro testování a instalování ladících verzí programů apod.

Každá podporovaná platforma má tedy svůj oficiální buildovací stroj. Zdrojové kódy jsou připojeny ze vzdáleného source serveru na každý build systém tak, aby byla zachována konzistence těchto kódů. Na každém buildovacím stroji je vytvořen adresář, který slouží jako tzv. build area. Všechny výstupní soubory build procesu se ukládají v adresářové struktuře v rámci build area.

Proces překladu zahrnuje generování různých souborů a binárních objektů a tyto objekty mezi sebou interagují. Při překladu C knihovny je např. potřeba z jednotlivých zdrojových souborů vygenerovat objektové soubory. Výsledná knihovna vznikne pomocí linkeru poté, co se vytvoří všechny objektové soubory. Objem dat, zpracovávaný během překladu, může být velký, a proto by adresář, kde se nachází build area, měl být pro daný buildovací stroj lokální. Kdyby lokální nebyl (tzn. byl by vzdálený), docházelo by k přenosu značného množství dat, jak ukazuje obrázek 8, a to by mělo značný vliv na zatížení sítě a především na výkon build systému. Toto je explicitně zmíněno, protože strom zdrojových kódů je k buildovacímu stroji připojen vzdáleně ze source serveru.

Vzhledem k tomu, že výsledné artefakty vytvořené na jedné platformě můžou figurovat jako vstupní artefakty pro build proces nebo část build procesu na jiné platformě (např. packaging), je nutné, aby build area jednotlivých buildovacích strojů byla přístupná pro ostatní buildovací stroje (případně pro source server). Proto jsou build area na všech buildovacích strojích exportovány a poté připojeny na ostatní stroje.

K systému, na kterém probíhá překlad, jsou zdrojové kódy připojeny vzdáleně. Toto a výše zmíněná podmínka vyžadující, aby výstupní soubory build procesu byly zapisovány do lokálního build area implikují potřebu mít mechanismus, který by umožnil oddělit strom zdrojových kódů od stromu přeložených artefaktů. Různé strategie, jak toho dosáhnout, budou popsány v kapitole 4.7.



Obrázek 8 - Vzdálený build area

4.2.1.3 Testovací stroje

Testovací stroje nefigurují v samotném procesu překladu, ale jsou komponentou, která se promítne do výsledné podoby build systému a kterou musí brát build systém v potaz. Je nezbytné, aby vývojáři měli možnost co nejjednodušeji pracovat s vývojovými verzemi jednotlivých komponent softwaru. V praxi to znamená, že potřebují systém respektive prostředí, kde můžou software otestovat, podívat se jestli změny, které provedli, dělají to, co mají atd. Protože jednotlivé buildovací stroje jsou dedikovány čistě pro potřeby překladu software, je potřeba definovat třídu testovacích strojů.

Tyto testovací stroje slouží jako "hřiště", kde si vývojáři mohou zkoušet libovolné úkony spjaté s vyvíjeným software. Jako takové je potřeba počítat s tím, že systém se může nacházet v nestabilním stavu po zásahu některého z vývojářů. Je proto dobré, pokud je definován mechanismus, který umožní po otestování provedených změn uvést systém zpět do definovaného stavu, ve kterém se nacházel předtím, než vývojář začal s testováním.

Tento problém řeší použití virtualizačního software. V dnešní době je k dostání široká nabídka virtualizačního software (VMware, Solaris Zones, LDOMs atd.), která umožňuje nainstalovat a nakonfigurovat testovací systém, před jakýmkoliv dalším zásahem pořídit snapshot "čistého stavu" a kdykoliv se testovací systém dostane do nedefinovaného

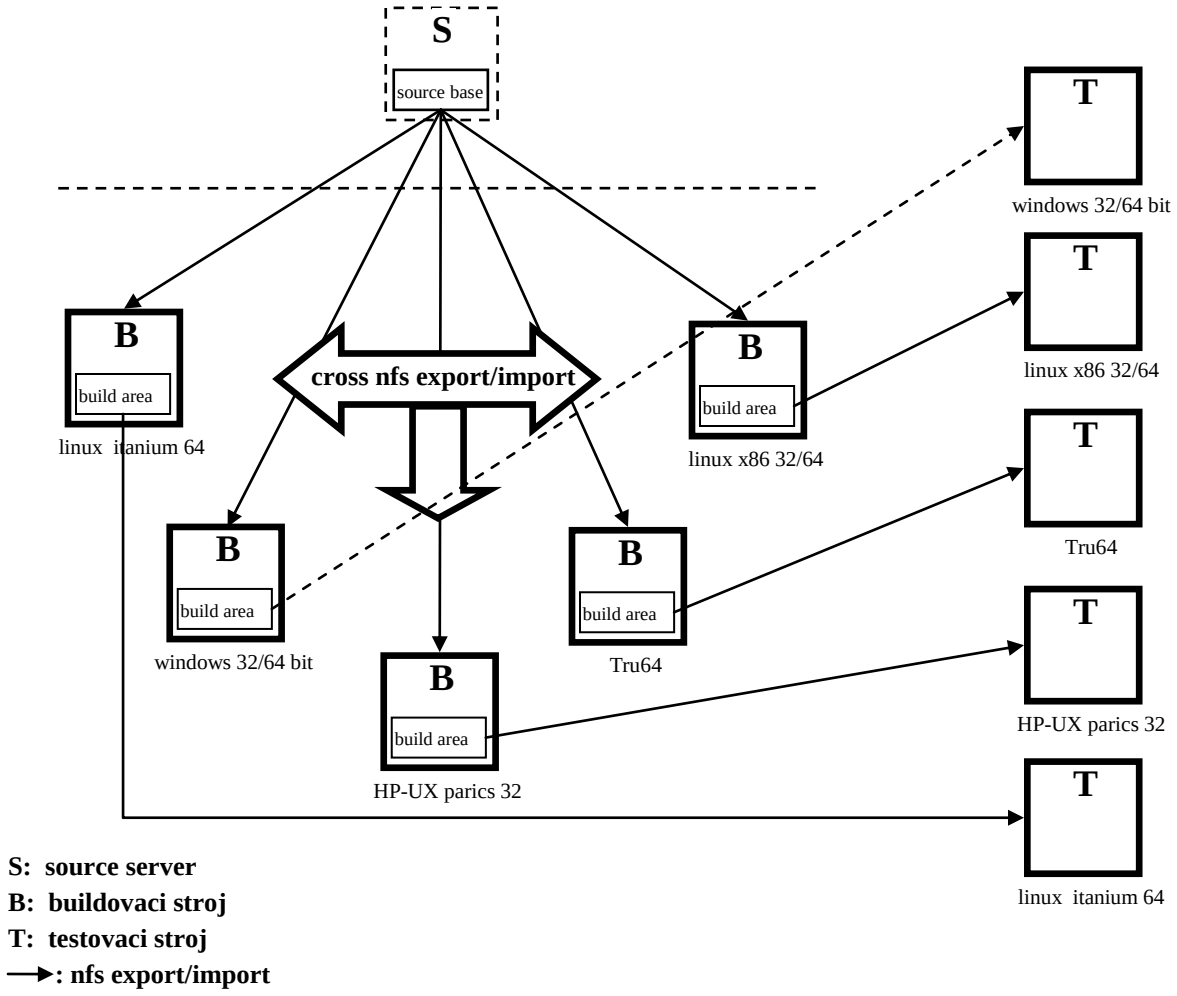
stavu na základě zásahů provedených vývojáři, použít tento snapshot pro znovuuvedení testovacího systému do původního stavu.

Mezi build systémem a testovacími stroji je definovaná úzká vazba. Primárním cílem integrace testovacích strojů a build systému je co nejrychleji a nejjednodušeji přenést změnu provedenou ve zdrojovém kódu při inkrementálním buildu do run-time prostředí, kde je možné ověřit funkčnost této změny. To umožní rapidní inkrementální vývoj softwaru, a ačkoliv by se mohlo zdát, že jde o okrajovou záležitost, je právě možnost rychle ověřit funkčnost změny klíčovým elementem, určujícím efektivnost vývoje software.

Implementace integrace navrhovaného build systému a testovacích strojů využije skutečnosti, že testovací stroje jsou definovány v rámci vývojového prostředí a správce build systému má nad nimi plnou kontrolu, a dále skutečnosti, že buildovací stroje exportují své build area.

Samotná integrace je díky těmto skutečnostem vyřešena velmi jednoduše nepřímým přístupem do build area z testovacích strojů. V praxi to znamená, že na testovací stroj je importován build area z buildovacího stroje, který platformově odpovídá testovacímu stroji a pro soubory, které chce vývojář propagovat na testovací stroj se v run-time lokaci jednotlivých souboru na testovacím stroji vytvoří místo originálních souboru symbolické linky na odpovídající soubory v build area přimountovaném z buildovacího stroje (originální soubory v originálních lokacích se můžou zazálohovat). Tímto nepřímým přístupem se změny při jednotlivých překladech automaticky propagují přímo na testovací stroj.

Celkový pohled na výslednou architekturu poskytuje obrázek 9. Srdcem struktury je tedy source server, který hostuje zdrojové kódy všech vývojářů, a tyto zdrojové kódy exportuje pomoci síťového filesystému (NFS, Samba). Jednotlivé buildovací stroje, na kterých se spouští samotný proces překladu, si importují adresář se zdrojovými kódy ze source serveru, a naopak exportují svůj lokální adresář deklarovaný jako build area.



Obrázek 9 - Vývojové prostředí

4.2.2 Formalizace vývojového prostředí

Vzhledem ke zvolenému modelu správy konfigurací, kdy místo implementace složité konfigurační vrstvy jsou buildovací stroje určeny staticky a tudíž korektnost build procesu závisí čistě na konfiguraci těchto strojů, deklarovaných jako oficiální buildovací stroje, je nutné zajistit, že konfigurace těchto strojů je reprodukovatelná a úplná.

Reprodukovatelná znamená, že v případě problémů (havárie, poškození disku…) je možno vytvořit buildovací stroj znovu a buildovací proces spuštěný na tomto novém stroji bude ekvivalentní s původním. To znamená, že je potřeba pojmenovat a formalizovat všechny komponenty a jejich verze, mající vliv na build systém. To může zahrnovat například následující komponenty: nástroj pro řízení překladu a jeho verze, knihovny a jejich verze, operační systém a jeho verze, kompilátor a jeho verze apod.

Úplná znamená, že software sestavený v rámci této konfigurace pokrývá všechny požadované cílové konfigurace (platformy, verze aj.). Zde hraje speciální roli operační systém a jeho verze, protože verze operačního systému je většinou ta granularita, se kterou komerční software pracuje. Nejjednodušším řešením je sestavit software na nejnižší podporované verzi dané platformy a poté se spolehnout na zpětnou komptabilitu, která je většinou dodavatelem dané platformy garantována. Jedna binární podoba software je pak použitelná na všech podporovaných verzích dané platformy.

4.3 Princip vzdáleného překladu

Princip vzdáleného překladu umožnuje vytvořit skutečně kompletní strom závislostí. Source server spouští závislostní engine, který z uživatelských definic (skriptů) sestaví kompletní strom závislostí, obsahující závislosti ze všech platforem. Při samotné fázi překladu spouští příkazy pro překlad, které jsou vykonávány vzdáleně na build strojích. Pro samotný source server je vytvořen dojem, že on sám je schopen lokálně přeložit libovolný cíl, libovolné platformy. Takový dojem (abstrakce) je nutný k tomu, aby bylo možné definovat kompletní, korektní strom závislostí, který zahrnuje závislosti všech možných platforem. Této abstrakce je dosaženo následujícími třemi mechanismy.

Build systém generuje sadu proxy nástrojů pro překlad. Tyto proxy nástroje se tváří jako standardní nástroj dané platformy (například překladač jazyka C pro plaformu Sun Solaris – suncc). Při překladu objektového souboru pro platformu Solaris pak build systém zavolá tento proxy příkaz, předá mu stejné parametry jako skutečnému suncc (build engine o proxy mechanismu nic neví) a proxy příkaz se pak postará o jeho skutečné zavolání na build stroji dané platformy.

Aby takovýto mechanismus fungoval, musí být zajištěno, že adresářové cesty vstupních i výstupních souborů pro proxy nástroje zůstanou konzistentní. To je dosaženo automaticky tím, že source server i build stroje sdílí stejný pohled na adresářovou strukturu díky použití síťového filesystému (např. NFS).

Poslední potřebný mechanismus je třída RemoteEvironment. To je rozšíření Scons třídy Environment o možnost specifikovat platformu a vzdálený Build stroj tak, aby při zavolání Builderu instance této třídy, Builder použil vhodný proxy nástroj. RemoveEnvironment přijímá kromě standardních parametrů třídy Environment parametr remote_cfg. Tento parametr určuje, pro jakou konfigurační platformu se bude používat tato instance třídy RemoteEnvironment. Zde je ukázka parametru remote_cfg, který specifikuje, že daná instance třídy RemoteEnvironment bude vytvářet cíle pro 64-bit verzi systému Solaris 10 na archtektuře x86.

```
remote_cfg:{'OSNAME':'solaris',
            'BITARCH':'64',
            'OSVER':'10',
```

'MACHINEARCH':'x86'}

Vhodný build stroj je tomuto Environmentu přidělen automaticky na základě této konfigurace. Build systém si při startu načte informace o existujících build strojích z konfiguračního souboru a při vytváření instance třídy RemoteEnvironment přidělí této instanci vhodný build stroj.

4.4 Paralelizace překladu

Jednotný strom závislostí přes všechny platformy s sebou přináší problém, jak efektivně využít build stroje a maximálně urychlit proces překladu. Pokud build engine začne procházet strom závislostí a překládat software, tak, jako by se jednalo o čistě lokální překlad, zjistíme, že většinu času build stroje nic nedělají a čekají, až přijde v překladu řada na nějaký cíl, patřící jejich platformě. Pokud například existují tři varianty knihovny libfoo – Linux, Solaris a AIX, zjistíme, že build engine první začne spracovávat všechny objekty linuxové verze, poté všechny objekty verze pro Solaris a až poté všechny objekty verze pro AIX. Je to dáno tím, že Scons třída, starající se o průchod stromu závislostí – Taskmaster, dělá průchod stromu do hloubky, zatímco varianty knihovny libfoo se ve stromu nachazí "vedle sebe". SCons engine proto musí být upraven tak, aby při průchodu stromem bral v úvahu existenci build strojů a vhodně průchod stromem závislostí paralelizoval. Tato paralelizace je řešena třídou Parallelizer. Ta si udržuje informace o tom, kolik úloh daný build stroj zpracovává a spolupracuje s třídou Taskmaster na distribuci úloh na build stroje. Tento problém je netriviální. Pokud máme build stroj, který v danou chvíli nezpracovává žádnou úlohu, ale někde ve stromu závislostí se nachází uzly, které mohou být zpracovány tímto uzlem, nejde z pohledu build enginu říct, kde a za jak dlouho na ně třída Taskmaster narazí. To způsobuje veliké prodlevy, celkovou neefektivu a velmi nízkou paralelizaci celého překladu. Tento problém je vyřešen vytvořením seznamu tzv. lokálních kořenů (local_root). Lokální kořen ve stromu závislotí je definován následovně:

$$Local\_roots = \{node \mid \forall\ parent\ of\ node;\ parent.buildAgent\ !=\ node.buildAgent\}$$

Seznam lokálních kořenů se vygeneruje při sestavování stromu závislostí (tím se zamezí nákladnému procházení stromu znovu a znovu, kdykoliv je takový kořen potřeba). Při překladu pak třída Parallelizer hlídá zatížení jednotlivých build strojů a ve chvíli kdy zjistí, že nejaký stroj nezpracovává žádnou úlohu, injektuje do seznamu kandidátů na zpracování nějaký lokální kořen patřící danému build stroji. Tím dostane nezaneprázdněný build stroj úlohu na zpracování, a navíc je velká pravděpodobnost, že tento lokalní kořen je kořenem hlubšího stromu závislostí patřících tomuto build stroji, takže takové injektování udrží tento stoj na delší dobu zaneprázdněný. Jako příklad, s knihovnou libfoo, by knihovna libfoo figurovala jako lokalní kořen, ale proto, že tato knihovna je závislá na objektových souborech, tyto objektové soubory se okamžitě dostanou do seznamu kandidátů na

zpracování společně s tímto lokálním kořenem. To na nějakou dobu zaneprázdní daný build stroj, než bude znova potřeba injektovat další lokalní kořen.

4.5 Struktura build systému a build skriptů

Tato kapitola se snaží v kontextu kapitoly 3.2 definovat obecná pravidla pro to, jak aplikovat princip divide et impera a tedy jak modularizovat BS a jak rozdělit definici AOG do více build skriptů. Tato pravidla jsou pak využívána při konstrukci reálného BS.

4.5.1 Zdrojové uzly – hlavní ukazatel struktury rozložení build skriptů

Struktura BS se nutně musí odvíjet od struktury SP a to především od struktury zdrojových uzlů. Důvodem je fakt, že před samotným překladem jsou zdrojové uzly jedinými existujícími entitami z AOG. Zbylé entity (tzn. vnitřní uzly) jsou jaksi virtuální, existující pouze ve smyslu definic v build skriptech. To znamená, že při psaní build skriptů pracujeme v podstatě pouze s těmito zdrojovými uzly a tudíž to, jak rozdělit definici AOG do jednotlivých build skriptů závisí především na struktuře rozložení zdrojových uzlů.

Zdrojové uzly jsou v konečném důsledku soubory ve filesystému a jako takové tedy tvoří adresářovou stromovou strukturu. Struktura zdrojových uzlů ve file systému reflektuje strukturu SP. Související entity se nachází ve stejných adresářích, které se dále sdružují do podadresářů atd.

Typickým příkladem je knihovna: zdrojové kódy knihovny A sídlí v jednom adresáři. Skupina knihoven zajišťujících příbuznou funkcionalitu (např. knihovny starající se o GUI aplikace) pak například může sídlit v jednom společném podadresáři ve kterém má každá z těchto knihoven vlastní adresář se svými zdrojovými texty. Struktura build skriptů by pak tedy měla reflektovat tuto strukturu zdrojových uzlů. V příkladu s knihovnou bude do adresáře knihovny umístěn build skript definující překlad této knihovny ze zdrojových kódů.

Obecné pravidlo pak zní, že jednotlivé build skripty by měly být vytvářeny a umístěny tak, aby všechny zdrojové uzly z AOG a všechny definice vnitřních uzlu z AOG uvedené ve skriptu, byly kladné. Kladným se rozumí uzel, který je v adresářové struktuře na stejné nebo nižší úrovni než build skript (ukázka záporné reference: "*../../main.c*"). To značně zvyšuje přehlednost a udržovatelnost build systému, protože jednotlivé adresáře pak tvoří určitý druh jmenného, respektive pracovního prostoru, v rámci kterého jsou všechny entity definovány, a v rámci kterého se implementuje daný build skript.

Dalším pravidlem je, že definice vnitřních uzlů AOG by měly ležet v takovém build skriptu, který je v adresářové struktuře zdrojových uzlů umístěn co nejblíže podadresáři, který obsahuje všechny zdrojové uzly, na kterých je daný vnitřní uzel závislý (tzn. mezi

tímto podadresářem a adresářem obsahujícím build skript je co nejméně úrovní, v nejlepším případe se tyto adresáře rovnají).

Dáme li tyto dvě pravidla dohromady, můžeme definovat pomocný algoritmus, jehož výsledkem je struktura rozložení build skriptů v BS.

4.5.1.1 Algoritmus RZU

Algoritmus RZU (Rozdělení podle Zdrojových Uzlů) tedy dostane jako vstupní parametr strom AOG a jeho výsledkem je rozvržení definic vnitřních uzlů do jednotlivých build skriptů. Algoritmus se dá popsat pomocí následujících kroků:

1. Dokud existuje neoznačený uzel v AOG.
2. Vezmi neoznačený vnitřní uzel U z AOG a označ ho.
3. Vezmi podstrom tohoto uzlu a získej množinu Z všech zdrojových uzlů.
4. Najdi nejhlubší podadresář P takový, že obsahuje všechny prvky množiny Z.
5. Jestli v adresáři P existuje build skript, přidej definici U do tohoto build skriptu, přičemž definiční cesta k U je kladná. Jestli build skript v adresáři P neexistuje, tak ho vytvoř.

Algoritmus funguje dobře, pokud je SP, respektive struktura zdrojových uzlů dobře a čistě navržena. V příkladu s knihovnou A leží všechny zdrojové kódy v jednom adresáři, takže algoritmus skutečně vygeneruje build skript, který leží v tom samém adresáři jako zdrojové kódy a definuje výstupní knihovnu také v tomto adresáři, takže po překladu bude také zde, což je očekávaný výsledek. V praxi ale dochází k nejrůznějším situacím a navíc ne vždy je struktura SP dobře navržena.

Pro ilustraci řekneme, že adresář knihovny K z předchozího příkladu se nachází někde hluboko ve filesystému, zanořený v několika adresářích zastřešujících například nějakou komponentu složitého programu. Řekněme, že existuje jiná knihovna B, respektive adresář obsahující její zdrojové kódy, ve zcela jiném podstromu jiné komponenty. Knihovna A však potřebuje zdrojový soubor foo.c knihovny B obsahující například nějaké obecné makro či funkci které by vývojář knihovny rád použil v A. Nejrychlejší způsob je, aby přidal foo.c do překladu A čímž se A stane závislá na zdrojovém kódu foo.c, nacházejícím se v úplně jiném podstromu. V takovém SP by pak výsledkem algoritmu bylo to, že by kvůli jedinému zdrojovému souboru vložil algoritmus definici knihovny A někam velice vysoko ve stromu zdrojových uzlů, někam zcela mimo podstrom programové komponenty, do které A náleží.

Je tedy vidět, že Algoritmus nelze v praxi slepě následovat a je potřeba dělat výjimky. Algoritmus slouží spíše jako orientační bod, který říká, kdy postupujeme správně, a kdy se odchylujeme od teoreticky korektního modelu a děláme výjimky z obecného pravidla. Nejlepší z dostupných řešení je v případě knihovny A porušit pravidlo o nepoužívání záporných cest a jednoduše specifikovat cestu k foo.c jako "*../../../../componentB/bar/foo/foo.c*".

Důležitou připomínkou je to, že situace, která vznikla v případě knihovny A, vznikla na základě špatného inženýrského rozhodnutí, případně na základě chyby v designu kódu: pokud vývojář knihovny A potřeboval něco z foo.c tak to znamená, že foo.c obsahuje obecný, znovupoužitelný fragment kódu (je potřebný nejméně na dvou místech, v knihovně A v knihovně B) a jako takový by měl být poskytnut jako samostatná knihovna, archiv, případně hlavičkový soubor nebo by měl být umístěn do samostatného zdrojového kódu a umístěn na nějakou "obecnější úroveň" než je úroveň "knihovna B v komponentě B". Takové řešení by bylo nejkorektnějším řešením z pohledu designu a algoritmus RZU by se choval správně.

Z praxe se dá říci, že pokud algoritmus RZU generuje nesprávná nebo neschůdná rozdělení definic AOG do jednotlivých build skriptů, velice pravděpodobně to ukazuje na problém v návrhu či rozvržení SP či zdrojových kódů.

4.5.2 Vnitřní uzly

Důležitým pozorováním je, že algoritmus RZU při hledání správného build skriptu bere v úvahu pouze zdrojové uzly (množina Z je generována ze zdrojových uzlů). Důvod, proč RZU nebere v potaz vnitřní uzly je ten, že v rámci SP v počátečním nepřeloženém stavu tyto uzly neexistují, a neexistuje ani adresářová struktura, s těmito uzly spojená. To znamená, že neexistují ani adresáře, ve kterých by měl případný build skript ležet. Jak již bylo řečeno, vnitřní uzly jsou na počátku pouze virtuální a existují pouze jako definice v build skriptech. Z tohoto faktu vyplývá následující pozorování.

Specifikace závislostí na zdrojových uzlech je do značné míry dána staticky, jasně určena jménem zdrojového uzlu ve filesystému. Specifikace závislostí na vnitřních uzlech už takovou statickou povahu nemá. Důvodem je, že uzly při psaní build skriptu fakticky neexistují. Navíc, vnitřní uzly a závislosti jsou definovány dynamicky při zpracovávání build skriptu a jméno daného vnitřního uzlu ve filesystému se může podle různých okolností měnit.

Příkladem může být různá přípona souboru knihovny, kdy např. stejná knihovna bude mít na Linuxu příponu .so kdežto na systému HP-UX koncovku .sl (pokud se jedná o původní PARISCový systém, kde je jiný linkovací model než ELF). Je proto vhodné a mělo by být pravidlem, že ke všem vnitřním uzlům by se mělo přistupovat symbolicky a ne staticky přes jména ve filesystému. To odstíní potřebu znát přesně jednak přesnou cestu k uzlu a jednak přesné jméno, které vznikne jako výsledek zpracovávání build skriptu.

Symbolický přístup je obecně realizován, a to nejen v programovacích jazycích, konceptem proměnných - nějaké entitě přiradíme symbolické jméno a dále pracujeme s tímto jménem a ne se samotnou entitou.

4.5.2.1 Rozhraní přístupu k vnitřním uzlům

Z objektově orientovaného programování jsou známy principy zapouzdření, kdy autor třídy, přes dobře definované a veřejně známé rozhraní, zveřejní pro okolní svět všechny metody a datové objekty, které by měly být používány zvenčí a zbytek, vnitřní části třídy, skryje. To spřináší spousty výhod, protože autor třídy je ten, který ví, které části by se měly používat jinými třídami a které ne. Pro ty, které by měly být veřejné, definuje dostatečně jednoduché rozhraní, aby odstínil uživatele rozhraní od implementačních detailů. Stejný princip lze využít i u implementace BS.

Při implementaci konkrétního build skriptu správce tohoto skriptu definuje všechny vnitřní uzly, zdrojové uzly a závislosti, které spadají po správu tohoto skriptu (všechny jejich cesty jsou kladné). Poté správce zveřejní přes dobře definované rozhraní všechny uzly, které jsou určeny k použití mimo rámec lokálního build skriptu. Všichni správci ostatních build skriptů, kteří pak potřebují využít některý ze zveřejněných uzlů ve svém build skriptu pak použijí toto rozhraní při přístupu k tomuto uzlu. Tím se docílí toho, že build skript zůstane konzistentní, všechny cesty v něm zůstanou kladné a je jasně určeno, co jsou lokální zdroje patřící pod správu tohoto build skriptu a co jsou externí uzly patřící pod správu jiného build skriptu. Navíc se ke všem externím závislostem přistupuje jednotným způsobem, přes jednotné rozhraní a nedělají se rozdíly mezi tím, zda je dána závislost externí pro celý SP nebo zda je externí pouze pro daný build skript (tzn. je definovaná v jiném build skriptu, ale je součásti SP). To má obrovský význam pro modulárnost.

Toto rozhraní vypadá různě a závisí na typu objektu, který se přes rozhraní exportuje/importuje. Je tedy na vývojáři BS, aby definoval a implementoval jednotlivá rozhraní pro různé typy objektů. Uvedeme si příklad takovéhoto rozhraní v příkladu s knihovnou A.

Řekněme, že vývojář knihovny A se rozhodne využít služeb knihovny B a chce tedy slinkovat A s B. Pokud by neexistoval mechanismus rozhraní popsaný výše, musel by si zjistit, kam se knihovna B přeloží a odtud ji přilinkovat ke knihovně A. To by vypadalo (v případě C knihovny) během překladu takto:

gcc -o libA.so -L ../../../../ComponentB/bar/foo -l libB

Podívejme se na chvíli, jak to funguje v systému UNIX při linkování nějakého objektu se standardní knihovnou jako např. matematickou knihovnou libm. Zde se nestaráme o to, kde knihovna leží, kompilátoru pouze řekneme, že chceme použít knihovnu m tím, že mu předáme parametr -lm. Podíváme-li se na systém UNIX blíže, nalezneme zde pojem standardní cesty: v systému se nachází několik míst, které jsou deklarovány jako standardní pro hledání knihoven (např. /lib a /usr/lib) tudíž jediné, co musíme znát, pokud chceme použít nějakou standardní knihovnu (obecněji jakoukoliv instalovanou knihovnu, kde instalací se myslí instalace tak, jak je definována např. balíčkem libtool) je její jméno. To je chování, které by bylo dobré mít i na úrovni BS.Jediné, co by měl být správce build

skriptu nucen specifikovat, je jméno knihovny, kterou potřebuje - to kde leží je nezajímavé. Interface pro definování závislostí na knihovnách bude tedy v BS definován následovně.

BS definuje standardní cestu, řekněme "*<build_root>/lib*". Tato cesta se bude prohledávat vždy během algoritmu vyhledávání knihoven, který je prováděn v linkovací fázi při překladu. Správce build skriptu, který definuje knihovnu B tuto knihovnu zveřejní tím, že po překladu vytvoří symbolický link do *<build_root>/lib* případně ji tam zkopíruje. Touto akcí dává najevo, že je to objekt určený pro veřejné použití v rámci BS. Správce knihovny A pak jednoduše deklaruje závislost na knihovně B a tuto deklaraci realizuje pouze tím, že poskytne linkeru jméno této knihovny. Při vykonávání to bude vypadat takto:

"*gcc -o libA.so -l libB*".

Tady přestal být rozdíl mezi tím, jakým způsobem linkujeme externí systémovou knihovnu, která je externí závislostí z pohledu celého SP (libm) a tím, jakým způsobem linkujeme externí závislosti, které jsou externí z pohledu konkrétního build skriptu (libB). Tím se také odstraní potřeba specifikovat zápornou cestu, která určovala umístění knihovny B. Způsob implementace těchto rozhraní se bude lišit v závislosti na typu objektu. Např. rozhraní pro vytváření závislostí na hlavičkových souborech bude podobné výše popsanému rozhraní pro knihovny hlavně z toho důvodu, že v jistém smyslu kompilátor zachází s hlavičkovými soubory velice podobně jako s knihovnami. Pro jiné typy objektů však takovéto rozhraní může vypadat značně odlišně a je na vývojáři BS, aby jej správně definoval.

4.6 Varianty

Kapitola 3.3 nastínila problematiku variant a následující kapitola se snaží tuto problematiku řešit. Jak bylo ukázáno v analýze, hlavní problém, který je nutné adresovat je nemožnost zachycení dvou variant pocházejících z dvou různých iterací procesu překladu v rámci jednoho AOG.

4.6.1 Přesun rozhraní

Problém je nutné vyřešit obecně na úrovni AOG. V příkladu ukázaném v kapitole 3.3 potřebujeme AOG, který obsahuje kompletní definici všech uzlů a závislostí, a to od množiny počátečních zdrojových uzlů až k výslednému DVD obrazu. Předpokládejme nyní, že máme takový AOG a podívejme se, co se stane, pokud na tento strom použijeme algoritmus RZU. Výsledkem je, že algoritmus rozdělí definice vnitřních uzlů do jednotlivých build skriptů. Co se ovšem stane je, že pro některé entity, které jsou na sémantické úrovni shodné, zde budou dvě definice. Tím, že jsou na stejné sémantické

úrovni, je myšleno to, že ačkoliv v grafu AOG jsou to dvě různé entity, ze sémantického pohledu, tedy z pohledu toho jaké využití tyto entity mají, se jedná o jednu entitu. Jako ukázku takových entit můžeme předpokládat, že ukázkový projekt obsahuje nativní knihovnu funkcí k.so. Tato knihovna poskytuje jak na Linuxu, tak na Solarisu stejné API, linkuje se se stejnými závislostmi a překládá se se stejnými parametry. Jediný rozdíl je, že výsledné ABI (application binary interface) knihovny je různé podle toho, na jakém systému byla knihovna přeložena.

Je nepraktické, aby pro každou takovou variantu jednoho sémantického objektu byla v build skriptu speciální definice - software kromě Linuxu a Solarisu podporuje několik dalších platforem, procesorových architektur s různou bitovostí apod. Proto je žádoucí, aby pro jeden sémantický objekt byla v build skriptu pouze jedna definice. Současně ale v zájmu obecnosti implementovaného BS a v zájmu konzistentnosti a úplnosti grafu AOG chceme, aby tyto různé varianty byly součástí jednoho grafu AOG, pokud je to potřeba (a v tomto příkladě to potřeba je, protože potřebujeme dvě různé varianty současně, abychom mohli vytvořit finální DVD obraz).

Tyto dva požadavky jsou ale v přímém rozporu: pokud je v build skriptu pouze jedna definice, můžeme získat pouze jednu variantu. Pro získání druhé varianty musíme spustit build proces znovu, čímž ale vytváříme dva oddělené grafy AOG. Naopak, pokud máme jednotný AOG, který obsahuje všechny potřebné varianty daného sémantického objektu, pak po aplikaci algoritmu RZU máme ve výsledných build skriptech několik definic pro jeden sémanticky objekt.

Počet variant daného sémantického objektu je obecně neomezený. Pro ilustraci lze uvažovat, že knihovna je zkompilovaná s parametrem *-DPRODUCT_NAME="productX"*, kde *PRODUCT_NAME* je specifikován jako vstupní parametr pro BS a může nabývat libovolnou řetězcovou hodnotu, To znamená, že počet různých variant daného sémantického objektu nelze shora omezit. Co však omezit určitě jde, je počet variant daného sémantického objektu, které je potřeba mít současně zahrnuté v jednom grafu AOG. Ať už je požadavek respektive potřeba uživatele jakkoliv náročná, lze zaručit a dokázat, že počet variant libovolného sémantického objektu je konečný (AOG má konečný počet uzlů).

Pokusíme se tedy dát tyto poznatky dohromady a najít způsob, jak by bylo možné, aby AOG byl úplný a současně, aby pro různé varianty daného sémantického objektu byla v build skriptech specifikována pouze jedna definice.

Existence variant je způsobena tím, že BS je ovlivněn faktory, jako jsou vstupní uživatelské parametry, prostředí apod. Proto pro každý uzel v AOG definujeme množinu M vstupních faktorů, které mají vliv na tento uzel. Vezmeme-li úplný graf AOG obsahující všechny potřebné závislosti a všechny varianty (v našem příkladu variantu pro Linux a pro Solaris) a aplikujeme-li na tento graf algoritmus RZU, získáme tím build skripty obsahující definice všech vnitřních uzlů jako variant pro daný sémantický objekt.

Co je však nyní možné provést je, že pro nějaký sémantický objekt (např. knihovnu k.so) vezmeme definice všech jeho variant a nahradíme ji jednou definicí parametrizovanou

pomocí množiny M a tuto definici použijeme opakovaně pro každou variantu existující v AOG pro daný sémantický objekt. Lze si to představit tak, že provedení definice vnitřního uzlu AOG je vložena do funkce, která má jako vstupní parametr množinu M a v závislosti na tomto vstupu provede po zavolání definici daného vnitřního uzlu AOG respektive jeho varianty. Každé zavolání takovéto funkce s patřičnými parametry vytvoří novou variantu v AOG. Toto má jeden netriviální důsledek, který však umožňuje vyřešit problematiku variant na obecné úrovni.

Tím, že se napříč všemi build skripty vytvořily znovupoužitelné procedury, vytvářející definice vnitřních uzlů, se přeneslo rozhraní, přes které se parametrizuje build systém z vnějšku BS dovnitř BS, na jednotlivé uzly AOG. Když bylo toto rozhraní vně BS, řekněme mezi uživatelem a BS, (uživatel byl ten, kdo dodá vstupní parametry BS a "spustí" zpracovávání build skriptu) dalo se k tomuto rozhraní přistupovat pouze z vnějšku (**uživatelem iniciovaná a parametrizovaná exekuce**). Nexistovala žádná možnost, jak by BS samotný k tomuto rozhraní přistoupil. To implikuje fakt, že bylo možné získat pro daný sémantický objekt (k.so) pouze jednu variantu. Když je ale nyní toto rozhraní uvnitř BS znamená to, že BS k němu může libovolně přistupovat a vytvářet libovolné množství variant daného sémantického objektu. Tím je BS schopen vytvořit úplný a korektní AOG obsahující všechny potřebné vnitřní uzly tak, aby bylo možné definovat kompletní závislosti a to od množiny počátečních zdrojových uzlů, až ke skutečnému finálnímu výsledku, který požadujeme. V příkladu z kapitoly 3.3 to znamená, že BS je takto schopen definovat variantu pro Linux, variantu pro Solaris a to jakým způsobem se z nich vytvoří finální DVD v jednom uceleném AOG. To umožní získat finální produkt v jednom spuštění BS bez jakýchkoliv manuálních mezikroků.

Zajímavým pozorováním je, že BS je nyní sám schopen řídit vytváření variant daného sémantického objektu a je technicky možné definovat závislosti **mezi** jednotlivými variantami jednoho sémantického objektu. Z pohledu grafu AOG žádné varianty neexistují, všechno jsou „jen“ uzly v grafu a není jediný důvod, proč by mezi nimi nemohl existovat vztah závislosti. Otázkou je, do jaké míry dává vytváření takovýchto závislostí smysl. Syntakticky zde žádný problém není, ale sémanticky takové závislosti většinou nedávají dobrý smysl. Je proto na uvážení vývojáře zda by měl takové závislosti vůbec definovat. Definováním takové závislosti říkáme, že jedna varianta sémantického objektu nemůže existovat bez druhé (linuxová verze k.so nemůže existovat bez solarisové verze k.so). Tím v podstatě znemožníme samostatnou existenci závisle varianty. To může mít značný negativní vliv na modulárnost a flexibilitu celého BS.

4.6.2 Implementace v doméně nástroje SCons

4.6.2.1 Vytváření variant

K implementaci variant se použije třída Environment. Této třídě se poté předávají parametry jako záznamy do jejího implicitního slovníku (dictionary).

Samotná parametrizace se pak bude provádět pomocí volání specifických Builderů. Builder jako takový je metodou objektu třídy Environment.

```
myenv.SharedLibrary('m','m.c')
```

Výše uvedené volaní vytvoří v kontextu prostředí myenv entitu v AOG reprezentující knihovnu m. Vytváření různých variant stejného sémantického objektu pak lze provést voláním daného builderu v kontextu různých prostředí, jak ukazuje následující kód.

```
for env in [myenv,myenv1]:
        env.SharedLibrary('m','m.c')
```

Příklad, uvedený výše, je triviální. V praxi ale může být kontext, ve kterém se vytváří daný objekt složitější. Tento kontext je z povahy build skriptu psaných v jazyce Python imperativní mělo by se k němu takto přistupovat i při tvorbě variant. To ale znepřehledňuje a přehlednost je u build skriptů jednou z hlavních priorit.

Následující build skript definuje ukázkovou knihovnu k. Build skript neuvažuje existence souběžných variant a výsledný kód je díky tomu čistý a přehledný. Přehledností se blíží build skriptu psaném v deklarativním jazyce.

```
SRCS1 = ['m.c','k.c']
SRCS2 = ['m.c','k.c']

objs = env.SharedObject(SRCS1)
Env.SharedLibrary('m1',objs)

Env.MyBuilder('some_object_architecture_independent','source_file')

objs = env.SharedObject(SRCS2)
Env.SharedLibrary('m2',objs)
```

Jakmile se začnou uvažovat i varianty, build skript znatelně ztratí na přehlednosti:

```
SRCS1 = ['m.c','k.c']
SRCS2 = ['m.c','k.c']
for env in [myenv,myenv1]:
```

```
        objs = env.SharedObject(SRCS1)
        Env.SharedLibrary('m1',objs)

    Env.MyBuilder('some_object_we_need_only_single_variant,'source_file')

    for env in [myenv,myenv1]:
        objs = env.SharedObject(SRCS2)
        Env.SharedLibrary('m2',objs)
```

Problém by se dal řešit vytvořením kontejneru pro jednotlivá prostředí a voláním Builderu jako metody tohoto kontejneru s tím, že by kontejnerový objekt zajistil, že Builder bude ve finále zavolán nad všemi environmenty, které kontejner obsahuje. Imperativní charakter kódu však znemožňuje obecnou použitelnost takovéhoto kontejneru. Při imperativních operacích nad kontejnerem se agregací výstupů Builderů z jednotlivých Environmentů v kontejneru ztratí vazba mezi těmito výstupy a aplikací těchto výstupů v následných voláních jiných builderů, kde je potřeba udržovat vazby mezi jednotlivými entitami vytvořenými v rámci daného Environmentu.

Pro lepší pochopení ukazují následující dva příklady build skript, který nevyužívá imperativní vlastnosti jazyka a kde by bylo možné použít takovýto kontejner a build skript, který využívá imperativní vlastnosti a kde by použití takového kontejneru nemělo smysl.

```
SRCS1 = ['m1.c','k1.c']
SRCS2 = ['m2.c','k2.c']

envs = EnvironmentContainer()
envs.AddEnvironment([myenv1,myenv2])

envs.SharedObject(SRCS1)
envs.SharedLibrary('m1', ['m1.o','k1.o'])

myenv1.MyBuilder('some_object_architecture_independent','source_file')

 envs.SharedObject(SRCS2)
envs.SharedLibrary('m2',['m2.o','k2.o'])
--------------------------------------------------------------------------------

SRCS1 = ['m1.c','k1.c']
SRCS2 = ['m2.c','k2.c']

envs = EnvironmentContainer()
```

envs.AddEnvironment([myenv1,myenv2])

***objs** = envs.SharedObject(SRCS1)*
*envs.SharedLibrary('m1', **objs)***

myenv1.MyBuilder('some_object_architecture_independent','source_file')

***objs** = envs.SharedObject(SRCS2)*
*envs.SharedLibrary('m2',**objs)***

Protože neexistuje řešení, jak toto dostatečně efektivně vyřešit, není možné takovýto kontejner použít a je lepší soustředit se na příčinu zvýšení složitosti build skriptu. Jádrem problému je ve skutečnosti to, že jak bylo popsáno v kapitole 4.6.1, rozhraní přes které se parametrizuje build systém se přeneslo z vnějšku BS dovnitř BS až na jednotlivé uzly AOG. Taková granularita je sice nejobecnější možná, ale jak bylo ukázáno výše, rapidně snižuje přehlednost build skriptu, protože uživatel se musí na úrovni jednotlivých entit starat o to, aby byly vytvořeny všechny potřebné varianty dané sémantické entity – tzn. musí se starat, aby byly všechny Buildery korektně zavolány v rámci patřičných Environmentů a aby byly dodrženy korektní závislostní vztahy mezi entitami.

Vhodným kompromisem mezi obecností build skriptů a jejich jednoduchostí je přesunout rozhraní na úroveň jednotlivých build skriptů místo jejich přesunutí na úroveň jednotlivých entit. Tím se ztrácí možnost vytvářet pro jednotlivé sémantické objekty, které leží v jednom build skriptu navzájem od sebe různé množiny variant (rozhraní přes které se předává množina M – neboli Environment je na úrovni celého souboru) Praxe ovšem ukazuje, že většinou je množina požadovaných variant v rámci jednoho build skriptu stejná pro všechny sémantické objekty, obsažené v tomto build skriptu. Protože build skripty jsou python moduly, je scope build skriptu stejný jako lokální scope pythonovského modulu, čímž se na obecné úrovni řeší imperativní problémy zmíněné výše.

V případě nutnosti, není problém pro izolované případy granularitu zjemnit a pro nějaký build skript přesunout rozhraní zpátky na jednotlivé sémantické objekty. Když je tedy rozhraní přesunuto na úroveň build skriptu, je najednou možné dosáhnout toho, že build skripty samotné jsou přehledné, a současně umožňuji generování variant, jak ukazuje následující příklad.

File: ./m/Sconscript:

SRCS1 = ['m.c','k.c']
SRCS2 = ['m.c','k.c']

objs = env.SharedObject(SRCS1)
Env.SharedLibrary('m1',objs)

Myenv1.MyBuilder('some_object_architecture_independent','source_file')

objs = env.SharedObject(SRCS2)
Env.SharedLibrary('m2',objs)

File: ./Sconscript:
For env in [myenv1,myenv2]:
Env.Sconscript('m/SConscript',exports ={'env':env})

4.6.2.2 Prefixování variant

Pokud by byl kód z předchozího příkladu spuštěn tak jak je, proces překladu by nejspíše skončil následující chybou:

"scons: warning: Two different environments were specified for target"

Důvod vychází z toho, že pojem varianty jako takové v samotném AOG neexistuje. Z pohledu AOG existují pouze entity, které reprezentují různé artefakty. Dalo by se říct, že pojem varianty existuje pouze při definování AOG a slouží k usnadnění tohoto definování. AOG tedy nezná pojem varianty a SCons identifikuje výsledné artefakty pomocí cesty v adresářové struktuře. Proto dochází na úrovni filesystému ke kolizi variant - ačkoliv jde v rámci AOG o dvě různé entity, tak na úrovni build skriptu jsou varianty identifikovány stejným jménem, které se pak použije k unikátní identifikaci entity v rámci filesystému.

Problém se vyřeší prefixováním variant, kdy každé variantě nějakého sémantického objektu je vložen před její jméno prefix, který identifikuje tuto variantu.

Environment je množinou M, a jednoznačně určuje variantu daného sémantického objektu (sémanticky objekt je definován daným Builderem). Má proto smysl, aby byl prefix definován pravě v objektu třídy Environment. Tím se zaručí konzistence v tom smyslu, že kdykoliv se bude vyvářet varianta nějakého sémantického objektu, která je určena jednoznačně Environmentem, ve kterém byl zavolán Builder pro tento objekt, bude tato varianta prefixována prefixem získaným z tohoto Environmentu. Ve skriptech to může být realizováno takto.

for env in [myenv,myenv1]:
env.SharedLibrary('%s/m' % (env['VARIANT_PREFIX']),'m.c')

V případě rozhraní na úrovni build skriptu je možno využít argument metody Sconscript "variant_dir" který spustí zpracování build skriptu jako by se nacházel v cestě určené tímto argumentem (pokud cesta neexistuje, je automaticky vytvořena). Tím je

automaticky přidán prefix určený argumentem variant_dir do všech cest všech entit resp. variant vytvořených v rámci zpracování tohoto build skriptu.

```
for env in [myenv1,myenv2]:
        Env.Sconscript('m/SConscript',
                       exports ={'env':env},
                       variant_direnv['VARIANT_PREFIX'])
```

4.7 Separování cílů od zdrojů

Důvodů, proč je dobré mít možnost oddělit výsledné entity od zdrojových je několik. Oddělení umožní:

- Nastavit strom zdrojových kódů jako read only takže nikdo nemůže modifikovat zdrojové kódy a současně kdokoliv z nich může sestavit výsledný software.
- V případě, že je strom zdrojových souborů připojen vzdáleně, umožní sestavit software lokálně za použití vzdáleného stromu zdrojových kódů, což má velký dopad na zatížení sítě a na rychlost procesu překladu.
- V případě multi-platformního projektu využívajícího více buildovacích strojů umožní spustit build procesy na všech strojích paralelně, aniž by docházelo ke kolizím, kdy stejné objekty sestavené na různých platformách by ve filesystému kolidovaly.
- Usnadňuje packaging či vytváření SDK balíčků, protože výstupní soubory dané platformy (varianty) jsou koncentrovány na jednom místě ve stromu, který má jeden kořen.

Existují prakticky dva hlavní způsoby jak takovéto oddělení realizovat. Jedním způsobem je prefixování cílů a druhým tzv. VPATH technika.

4.7.1 Separování pomocí VPATH

Tato technika je pojmenována podle proměnné VPATH z nástroje GNU make, pomocí které je tato technika realizována. Techniku používá pro oddělení výsledných souboru např. GNU build systém.

Pro pochopení této techniky je dobré si uvést definici proměnné VPATH a definici algoritmu prohledávaní adresářů popsanou v manuálu nástroje GNU make.

„The value of the make variable VPATH ***specifies a list of directories that make should search.*** *The algorithm make uses to decide whether to keep or abandon a path found via directory search is as follows:*

1. *If a target file does not exist at the path specified in the makefile, directory search is performed.*
2. *If the directory search is successful, that path is kept and this file is tentatively stored as the target.*
3. *All prerequisites of this target are examined using this same method.*
4. *After processing the prerequisites, the target may or may not need to be rebuilt:*
    a. *If the target does not need to be rebuilt, the path to the file found during directory search is used for any prerequisite lists which contain this target. In short, if make doesn't need to rebuild the target then you use the path found via directory search.*
    b. *If the target does need to be rebuilt (is out-of-date), the pathname found during directory search is thrown away, and the target is rebuilt using the file name specified in the makefile. In short, if make must rebuild, then the* ***target is rebuilt locally, not in the directory found via directory search****"* (Stallman et al., 2004)

Stěžejní části jsou zvýrazněny tučně. Tím, že cíle jsou vytvořeny lokálně, je možné nastavením proměnné VPATH na kořen adresáře obsahujícího zdrojové kódy a spuštěním build procesu z jiného venkovního adresáře, kde chceme, aby byly výstupní soubory vytvořeny, oddělit výsledné sestávané soubory od stromu zdrojových kódů.

Výhodou této techniky je, že při psaní build skriptu není třeba vůbec uvažovat o tom, kde by měly ležet výstupní soubory a je možné psát build skripty čistě v kontextu stromu zdrojových kódů. Oddělení samotné je realizováno mimo build skripty prostě tím, že se build proces spustí z venkovního adresáře a nastaví se proměnná VPATH na patřičnou hodnotu.

V doméně nástroje SCons lze tento princip realizovat stejným způsobem s tím, že ekvivalentní náhrada za VPATH je SCons API metoda Repository().

Tato technika má ovšem nevýhodu v tom, že všechny Buildery použité v build systému musí s použitím této techniky počítat. Příkladem může být defaultní builder Program(). Pokud se program linkuje s knihovnou K, která se nachází např. v adresáři "toplevel/m/" je nutné, aby byl kompilátoru předán argument se search cestou pro tuto knihovnu ve formě "-Ltoplevel/m". Z podstaty VPATH techniky je ale v případě jejího použití nutné, aby kromě této cesty, byla ještě přidána ekvivalentní cesta ve stromu zdrojových kódů, takže výsledný příkaz by měl vypadat nějak takto:

$: gcc –o prog –Ltoplevel/m –Lroot_of_source_tree/toplevel/m –lK main.c

Jinými slovy, všechny uživatelsky definované buildery (zabudované buildery už s použitím Repository počítají) je nutné navrhnout s ohledem na možné použití této techniky.

Jde především o korektní zpracování argumentů, jejichž přítomnost implikuje vytvoření implicitní závislosti. Explicitní závislosti, v příkladu výše tedy soubor *main.c*, jsou vyřešeny korektně vždycky, protože SCons jednoduše dá na příkazový řádek "správnou cestu" k existujícímu souboru. U implicitních závislostí toto možné není, protože argumenty příkazu nejsou obecná filesystémová jména" ale prosté řetězce. SCons tedy nemůže jejich podobu automaticky měnit. V příkladu výše SCons toto vyřeší přidáním alternativní cesty "–*Lroot_of_source_tree/toplevel/m*". To je korektní, protože SCons nemění žádné cesty ani argumenty, ale pouze přidává na příkazovou řádku nový argument.

Kromě nevýhody spojené se složitou implementací nových, uživatelsky definovaných builderů, je problémem, že ne vždy je pro nějaký nástroj vůbec možné podporu VPATH techniky implementovat. V příkladu výše to možné bylo, protože gcc definuje mechanismus přidávání search cest pomocí parametru –L, obecně ale platí, že pro nějaký nástroj může přidání specifických argumentů způsobit vytvoření implicitních závislostí a přitom nástroj nebude poskytovat žádný mechanismus podobný jako –L u gcc.

4.7.2 Separování pomocí prefixování

Tato technika není na první pohled tak elegantní jako technika VPATH, ale je robustnější a obecnější, a proto bude preferovanou technikou pro navrhovaný build systém. Oddělení se provede tak, že před každý cílový soubor se připojí cesta, která tento soubor vydělí ze stromu zdrojových kódů.

Ačkoliv by bylo možné prohlásit, že tento "objektový prefix" musí specifikovat cestu, která je úplně mimo strom zdrojových kódů a docílit tak "úplného" oddělení stromu strojových zdrojových kódů od stromu výsledných objektů stejně jako u techniky VPATH, navrhovaný build systém zvolí poněkud odlišný přístup.

Build systém deklaruje, že každý výstupní objekt musí být prefixován řetězcem (resp. adresářem) "*build/variant_prefix*". Tím je zaručeno, že ať už se bude přeložený výstupní soubor nacházet ve stromu zdrojových kódů kdekoliv, vždy bude mít někde ve své cestě adresář *build*.

Bylo by poněkud nešikovné manuálně prefixovat všechny specifikace cílů napříč všemi build skripty v build systému. Proto je toto prefixování vyřešeno automaticky upravením API metody *Sconscript()* tak, že toto volání bude navíc automaticky přidávat prefix "*build/*" do řetězce specifikovaného argumentem variant_dir.

Všechny cílové soubory se tedy nyní nachází pod adresáři "*build*" respektive "*build/variant_prefix*", které jsou vytvářeny v jednotlivých adresářích zdrojových kódů jednotlivých komponent. Problémem je, že tyto adresáře se stále nacházejí uvnitř stromu zdrojových kódů, takže většina benefitů spojených se separováním zdrojových kódů od přeložených souborů zůstává nevyužita.

Nyní je možné využít skutečnosti, že všechny přeložené objekty v celém build systému mají někde ve své cestě obsazen adresář "*build/variant*". Vzhledem k tomu, že

tento adresář je vytvářen build systémem, tedy nad vytvářením tohoto adresáře má build systém plnou kontrolu, je možné vytvořit adresář build automaticky jako symbolický link do adresáře nacházejícího se v **build area** buildovacího stroje. Navíc je možné, aby adresář nacházející se v build area kopíroval adresářovou strukturu rodičovského adresáře, adresáře *build* nacházejícího se ve stromu zdrojových kódů.

Tímto je docíleno toho, že všechny výsledné objekty jsou uloženy v jednom stromu, nacházejícím se v build area daného buildovacího stroje a tento strom je zcela oddělen od stromu zdrojových kódů. Navíc je možné nastavit celý strom zdrojových kódů jako read only, protože zápis probíhá jen v build adresářích, které jsou ale symbolickými linky do build area.

4.8 Procesy mimo závislostní systém

Ačkoliv je možné jakoukoliv operaci nebo objekt spjatý s procesem překladu zahrnout do stromu AOG a reprezentovat je zde pomocí akcí objektu a závislostí, je pro některé takové operace či objekty z důvodů jejich speciálního významu lepší je nezahrnovat do závislostního systému respektive do stromu AOG generovaného nástrojem pro řízení překladu. Tato kapitola zmíní dvě takové situace.

4.8.1 Externí závislosti

Externí závislosti jsou závislosti na objektech, které nespadají do softwarového projektu. Jedná se především o externí komponenty, na kterých je software závislý, jako například knihovny, hlavičkové soubory, knihovny tříd atd.

Externí závislost už ze své podstaty figuruje jako prerekvizita pro celý softwarový projekt. Jinými slovy před samotným sestavováním našeho softwaru je nutné mít tuto externí závislost.

Dalším důležitým atributem externí závislosti je, že externí závislost je statická tzn. soubory reprezentující externí závislost se typicky nemodifikují, a to ani na základě uživatelských zásahů ani jako výsledek modifikace objektů, které tvoří pro tyto soubory závislosti. Ve skutečnosti by modifikace na základě změn v závislostech ani nebyla možná. Těžko si asi lze představit, že by se v rámci nějakého softwaru, který závisí na standardní knihovně *libc*, inicializoval rebuild této knihovny jen proto, že někdo udělal změnu do hlavičkového souboru *stdio.h*, který je součástí stejného instalačního balíčku jako *libc* a na kterém *libc* řekněme závisí –není možné v rámci jednoho softwaru konstruovat strom závislostí AOG pro cizí software prostě proto, že ho neznáme.

Z těchto důvodů by bylo zbytečnou komplikací zanášet externí komponenty do AOG. V doméně nástroje Scons by to obnášelo definování builderů pro objekty externí komponenty a korektní zpracování objektu externí komponenty pomocí těchto builderů.

Kromě toho, že by nejspíš bylo nutné použít API funkci *Glob()*, protože není možné znát a definovat explicitně obsah externí komponenty (obsah se může změnit, např. v nové verzi SDK), bylo by vnášení všech objektů externí komponenty do stromu AOG při každém spuštění procesu překladu zbytečné (kvůli statické povaze externí závislosti). Takové vnášení by bylo i náročné, protože externí komponenta může obsahovat tisíce hlavičkových souborů, a knihoven, přičemž v daném softwarovém projektu je z nich použito jen velmi malé procento. Spracování velkého množství uzlů má samozřejmě vliv na rychlost (obzvlášť pokud jsou těmito uzly hlavičkové soubory, u kterých se provádí scanning implicitních závislostí).

Z důvodů uvedených výše proto navrhovaný build systém neintegruje externí komponenty do závislostního systému (tedy do AOG). Místo toho definuje rozhraní mezi externími závislostmi a softwarovým projektem v podobném smyslu jako bylo popsáno v kapitole 4.5.2 a dále definuje samostatný instalační skript, který se stará o nasetupování externích komponent tak, aby byly dostupné v samotném procesu překladu software.

Rozhraní je realizováno tak, že se v rámci build systému definuje adresář "*externals*" a do tohoto adresáře se nainstalují jednotlivé externí komponenty a to takovým způsobem, aby odpovídaly rozhraní, dohodnutému pro jednotlivé typy souborů.

Jako modelový příklad, nechť externí komponenta K je distribuována ve formě SDK, které obsahuje několik hlavičkových souborů a několik knihoven. Celý postup je pak následující:

1. Instalační skript získá balíček SDK, například stažením z webu nebo zkopírováním ze vzdálené lokace uvnitř vnitro-firemní sítě.
2. Instalační skript zkopíruje knihovny z K do adresáře *externals/lib/<variant_name>*
3. Instalační skript zkopíruje knihovny z K do adresáře *externals/include/<variant_name>*
4. V build skriptech je mezi standardní cesty LIBPATH respektive CPPPATH přidána cesta *externals/lib/<variant_name>* respektive *externals/include/<variant_name>*

Nabízí se otázka, proč vůbec vytvářet takovou úroveň indirekce a proč nezapomenout na externí závislosti a neinstalovat je jako jakýkoliv jiný software do standardních míst jako je např. */usr/lib nebo /usr/include*. Toto má několik důvodů.

Prvním důvodem je, že externí komponenty ve firemním prostředí často nemají definované žádné instalační rozhraní a jsou distribuovány jako např. zabalený archiv souborů. Je proto nutné je "nějak" zpracovat. Zkopírování souboru do standardních lokací není ideální řešení. Tím že externí komponenta nemá definováno instalační rozhraní, nemá definováno ani odinstalační či aktualizační rozhraní.. Instalací do standardních cest by se tak ztratila kontrola nad těmito soubory.

Druhým důvodem je, že navrhovaný build systém podporuje vyváření souběžných variant a to může vyžadovat vědomí o variantách i co se týče externích komponent. Pokud se na jednom stroji budou například kroskompilovat dvě varianty pro dvě různé verze

operačního systému, je potřeba, aby daná varianta využívala správnou verzi externí komponenty. Toto by bylo problematické při použití standardních cest.

Protože navrhovaný build systém je obecně multi-uživatelský, mohlo by docházet ke zbytečnému instalování shodných externích komponent v rámci build stromu jednotlivých uživatelů. To z toho důvodu, že adresář *externals* je obecně lokální pro daný build strom. Toto lze jednoduše vyřešit tím, že se definuje globální adresář *externals*, který je mimo jakýkoliv build strom patřící jednotlivým uživatelům a do kterého se nainstalují potřebné externí komponenty. Pokud je poté instalační skript spuštěn uživatelem v jeho build stromu, externí komponenty jsou do jeho stromu nainstalovány lokálně do adresáře *externals* jen v tom případě, že globální adresář *externals* neexistuje. V opačném případě je místo instalace externích komponent vytvořen symbolický link na globální adresář *externals*.

4.8.2 Integrace s nástroji pro správu revizí

SCons poskytuje out-of-the-box prostředky pro integraci s nástroji pro správu revizí. Tyto prostředky poskytují skrze API možnost, jak získávat zdrojové entity z repositáře systému pro správu revizí.

Podpora pro tuto integraci se skládá ze dvou základních prvků. Prvním je API metoda Environmentu SourceCode() a druhým prvkem je Builder, specifický pro konkrétní nástroj pro správu revizí. Volání SourceCode() přebírá dva argumenty. Prvním je seznam adresářů případně seznam konkrétních souborů, které je potřeba získat z nástroje pro správu revizí a druhým je právě Builder definující nástroj pro správu revizí. Tento builder přebírá jako argument konkrétní repositář spravovaný nástrojem pro správu revizí. Výsledná použití můžou vypadat následovně.

```
env.SourceCode('.', env.BitKeeper('/usr/local/BKsources'))
env.SourceCode('src', env.CVS('/usr/local/CVSROOT'))
env.SourceCode('/', env.RCS())
env.SourceCode(['f1.c', 'f2.c'], env.SCCS())
env.SourceCode('no_source.c', None)
```

Verze 1.2 nástroje SCons podporuje následující nástroje pro správu revizí: SCCS, RCS, BitKeeper, CVS.

Navrhovaný build systém tuto vlastnost nástroje SCons nevyužívá. Primárním úkolem nástroje SCons je při každém spuštění vykonstruovat AOG a pak provést průchod tímto stromem a postupně vytvářet cílové objekty. Do tohoto schématu ovšem interakce s nástrojem pro správu revizí příliš nezapadá, jak bude ukázáno nyní.

Metoda *SourceCode()* nevytváří vazbu (závislost) mezi souborem v repositáři nástroje pro správu revizí a mezi odpovídajícím lokálním souborem vytvořeným voláním *SourceCode()*. Vytvoření takové vazby by navíc ani nedávalo dobrý smysl - je nežádoucí,

aby při každém inkrementálním spuštění překladu byl lokální soubor aktualizován verzí nacházející se v repositáři, pokud si takovouto aktualizaci uživatel sám nevyžádá. S repositářem interagují ostatní vývojáři. Mohlo by se tak lehce stát, že při spuštění buildu by se přepsaly některé zdrojové soubory (včetně těch, do kterých uživatel vytvořil nějaké změny a tyto změny přitom ještě nebyly poslány do repositáře) aktuálnějšími verzemi nacházejícími se v repositáři.

Metoda *SourceCode()* proto funguje spíš jako pomocná berlička, která zajišťuje, že pokud soubor neexistuje, tak se stáhne z repositáře. Vzhledem k tomu, že ale standardně uživatel provádí check-out celého stromu (pomocí specifických utilit poskytovaných samotným nástrojem pro správu revizí), který zajistí, že všechny potřebné soubory budou vytvořeny, nebude navrhovaný build systém metodu *SourceCode()* využívat a veškerou interakci s nástrojem pro správu revizí ponechá na specifických nástrojích systému pro správu revizí. Tím se také zabrání vzniku duality, kdy check-out by se prováděl skrze build skripty respektive skrze spuštění build procesu a ostatní operace jako je check-in by zůstaly na nástrojích systému pro správy revizí.

Dalším argumentem proti používání metody *SourceCode()* je skutečnost, že tím, že získávání zdrojových souborů je definováno v build skriptech (které jsou typicky součástí stejného stromu jako zdrojové soubory), je nutné při prvotním check-outu nejprve "nějak" získat build skripty, aby je bylo možné spustit. Tyto build skripty se tak můžou postarat o check-out zbylého stromu. Jinými slovy je tu zase dualita – proč napřed stahovat manuálně build skripty a až pomocí nich stáhnout zbytek stromu, když je možné stáhnout celý strom najednou.

5 Nasazení

Navrhovaný build má již za sebou jedno nasazení. Je použit jako build systém pro produkt CA Unicenter NSM společnosti CA (Computer Associates).

NSM (Network and Systems Management) je komplexní balík software sloužící k monitorování a správě rozsáhlých vnitrofiremních IT infrastruktur. Tento software se skládá z velkého množství komponent zahrnující instalované agentní služby, distribuční servery, terminály pro správu, databázové komponenty, webové služby a rozhraní atd.

Instalovaná agentní část je podporována na velkém množství platforem (tabulka přílohy A). V NSM se používá velké množství technologií. Programovací jazyky použité v tomto software zahrnují C, C++, Python, Java, SQL, Perl, Bash, Assembler. Vývojářská základna čítá několik týmu rozprostřených po celém světě.

NSM je rozsáhlý balík a navrhovaný build systém je v současné době použit jen pro jeho agentní část (která je však podporována na nejvíce platformách).

5.1 Motivace

Iniciativa na vznik navrhovaného build systému vznikla z potřeby vyřešit špatný stav toho stávajícího. Původní build systém byl postavený na nástroji Make na UNIXových systémech a na nástroji Visual Studio na platformě Windows. Tato dualita způsobovala množství potíží a nekonzistencí, protože změny prováděné v jednom build systému musely být ručně přenášeny do druhého.

Build systém nebyl nijak standardizován ani formalizován a vyvíjel se živelně. Jednotlivé části build systému prošly pod rukama velkému množství lidí a každý si podle své potřeby a „svého stylu“ přidával jednotlivé fragmenty systému.

Chybějící formalizace v kombinaci s nedostatečným obecným vhledem při vytváření build systému a s jeho dualitou vyústila v situaci, kdy veškerá interakce s build systémem značně brzdila vývoj samotného software. Hlavními problémy bylo:

- Nespolehlivost – strom závislostí byl porušený či nekonzistentní. Inkrementální build se nedal bezpečně použít. Jediný způsob, jak zaručit, že změna byla propagována do všech potřebných souborů, bylo provedení **clean upu** a poté provedení čistého buildu.
- Rozšiřitelnost – přidávání nových prvků stálo značné úsilí. Nejvíce se projevovalo při přidávání nových podporovaných platforem.
- Nesrozumitelnost – každý nový vývojář měl problémy sžít se s tímto systémem. Množství nelogických procesů vytvářelo ze systému komplikované bludiště

- Neefektivnost – používání build systému vyžadovalo množství zbytečných manuálních kroků.

5.2 Přechod

Samotné nasazení probíhalo ve dvou fázích. V první fázi se vytvořil build systém popsaný v návrhu a oba systémy fungovaly souběžně. Původní systém se nadále používal pro produkční buildy a nový systém se používal pro testování a rutinní vývojářské úkony. V druhé fázi, jakmile by ukončen vývoj stávající verze softwaru a začal nový cyklus vývoje nové verze, se přešlo na nový systém i pro produkční buildy.

Je zajímavé, že samotné vytvoření nového build systému na základě toho původního nebylo tak náročné, jak by se mohlo na první pohled zdát. Náročné bylo získat stabilní a dostatečně obecný návrh, který by splňoval všechny potřebné požadavky. Tento návrh byl prezentován v této práci. Po vytvoření návrhu byla hrubá transformace provedena tak, že se vzaly build logy původního build systému z jednotlivých buildovacích strojů a v novém build systému se pomocí algoritmu RZU a postupů popsaných v kapitole 4.55 definovaly jednotlivé entity tak, aby výsledný výstup nového build systému zhruba odpovídal výstupu toho starého. Jakmile byla hrubá kostra hotová, přidávaly se chybějící parametrizace a dolaďovaly se detaily. Tento způsob využívá skutečnosti, že konfigurační vrstva build systému je statická a build logy z oficiálních buildovacích strojů jsou jediné platné. Ve sféře open source by díky značně komplikovanější konfigurační vrstvě build systému byl přechod složitější.

5.3 Přijetí

Systém byl vývojáři přijat pozitivně a přinesl očekávané zefektivnění vývoje produktu. Za dobu jeho používání se nashromáždilo několik připomínek, které by bylo dobré brát v úvahu a které můžou poskytovat potenciál pro další zlepšení návrhu.

- Využitím optimalizačních featur nástroje SCons lze dosáhnout značného zlepšení, ~~ale~~ Pro perfektní použitelnost by bylo SCons potřeba dále optimalizovat. Tento problém se týká jen inkrementálních buildů.
- Zdá se, že je trochu problematické spravovat Cross exporty/importy síťových filesystémů popsaných v kapitole 4.2.1. Implementace těchto síťových služeb se liší systém od systému a ne všude je implementace zvládnuta dobře.
- Rychlé inkrementální testování je definováno jen pro testovací stroje, které jsou součástí interního vývojového prostředí. Inkrementálně je občas potřeba testovat i mimo toto prostředí (např. na strojích týmu zajišťujícího kvalitu nebo u zákazníka).

6 Závěr

V práci se podařilo vytvořit návrh build systému, vhodného pro nasazení v multiplatformním software vyvíjeným komerčními organizacemi. Ohled byl brán kromě těchto atributů i na co největší obecnost a tedy aplikovatelnost v co nejširším spektru projektů.

Práce definovala kontext. přinesla přehled dostupných technologií v oblasti řízení překladu software. Z tohoto přehledu byl pro použití pro jádro build systému vybrán nástroj SCons. Ten splňuje požadavky, vyžadované pro třídu softwarových projektů, kterými se práce zabývá a svým charakterem umožňuje dostatečně obecný návrh build systému.

Analýza problematiky ukázala, jak problém správně uchopit a analyzovala klíčové body build systému. Analýza se také věnovala zkoumání dvou odlišných přístupů k samotnému řízení překladu software a ukázala, že přístup zvolený nástrojem SCons má své nesporné výhody.

Návrh byl pojat formou nalezení vhodných řešení pro klíčové otázky a také formou poskytnutí postupů pro vytváření build systémů. Podařilo se vytvořit systém, který vytváří jednotný strom závislostí a současně řeší problém paralelizace, který vznikne při vytvoření jednotného stromu závislostí.

Navrhovaný build systém byl v praxi nasazen pro softwarový produkt NSM a byl vývojáři pozitivně přijat. Ukázalo se, že praktická zkušenost s reálným projektem byla neocenitelná pro vytvoření prezentovaného návrhu. Umožnila soustředit se na skutečné problémy, vyskytující se v build systémech a přinést takový návrh, který neopomíjí požadavky či předpoklady, které nejsou na první pohled zřejmé.

Seznam použité literatury

[1] STALLMAN, Richard M., Roland MCGRATH a Paul D. SMITH. *GNU Make*: *A Program for Directing Recompilation*. 3.81. Boston: Free Software Foundation, 2006. 0.70. ISBN 1-882114-83-5. Dostupné z: https://notendur.hi.is/jonasson/software/make-book/make-manual.pdf

[2] KNIGHT, Steven a Anthony ROACH. Scons - a software construction tool. In: *Http://scons.org/* [online]. December 21, 2008 [cit. 2012-03-11]. Dostupné z: http://scons.org/doc/1.2.0/HTML/scons-man.html#lbBW

[3] MILLER, Peter. *Recursive Make Considered Harmful* [online]. 10 March 2008 [cit. 2012-04-11]. Dostupné z: http://aegis.sourceforge.net/auug97.pdf

[4] MECKLENBURG, Robert, Robert WILLIAM a Andrew ORAM. *Managing projects with GNU make*: *Use the Power of GNU Make to Build Anything*. 3. vyd. USA: O'Reilly Media, Inc., 2004. ISBN 978-0-596-00610-5.

Seznam použitých zkratek

ABI - application binary interface

AOG - orientovaný acyklický graf

BS - Build Systém

CA- Computer Associates

IDE – Integrated development environment

NSM - Network and Systems Management

RZU - Rozdělení podle Zdrojových Uzlů

SCM – Software Configuration management

SP – Softwarový projekt

Příloha A

Platformy podporované produktem NSM

Platform Name	Minimum SP level	O/S Addressability	Hardware Architecture	Agent Binary Addressability
Windows XP Professional	SP2	32-bit	Intel x86	32-bit
Windows Server 2003 Standard, Enterprise, Data Center & Small Bus Svr	SP2	32-bit	Intel x86; AMD-64; EM64-T	32-bit
Windows Server 2003 x64 Edition	SP2	64-bit	AMD-64; EM64-T	64-bit
Windows Server 2003 IA-64 Edition	SP2	64-bit	Itanium-2	64-bit
Windows Server 2003 R2 Edition Standard, Enterprise, Data Center	SP2	32-bit	Intel x86; AMD-64; EM64-T	32-bit
Windows Server 2003 R2 x64 Edition Standard, Enterprise, Data Center	SP2	64-bit	AMD-64; EM64-T	64-bit
Windows Server 2008 x86 (All Versions)	Base	32-bit	Intel x86; AMD-64; EM64-T	32-bit
Windows Server 2008 x64 Edition (All Versions)	Base	64-bit	AMD-64; EM64-T	64-bit
Windows Server 2008 IA-64 Edition (All Versions)	Base	64-bit	Itanium-2	64-bit
Windows Vista x86 (Business, Enterprise, Ultimate)	SP1	32-bit	Intel x86; AMD-64; EM64-T	32-bit
Windows Vista x64 (Business, Enterprise, Ultimate)	SP1	64-bit	AMD-64; EM64-T	64-bit
Redhat Enterprise Server 4.0 x64 WS, ES and AS		64-bit	AMD-64; EM64-T	32-bit
Redhat Enterprise Server 4.0 IA-64 WS, ES and AS		64-bit	Itanium-2	64-bit
Redhat Enterprise Server 5.0 x86 WS, ES and AS		32-bit	Intel x86; AMD-64; EM64-T	32-bit
Redhat Enterprise Server 5.0 x64 WS, ES and AS		64-bit	AMD-64; EM64-T	32-bit
Redhat Enterprise Server 5.0 IA-64 WS, ES and AS		64-bit	Itanium-2	64-bit
SuSE Enterprise Server 9.0 x64		64-bit	AMD-64; EM64-T	32-bit

SuSE Enterprise Server 9.0 IA-64		64-bit	Itanium-2	64-bit
SuSE Enterprise Server 10.0 x86		32-bit	Intel x86; AMD-64; EM64-T	32-bit
SuSE Enterprise Server 10.0 x64		64-bit	AMD-64; EM64-T	32-bit
SuSE Enterprise Server 10.0 IA-64		64-bit	Itanium-2	64-bit
Solaris 8 UltraSPARC 64-bit		64-bit	UltraSPARC	64-bit
Solaris 8 x86		32-bit	Intel x86; AMD-64; EM64-T	32-bit
Solaris 9 UltraSPARC 64-bit		64-bit	UltraSPARC	64-bit
Solaris 9 x86		32-bit	Intel x86; AMD-64; EM64-T	32-bit
Solaris 10 UltraSPARC 64-bit		64-bit	UltraSPARC	64-bit
Solaris 10 x86		32-bit	Intel x86; AMD-64; EM64-T	32-bit
Solaris 10 x64		64-bit	AMD-64; EM64-T	64-bit
HP-UX 11.11 PA-Risc 64-bit		64-bit	PA-Risc-64	64-bit
HP-UX 11.23 PA-Risc 64-bit		64-bit	PA-Risc-64	64-bit
HP-UX 11.23 IA-64		64-bit	Itanium-2	64-bit
HP-UX 11.31 PA-Risc 64-bit		64-bit	PA-Risc-64	64-bit
HP-UX 11.31 IA-64		64-bit	Itanium-2	64-bit
AIX 5.2		32-bit	POWER	32-bit
AIX 5.2		64-bit	POWER	64-bit
AIX 5.3		32-bit	POWER	32-bit
AIX 5.3		64-bit	POWER	64-bit
AIX 6.1		64-bit	POWER	64-bit
Tru64 5.1B		64-bit	ALPHA	64-bit