Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Petr Doležal

REVERZNÍ PARALELNÍ KONFIGURAČNÍ MANAŽER

Katedra softwarového inženýrství
Vedoucí diplomové práce: **RNDr. Jakub Yaghob, Ph.D.**

Studijní program: Informatika

Na tomto místě bych chtěl poděkovat vedoucímu své diplomové práce RNDr. Jakubu Yaghobovi, Ph.D. za jeho připomínky, které přispěly k dokončení této práce. Zároveň chci vyjářit poděkování kolegům Janu Červákovi za jeho výborný parser příkazové řádky, který jsem v implementaci s jeho svolením použil, Jiřímu Kosinovi za pomoc, díky níž jsem byl schopen program odladit pro unixovské systémy, a Miroslavu Benešovi za komentáře k textu, které přispěly k jeho lepší čitelnosti a srozumitelnosti.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 3. 11. 2005

Petr Doležal

Obsah

Název práce: Reverzní paralelní konfigurační manažer
Autor: Petr Doležal
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
e-mail vedoucího: Jakub.Yaghob@mff.cuni.cz

Abstrakt: Existující nástroje pro správu překladu a konfigurací softwarových projektů trpí často různými nedostatky, typickými problémy jsou např. složitý zápis konfigurace projektu nebo obtížná přenositelnost popisu konfigurace projektu mezi různými platformami. Tato práce analyzuje výhody a nevýhody přístupů používaných existujícími implementacemi a navrhuje řešení založené na kombinaci osvědčených i zajímavých myšlenek různých existujících návrhů. Významnou součástí práce je implementace plně funkčního programu podle předloženého návrhu, což dovoluje návrh otestovat v praxi. Cílem implementace je vyhnout se nedostatkům existujících podobných programů při zachování jejich předností.

Klíčová slova: správa překladu, konfigurace softwarového projektu, varianty překladu

Title: Reverse Parallel Configuration Manager
Author: Petr Doležal
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: Jakub.Yaghob@mff.cuni.cz

Abstract: Existing tools for build and configuration management of software projects often suffer from various drawbacks, the typical problems include, for example, a complicated notation of a project's configuration or difficulties with the portability of a configuration description between various platforms. This work analyses the advantages and disadvantages of the approaches used by existing implementations and it proposes the solution that is based on merging the well-established and interesting ideas of various existing solutions. An important part of the work is the implementation of a fully functional program according to the proposed design, which allows to test the solution in practice. The goal of the implementation is to avoid the disadvantages of existing similar programs while keeping their advantages.

Keywords: build management, software project configuration, build variants

1. Úvod

S rostoucí složitostí softwarových projektů vzrůstají i nároky na správu jejich vývoje, řízení jejich překladu i na instalaci výsledného produktu. Již na přelomu 70. a 80. let se proto objevily první nástroje, které měly s těmito úkoly vývojářům pomoci. Byly tak položeny základy nového odvětví softwarového inženýrství nazývaného software configuration management (SCM).[1]

Komplexnost softwarových systémů se od té doby značně zvýšila a s ní i význam SCM, stejně jako se zvýšily i požadavky kladené na moderní SCM systémy a nástroje. Nyní se SCM systémy snaží řešit, více či méně úspěšně, širokou škálu problémů, které plynou z požadavků jejich uživatelů. Požadavky na SCM systémy lze rozdělit do dvou velkých skupin.

První skupinou úloh, se kterými se SCM systémy potýkají, je řízení procesu vývoje (process control). Tato funkce SCM systému slouží jako podpora pro řízení týmu vývojářů a má spíš administrativní charakter. Vedle samotného plánování a řízení vývojového cyklu projektu může zahrnovat také např. audit vykonané práce, vytváření zpráv o projektu nebo reflexi zpráv od uživatelů. Řízení procesu vývoje je prostředkem, jak udržet a zaručit danou úroveň a kvalitu vývoje, což může hrát důležitou roli v různých certifikačních procesech.

Druhá skupina úloh, kterou se SCM systémy snaží řešit a o které bude pojednávat zbytek této úvodní kapitoly, se týká vývojářovy rutinní činnosti. Zahrnuje vedle vlastní správy konfigurací (configuration management) také správu pracovního prostředí (workspace control) a řízení překladu, resp. vytvoření výsledného produktu (build management).

Ústřední komponentou SCM systému, s níž souvisejí pojmy konfigurace a správa konfigurací, je repositář (repository). Primárním úkolem repositáře je uchovávat všechny části projektu (zdrojové texty, moduly a další komponenty projektu) a sdílet je mezi vývojáři. Jelikož repositář slouží jako úložiště pro softwarové firmy citlivých a životně důležitých dat, musí být velmi spolehlivý; v ideálním případě by měl také nabízet autentizační a autorizační mechanismy pro řízení přístupu k uloženým objektům. Vedle spolehlivosti a bezpečnosti uložených dat je požadováno i jejich úsporné uložení, neboť počet uchovávaných objektů může dosahovat desítek až stovek tisíc. Protože repositář slouží pro sdílení dat velkému počtu uživatelů, musí také podporovat paralelní přístup, v některých případech může dokonce nastat potřeba, aby repositář byl distribuovaný. Všechny tyto požadavky se v podstatě kryjí s požadavky na databázový systém, ale navíc k nim přistupují požadavky další, které jsou již specifické pro SCM a které komplikují nasazení běžných databázových systémů a postupů [4] [6].

Hlavním z těchto specifických požadavků je nějaká forma verzování objektů (versioning, versioning control). Verzování dovoluje uchovávat historii celého projektu, a sledovat tak změny, které v průběhu jeho vývoje byly učiněny. Pokročilejší techniky

[1] SCM se často označuje jen zkráceně CM (configuration management).

vývoje a verzování pak zahrnují větvení vývoje a paralelní vývoj několika verzí, přejímání výsledků paralelních vývojových větví do hlavní větve a slučování větví. Větvení vývoje (branching) dovoluje pokračovat v údržbě starších verzí produktu či ve vytváření experimentálních vývojových větví, aniž by byla narušena hlavní větev – tedy dovoluje izolovat od sebe různá stádia vývoje. Slučování (merging) naopak umožňuje přenášet výsledky vývoje mezi větvemi, např. výsledky experimentální větve zařadit do hlavní vývojové větve nebo zařídit přenos opravy chyb z některé „udržovací“ větve do větví ostatních.

Existence více variant některých částí projektu přináší možnost tyto varianty různě kombinovat a sestavovat tak, aby bylo dosaženo požadovaných vlastností výsledného produktu – produkt např. může existovat ve variantách pro různé platformy či v různých škálách poskytovaných služeb a funkcí. Popis a výběr variant produktu je záležitostí správy konfigurací.

Konfiguraci lze definovat např. jako sadu souborů, které dohromady vytvoří požadovaný produkt [6]. Práce s konfigurací má dvě roviny. První rovinou je výběr správných variant souborů, které do zadané konfigurace patří a z nichž má být požadovaná varianta produktu vytvořena. Druhou rovinou je pak vlastní vytvoření produktu z vybraných souborů a s ohledem na požadované vlastnosti výsledku.

Je zřejmé, že výběr správných variant souborů se úzce dotýká repositáře a záleží na jeho možnostech a způsobu, jakým lze s repositářem pracovat. Základní modely přístupu k repositáři, způsobu verzování a větvení vývoje popisují [7] a [4], v [5] je uveden přehled několika SCM systémů a jejich konceptů.

Na způsobu práce s repositářem silně závisí i způsob, jakým se odehrává vlastní vytvoření produktu a který se musí možnostem repositáře přizpůsobit. Velké integrované SCM systémy se starají o obě stránky správy konfigurací, a tak mohou lépe sladit build management s možnostmi repositáře. Samostatné nástroje zpravidla s repositářem přímo nespolupracují a soustřeďují se na build management.

1.2 Cíle práce

Z úvodní pasáže je patrné, že SCM představuje rozsáhlou problematiku a SCM nástroje a systémy se musí vypořádat se širokou škálou problémů. Pozornost v této oblasti je věnována vedle procesu řízení vývoje hlavně modelům sestavování konfigurací a správy repositáře – cílem výzkumu na tomto poli je dostatečně univerzální model, který by sloučil přednosti dosud prozkoumaných přístupů a byl snadno přizpůsobitelný konkrétním potřebám.

Ačkoliv repositář je právem ústředním tématem SCM, nelze opomíjet ani build management a problematiku nástrojů pro řízení a automatizaci kompilace projektů (build managers)[2], na niž je zaměřena tato práce. Nástroje zabezpečující tuto funkci jsou součástí každého SCM systému a během vývoje odvětví SCM jich vznikly celé desítky.

[2] Pro tyto nástroje se užívá řada označení, žádné ustálené a jednoznačné pojmenování zřejmě neexistuje. Častá označení jsou konfigurační manažer (configuration manager), které je však méně přesné a užívá se i pro jiné druhy programů, product builder, project builder, (software) build tool, build management system, build manager aj. V souladu s úvodní kapitolou bude používán pro tyto nástroje důsledně termín build manager.

Většina implementací však trpí řadou opakujících se nedostatků. Cílem této práce je navrhnout a implementovat samostatný build manager, který by neměl trpět většinou nedostatků existujících obdobných nástrojů. Důležitým požadavkem na implementaci je nezávislost na operačním systému a hardwarové platformě.

1.3 Struktura práce

Podrobné analýze problému, zejména pak stručnému popisu důležitých rysů několika vybraných build managerů, je věnována druhá kapitola práce. Na základě zkoumání popsaných programů je předložen návrh implementace, který se snaží sloučit jejich výhody a vyhnout se jejich nevýhodám. Principy i návrh implementace jsou popsány ve třetí kapitole. Čtvrtá kapitola je věnována již vlastní implementaci a jejím výsledkům. Implementace samotná je k dispozici na přiloženém disku CD-ROM, stejně jako přílohy (např. uživatelská příručka), které z důvodu svého rozsahu jsou k dispozici pouze v elektronické podobě.

2. Analýza problému

Build managery lze rozdělit do dvou skupin podle toho, zda jsou implementovány jako samostatně použitelné programy, nebo zda jsou integrovány do nějakého prostředí jako jeho nedílná součást.

Jestliže je build manager implementován jako samostatný program, nemá jinou možnost, než specifické funkce, jako např. překlad zdrojového textu, svěřit externím programům. O přesném fungování těchto programů nemá build manager bližší informace a zpravidla ani na ně neklade zvláštní nároky. Uživatel si je při používání takového build manageru vědom přechodů mezi jednotlivými kroky, které build manager koná, a vidí rozhraní mezi nástroji, které používá. Tento přístup se označuje jako na nástrojích založený přístup (tool-based approach, resp. jako loosely coupled model).

Opakem popsaného tool-based přístupu je integrované prostředí, ve kterém jsou všechny funkce a nástroje spolu více či méně provázané, navzájem se znají a jejich případná spolupráce je před uživatelem při běžném používání skrývána. Build manager je v prostředí integrován a build management je pak jen další funkcí prostředí, o kterou se uživatel nemusí nijak zvlášť explicitně starat, nebo dokonce do něj ani nemůže výrazněji zasahovat.

Nelze upřednostnit jen jeden přístup, oba přístupy mají své výhody i nevýhody. Ve většině případů jsou výhody a nevýhody jen relativní vzhledem k požadavkům vývojáře a přístupu, který vývojář preferuje. S ohledem na zaměření a cíle této práce budou však dále rozebírány jen samostatné build managery.

2.1 Cíle build managerů

Korektní aktualizace produktu

Nejdůležitějším úkolem build manageru je na základě popisu konfigurace projektu zajistit aktualizaci odvozených objektů (derived objects), což jsou objekty vzniklé činností kompilátorů, preprocesorů, linkerů a dalších programů, které budou souhrnně označovány v dalším textu jako aktualizační nástroje. Mezi tyto odvozené objekty patří i výsledný produkt.

Konfigurace projektu specifikuje seznam zdrojových objektů, které se mají zpracovat. Tyto objekty mohou mít mezi sebou různé druhy závislostí přenášených i do odvozených objektů. Je tedy nutné tyto závislosti respektovat a spouštět aktualizační nástroje ve správném pořadí, aby výsledek byl korektní. Typickým příkladem takových závislostí je statická závislost programových modulů na rozhraních svých i na rozhraní jiných modulů: v případě změny rozhraní je nutné závislé moduly znovu zpracovat, aby tyto změny byly zaneseny do odvozených objektů.

Automatizace aktualizačního procesu a automatizace hlídání závislostí snižují riziko chyb a opomenutí vývojáře. Místo opakovaného vyvolávání příkazů k aktualizaci přímo vývojářem stačí, aby vývojář své požadavky zadal do popisu konfigurace a provedení aktualizace ponechal na build manageru.

Minimalizace počtu aktualizačních operací

Pokud jsou části projektu zpracovatelné separátně, lze popis závislostí využít k omezení počtu aktualizačních operací, přičemž aktualizační operací se myslí spuštění příslušného aktualizačního nástroje. Nutná je aktualizace pouze těch částí projektu, které jsou závislé na změněných vstupech. Build manager by měl být schopen tyto nutné závislosti odhalit a vyvolat jen nutné minimum aktualizačních operací.

Minimalizace celkové doby aktualizace

Jedním z cílů build manageru je zkrátit dobu, po kterou aktualizace probíhá. Hlavním prostředkem k dosažení tohoto cíle je minimalizace počtu aktualizačních operací, protože spuštění a běh aktualizačního nástroje zabírá nejvíce času. Dalšími možnostmi jsou využití paralelismu pro současnou aktualizaci nezávislých částí projektu a snižování režie spojené s provozem samotného build manageru.

Lepší využití zdrojů

Pod tento cíl spadá v první řadě využití paralelismu, nebo dokonce distribuovaného výpočtu. Paralelní nebo distribuovaný výpočet umožní využít lépe výkon dostupného hardware, a tak může zkrátit celkovou dobu aktualizace. Nezanedbatelné může být i sdílení dalších kapacit, zejména diskového prostoru; spousta dat – zvláště meziproduktů kompilace – existuje v rámci celého vývojářského týmu v mnoha kopiích, které vznikají nezávisle na sobě. Sdílení takových dat může přinést nejen úsporu celkově obsazeného diskového prostoru, ale i další zkrácení doby aktualizace, protože nebudou opakovaně vznikat kopie těchto meziproduktů, nebo aspoň ne jako výsledek běhu příslušného překladače.

Zkrácení latence překladu

Tento požadavek není zcela totožný s minimalizací aktualizačního času, dokonce může být v rozporu se snahou minimalizovat počet aktualizačních operací. Praxe ukazuje, že vývojář by neměl příliš dlouho čekat na výsledek aktualizace, poté co dokončí editaci zdrojových textů. Rozdíl mezi dvěma a deseti minutami čekání je značný – zatímco dvě minuty vývojář dokáže ještě čekat, v případě deseti minut jej už opustí koncentrace a vývojář má tendenci hledat si „jinou zábavu“ [19].

Build manager může pro snížení této prodlevy automaticky spouštět aktualizační operace i dříve, než si je vývojář explicitně vyžádá, např. zatímco vývojář ještě edituje zdrojový kód. Díky tomu mohou být výsledky aktualizace k dispozici mnohem dřív, než kdyby byla aktualizace vyvolána až po skončení editace. Na druhou stranu může nějaká operace ze strany vývojáře (např. uložení rozpracovaného souboru) narušit aktualizační proces a donutit build manager k jeho stornování a opětovnému spuštění.

2.2 Přehled vybraných build managerů

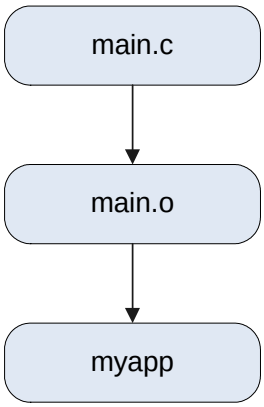
Těžištěm celé této kapitoly je následující přehled několika existujících programů. Vzhledem k tomu, že build managerů existují desítky, je vybrán jen velmi úzký vzorek, jenž by však měl zahrnovat řešení, která měla na vývoj této skupiny nástrojů významnější dopad nebo která zasluhují vzhledem k nějakému svému rysu pozornost.

2.2.1 *make* a *GNU make*

Program *make* [8] vznikl v polovině 70. let, tedy v době, kdy se teprve začala rýsovat potřeba SCM. Návrh *make* je poznamenán dobou jeho vzniku a obsahuje četné nedostatky. Tyto nedostatky inspirovaly vznik celé řady klonů a náhrad *make*. Samotný *make* prošel dlouhým vývojem a doznal četných rozšíření, z nichž některá převzal i od svých klonů. Zřejmě nejvíce rozšířenou a také nejvíce propracovanou současnou implementací *make* je *GNU make*[3].

Fungování *make*

Idea, na níž je *make* založen, je jednoduchá: vývojář potřebuje přeložit sadu zdrojových textů do formy spustitelného souboru nebo knihovny. To zařídí spuštěním překladače, resp. linkeru či podobných nástrojů, které z jednoho nebo více vstupních souborů umí vytvořit výstupní soubor. Soubory, které do překladu vstupují, figurují jako prerekvizity pro cíl překladu. Mezi prerekvizitami a cíli existuje přirozená závislost – změna některé prerekvizity vynucuje aktualizaci příslušného cíle. Každý cíl pochopitelně může představovat prerekvizitu pro jiný cíl, závislosti se tedy mohou řetězit. Prerekvizity, cíle a závislosti mezi nimi tvoří dohromady orientovaný graf – graf závislostí (dependency graph).



Obrázek 1: Graf závislostí. Schéma ukazuje příklad jednoduchého grafu, ve kterém jsou závislosti zřetězené a cíl jedné prerekvizity představuje prerekvizitu pro jiný cíl: *main.o* je cílem pro *main.c* a zároveň prerekvizitou pro *myapp*.

Konfigurační soubor pro *make*, makefile, obsahuje pravidla, která definují cíle a jejich prerekvizity. Součástí pravidel je akce, která dovede z prerekvizit požadovaný cíl vytvořit. Tato akce je uvedena ve formě příkazů shellu, *make* nemá k dispozici žádné

[3] Domovská stránka *GNU make*: http://www.gnu.org/software/make/make.html.

informace o smyslu těchto příkazů a plně se na ně spoléhá – *make* je tedy ukázkovým příkladem tool-based přístupu.

Aktualizační algoritmus *make* je založen na průchodu grafu závislostí do hloubky; graf samotný je jednoduše zkonstruován z popisu dodaného v makefile. S průchodem grafu se začíná od primárního cíle (prvního cíle v makefile nebo cíle zadaného *make* argumentem při jeho spouštění). Průchod grafem je post-order, při zpracování cíle se nejprve řeší rekurzivním způsobem jeho prerekvizity. Rekurzivní průchod se zastaví v případě zacyklení grafu (zacyklení není chyba, pouze se do cyklu nevstoupí), nebo dojde-li se k pravidlu, které již nemá prerekvizity (ani s ohledem na zabudovaná implicitní pravidla). Pokud aktuálně zpracovávaný cíl neexistuje jako diskový soubor, nebo odpovídající soubor je starší (podle data poslední modifikace příslušného souboru) než libovolná jeho prerekvizita, spustí se akce pro zpracovávaný cíl, o které se předpokládá, že příslušný soubor vytvoří nebo aktualizuje.

Strategie, kterou se popsaný algoritmus řídí, se nazývá backward chaining, nebo také demand-driven[4] – název se odvíjí od způsobu procházení cíli, který postupuje zpětně, od primárního cíle k jeho prerekvizitám, a aktualizační akce spouští jen tehdy, pokud se narazí na nekonzistenci v závislostech.

make jako preprocesor

Ve svých pravidlech (a jejich akcích) dovoluje *make* používat své vlastní textové proměnné a s jejich pomocí také podmíněně vybírat bloky makefile, které mají být zpracovány, nebo naopak vynechány. Podstata fungování *make* spočívá vlastně v úpravě shellového skriptu pomocí expanzí proměnných *make* a ve výběru bloků příkazů a výběru pořadí, ve kterém se mají příkazy vykonat. V konečném důsledku se *make* chová jako preprocesor shellovských skriptů.

První důsledek je pozitivní – je jím snadná srozumitelnost makefile (aspoň v jeho základní jednoduché podobě), neboť po uživateli se nevyžadují žádné zvláštní znalosti kromě znalosti shellu.

Druhý důsledek je však negativní a představuje zásadní slabinu: *make* nemá prostředky pro rozlišování různých druhů entit v makefile, zvládne pouze textové manipulace. Nemá např. typ proměnné pro jméno souboru či odkaz na soubor, což je velká nevýhoda u nástroje, který operuje se soubory a řídí podle nich své operace. Práce se soubory je tak vázána na manipulaci s textovými proměnnými, které mohou představovat jméno souboru nebo také cokoliv jiného. Řešení této slabiny se během vývoje *make* soustředilo na její důsledky, nikoliv na příčiny, což je evidentně motivováno snahou zachovat původní myšlenku *make* a zpětnou kompatibilitu. Prvním pokusem o řešení byly nové odvozené automatické proměnné a nová syntaxe pro expanzi proměnné, během níž se provede také textová substituce. Druhým pokusem pak bylo zavedení funkcí. Funkce představují obecnější a silnější nástroj, který však může snížit přehlednost makefile i jeho přenositelnost mezi různými verzemi *make*.

Syntaxe makefile

Syntaxe makefile trpí řadou stinných stránek. Jednou z těchto stinných stránek je řádková orientace makefile a nutnost používání tabulátorů, která je pro uživatele

[4] Termín backward chaining by se dal přeložit jako zpětné řetězení, demand-driven pak jako řízený požadavky.

nepříjemná.[5] Závažnější nedostatek představuje poměrně volná syntaxe, která má svůj původ v koncepci „*make* jako preprocesor“. Jazyk, kterým se makefile zapisuje, není založený na tokenizaci textu (token based), nýbrž na volné interpretaci textu a manipulacích s ním (text based). Syntaktická pravidla jsou značně nejasná, takže se uživateli může snadno podařit zapsat věci, které zapsat vůbec nechtěl. Srozumitelnosti makefile neprospívá ani fakt, že řada konstrukcí má několik možných ekvivalentních tvarů, které se nakumulovaly během dlouhého vývoje programu.

Syntaxe a logika makefile neponechává příliš prostoru pro vytváření nových jazykových konstrukcí a navíc je patrná snaha nové rozšiřující funkce vtěsnat do stávající syntaxe, aby výsledek byl aspoň trochu zpětně kompatibilní. Výsledkem je, že makefile sice má jednotnou syntaxi, jenže stejný syntaktický konstrukt popisuje často sémanticky odlišné věci.[6]

Univerzální pravidla

Univerzální pravidla představují šablony pravidel pro celé skupiny souborů. Typickým příkladem univerzálního pravidla je pravidlo pro překlad zdrojového textu jazyka C do binárního mezitvaru (object file). Pokud je možné univerzální pravidla automaticky řetězit tak, že build manager dovede sám aplikovat několik pravidel za sebou, aby ze zadaného vstupu dostal kýžený výstup, stává se z nich silný nástroj.

Již od počátku nabízel *make* univerzální pravidla, která byla založena na sufixech jmen souborů. Takto definovaná pravidla nebyla dost obecná a nebyla schopná některé situace zvládnout. Později byla nahrazena pravidly označovanými jako pattern rules, která podobnými nedostatky již netrpí a mohou být i automaticky řetězena; takto fungující univerzální pravidla představují již dostatečně silný nástroj.

Implicitní závislosti

V *make* není zabudován žádný vlastní mechanismus pro získávání závislostí mezi soubory plynoucími z jejich obsahu (tzv. implicitní závislosti). Nedisponuje ale ani takovým mechanismem, který by dovedl snadno využívat tyto informace generované externími programy. Implicitní závislosti lze do makefile dodat buď vygenerováním či úpravou makefile dalšími nástroji, nebo zneužitím direktivy *include*, která původně byla určena hlavně pro sdílení uživatelských univerzálních pravidel. Problémy s implicitními závislostmi při použití *make* rozebírá [16].

Viewpathing

Viewpathing funguje tak, že je zadán seznam cest, které se mají prohledávat v případě, že některý soubor zadaný v konfiguraci není nalezen. Díky tomu lze v konfiguraci používat pouze jména souborů bez cest, což lze využít jednak pro úspornější zápis konfigurace a jednak jako primitivní a omezenou náhradu repositáře, kdy změnou cest pro viewpathing lze vybírat různé varianty souborů, aniž by bylo nutné zásadně měnit konfigurační soubor.

[5] Příkazy shellu, jsou-li na samostatném řádku, musí v makefile začínat znakem tabulátor; tento fakt notně komplikuje život nejen začátečníkům.
[6] Zvlášť křiklavým případem jsou speciální cíle (např. .SUFFIX), které existovaly v make již od počátku.

Viewpathing poskytovaný *make* má však omezené možnosti – nelze jej řídit dostatečně jemně, např. na úrovni jednotlivých větví v grafu závislostí, které mohou představovat podprojekty. Je také jedinou podporou, kterou *make* nabízí pro překlad variant.

Výkonnost *make*, jeho používání a škálovatelnost

Vlastností často *make* vytýkanou je spouštění každého příkazu (řádku akce) v nové instanci shellu. To je spojeno jednak s další režií a jednak je to i uživatelsky nepříjemné kvůli nutnosti escapovat konce řádků příkazů pro jednu instanci shellu.[7]

Nepříjemnou vlastností *make* je fakt, že za výsledek aktualizační akce se považuje pouze cíl pravidla, ale není možné deklarovat vyprodukování více cílů jednou akcí. Kvůli tomu se uživatelé *make* spoléhají občas na vedlejší efekty aktualizačních akcí, nebo musí používat různé triky, aby úspěšná a korektní aktualizace nebyla pouze dílem náhody, zvlášť má-li probíhat více akcí paralelně, což *make* také podporuje.

Původně byl *make* navržen pro malé až středně velké projekty a funguje nejlépe tehdy, když jsou všechny zdrojové texty pohromadě s makefile, nicméně se *make* začal používat i na projekty velké, se kterými jeho návrh nepočítá. Navíc se ustálily způsoby jeho použití, které jsou špatné a nevhodné, ale přesto jsou podporovány dokonce i vývojáři *make* – týká se to zejména rekurzivního spouštění *make*. Problémy vzniklé rekurzivním spouštěním *make* a další otázky ohledně jeho výkonu a špatného používání podrobně analyzuje [16].

Bezstavovost a přesnost aktualizace

O vyvolání aktualizační akce rozhoduje *make* pouze na základě porovnání časů poslední změny cíle a jeho prerekvizit, ačkoliv průběh a výsledek akce závisí na mnohem širším kontextu, který by při vší důslednosti musel zahrnout např. i verze všech použitých nástrojů. Je zřejmé, že zachycení celého kontextu, který má vliv na průběh aktualizační akce, je prakticky neproveditelné; *make* se však nestará ani o jeho úzkou podmnožinu, která může přímo ovlivňovat obsah makefile (např. proměnné makefile zadané jako argument *make* či proměnné prostředí, které *make* importuje do vlastních proměnných).

I samotné rozhodování o vykonání aktualizační akce, založené na časových údajích, pracuje dobře, pouze pokud jsou splněny tři nutné předpoklady [10]:

- čas změny souboru je vždy při změně obsahu souboru skutečně nastaven na novou hodnotu,
- časy změn všech souborů se vztahují ke stejným hodinám,
- každá akce při aktualizaci skutečně aktualizuje také čas cíle.

Připočteme-li k těmto faktům i problém se získáváním (a následnou aktualizací) implicitních závislostí, jak bylo zmíněno výše, lze konstatovat, že *make* má vážné nedostatky v plnění základního úkolu build managera, tedy v korektní a přesné aktualizaci.

[7] Někdy je potřeba provedení akce v jediné instanci shellu, aby byly zachovány proměnné prostředí nebo bylo možné podle návratové hodnoty příkazu provést ještě nějakou další akci.

Přenositelnost

Jelikož makefile je pevně svázán se shellem a s jeho příkazy, je vytvoření skutečně přenositelného makefile velký problém a *make* pro to nemá žádnou výraznou podporu. Obtížná přenositelnost a komplikovanost rozsáhlejších makefiles vedla ke vzniku generátorů makefiles.[8] Tento pokus o řešení problémů s *make* lze charakterizovat jako sporný, neboť se opět soustřeďuje na důsledky návrhu *make*, nikoliv na příčiny tkvící v návrhu samotném. Řešení problémů se vlastně jen odsouvá o úroveň dále.

2.2.2 *nmake*: větev *make* čtvrté generace

V polovině 80. let se od *make* odštěpil program *new-make*, označovaný také jako *make* čtvrté generace. Tato větev vývoje se snažila řešit řadu problémů původního *make*, jak rozebírá [9] [10]. Základní rysy *new-make* se podobají *make*, ale postupným vývojem se z něj vyvinul dosti odlišný program, který je dnes vyvíjen pod názvem *nmake*.[9]

nmake jako interpreter

Oproti *make* vystupuje *nmake* již víc jako interpreter makefile než pouhý preprocesor shellového skriptu v něm zabaleného. Je totiž možné definovat pravidla, která se chovají jako procedury či funkce, a ty pak vyvolávat. Navíc *nmake* dovoluje i definovat operátory pro pohodlné vyvolávání těchto uživatelských akcí. Některé konstrukce lze dokonce zapisovat v jazyce interpretovaném *nmake* bez pomoci shellu. Proměnné v *nmake* jsou dvou typů: řetězcové a celočíselné, přičemž řetězcové se užívají spíš jako seznamy tokenů než pouhý text k substituci, jak tomu je v *make*. Proměnné mohou být i lokální, což je pro definice procedurálních pravidel potřebné. Pro manipulaci s proměnnými nabízí *nmake* okolo 60 operátorů (tzv. edit operators).

Procedurálnější přístup *nmake* dovoluje lépe shrnout pravidla do „knihoven“ a typický makefile se tím nejen zjednoduší, nýbrž i stane lépe přenositelný, neboť sám pak nemusí obsahovat nic platformně závislého.

Uchovávání kontextu

Ke každému makefile existuje soubor nazývaný statefile, do nějž jsou ukládány informace o zpracovaných souborech a perzistentní proměnné z makefile, na jejichž obsahu mohou cíle také záviset. Oproti *make* uchovává *nmake* tedy aspoň část kontextu aktualizačních akcí, a vyhýbá se tak většině problémů, které přináší bezstavovost.

Implicitní závislostí a viewpathing

Na rozdíl od *make* disponuje *nmake* vestavěným scannerem programovatelným s využitím regulárních výrazů. Scanner lze naprogramovat pro vybrané typy souborů, aby v nich vyhledal implicitní prerekvizity, jimiž nemusí být jen soubor, ale dokonce i

[8] Např. programy *automake* a *autoconf* – viz http://www.gnu.org.
[9] Domovská stránka *nmake*: http://www.bell-labs.com/project/nmake/. Program *nmake*, o němž je tu pojednáváno, vyvíjí v současnosti firma Lucent a nesmí se zaměňovat se stejnojmenným programem *NMAKE*, který vyvíjí firma Microsoft a který také přikládá ke svým kompilátorům.

např. makro preprocesoru[10], které může být vztaženo k některé perzistentní proměnné makefile.

Při hledání implicitních závislostí se uplatňuje viewpathing, který je srovnatelný s viewpathingem, jímž je opatřen *GNU make*. Díky použití viewpathingu při hledání implicitních závislostí lze konstatovat, že viewpathing v *nmake* má větší opodstatnění a důležitější roli než v *make*.

Syntaxe makefile

Syntaxe makefile pro *nmake* vychází ze syntaxe použité původním *make*. V *nmake* je však ještě patrnější snaha do syntaktického schématu pravidel vtěsnat vše, bez ohledu na sémantiku. Schéma pravidel tu slouží jako „lepidlo“ pro tři druhy zápisu – akce pravidel mohou obsahovat buď shellové příkazy, příkazy *nmake*, nebo pravidla pro vestavěný scanner. Kromě pravidel může makefile obsahovat ještě direktivy *nmake* analogické k direktivám původního *make* a také dokonce i konstrukce preprocesoru jazyka C. [11]

Zápis makefile dále znepřehledňují neintuitivní názvy operátorů pro manipulaci s proměnnými, kryptická jména automatických proměnných a existence téměř stovky speciálních cílů (special atoms), na nichž stojí řada funkcí *nmake*.

Paralelní a distribuované zpracování

Stejně jako *make*, dovoluje i *nmake* paralelní aktualizaci více nezávislých cílů, navíc však poskytuje uživateli dobrou kontrolu paralelismu pomocí semaforů. Dále je možné nechat paralelní akce distribuovat na nevytížené uzly v homogenní počítačové síti. Na stanicích, které mají sloužit jako build servery, musí běžet speciální systémový démon, který se stará o distribuci příkazů a load balancing, aby některé uzly nebyly přetěžovány. Stanice musí být propojeny sdíleným souborovým systémem, který poskytuje všem stanicím potřebné soubory, a musí mít dobře synchronizovaný čas (neboť i *nmake* se opírá o časové údaje).

Efektivita *nmake*

Místo opakovaného spouštění nových instancí pro každý příkaz, jak to dělá *make*, používá *nmake* paralelně běžící instance shellu. Tento krok sice mění sémantiku makefile, ale zpravidla ulehčuje život uživateli. Dalším krokem ke zvýšení efektivity je kompilace makefile do binární formy, což by mělo mít význam zvlášť pro opakované zpracovávání standardních pravidel. O reálném přínosu této optimalizace lze však při dnešním výkonu počítačů pochybovat.

[10] Typickým příkladem takových maker jsou makra C/C++ preprocesoru, která se často používají právě pro řízení překladu zdrojových textů jazyků C a C++.

[11] Použití tohoto preprocesoru je zastaralým, dnes již nepoužívaným dědictvím po *new-make* a jeho používání není doporučeno; zůstává však v *nmake* kvůli zpětné kompatibilitě.

2.2.3 *cake*: *make* páté generace

Pátá generace *make*[12] [18], nazvaná *cake*, vznikla nedlouho po *new-make* (na konci 80. let), ale na rozdíl od *new-make* představuje slepou větev. Přesto obsahuje několik zajímavých nápadů.

Preprocesor

V případě *cake* je patrná snaha vzdálit se pojetí „*make* jako preprocesor“. Preprocesorové zpracování konfiguračního souboru pro *cake*, cakefile, je ponecháno na drobně upraveném preprocesoru jazyka C. Samotný *cake* se pak orientuje čistě na řešení závislostí mezi soubory. Použití samostatného preprocesoru zásadním způsobem změnilo fungování a syntaktické rysy původního makefile, ale také ztížilo případné další rozšiřování cakefile.

Metapravidla a sdílení pravidel a metapravidel

Novinkou, již *cake* přinesl, byla metapravidla podobná pattern rules, jak jsou nyní v *GNU make* – tehdy *make* pattern rules ještě neuměl. Oproti nynějším pattern rules, které pracují při unifikaci s jediným „žolíkovým“ symbolem, umožňuje *cake* použít až 11 těchto symbolů.

Sdílení a opakované využití pravidel a metapravidel plně zabezpečovala direktiva *#include* preprocesoru; *cake* nepoužíval žádná zabudovaná pravidla, jako to dělá např. *make* a *nmake*. I ta nejzákladnější univerzální pravidla bylo nutné explicitně definovat, např. vnořením standardních hlavičkových souborů pro *cake*.

Implicitní závislosti a dynamické rozhodování závislostí

Dalším zajímavým rysem *cake* je konstrukt, který dovoluje dynamicky doplnit prerekvizity cíle o implicitní prerekvizity již během stavby grafu závislostí. Využívá se k tomu externích programů, jejichž výstup je přidán mezi prerekvizity, jako kdyby jej uživatel explicitně zapsal do cakefile.

Dynamický charakter pravidel je v *cake* ještě posílen o podmíněná pravidla. Takové pravidlo se uplatní pouze tehdy, pokud je splněna dynamicky vyhodnocovaná podmínka, se kterou je spojeno. Podmínky mohou být kombinovány logickými spojkami, pro některé typické operace *cake* definuje vestavěná primitiva, ostatní případy lze řešit vyvoláním externího programu.

Dynamicky vyhodnocované implicitní závislosti a podmíněnost pravidel pozdější implementace *make* nepřevzaly, ale *GNU make* umožňuje dosáhnout podobného efektu díky restartu zpracování při změně některého vnořovaného souboru. Pátrání po závislostech je tedy možné i v *make* zahrnout do aktualizačního procesu pomocí několika triků [16], které však nejsou jednoduché a používají pomocné soubory s popisem závislostí extrahovaných ze zdrojových souborů. Výhodou řešení nabízeného *cake* je jeho uživatelská jednoduchost a přímočarost, díky čemuž se spíš vyhne chybám v aktualizacích. Nevýhodou pak je menší efektivita, která v případě rozsáhlých projektů může představovat problém.

[12] Kam se ztratila druhá a třetí generace *make*, není známo. V literatuře se zmiňuje pouze *new-make* jako *4th generation make* a *cake* jako *5th generation make*.

2.2.4 *Optimistic make*

Zajímavou variantou *make* je *optimistic make* [3]. Uživatel v případě klasického *make* po skončení editace a uložení změněných souborů vyvolá *make*, který načte konfiguraci z makefile a postará se o aktualizaci požadovaných cílů. Optimistický *make* pracuje naproti tomu na pozadí – kontroluje průběžně změny prerekvizit v grafu závislostí a aktualizuje automaticky cíle po změně prerekvizit, jakmile jsou tyto prerekvizity k dispozici.

Využita je tu plně tzv. forward chaining (nebo také data-driven či optimistic) strategie, která je protikladem ke dříve zmiňované backward chaining strategii používané klasickým *make*. Při forward chaining strategii se postupuje při aktualizacích od prerekvizit k cílům podle jejich připravenosti a bez ohledu na to, zda se výsledek jejich zpracování použije.

Přístup, kterým se ubírá optimistický *make*, je zaměřen hlavně na lepší využití výpočetní kapacity a na zkrácení latence aktualizace. Prvního požadavku se dosahuje paralelní či dokonce distribuovanou exekucí příkazů a také samotnou organizací práce – většina práce se vykoná na pozadí, zatímco vývojář přemýšlí a píše, což tolik výpočetní kapacity nespotřebuje; výpočetní zátěž se tedy lépe rozloží v čase. Důležitější však je naplnění druhého požadavku, tedy zkrácení latence. Vývojář nemusí po skončení editace typicky tak dlouho čekat na skončení aktualizace, protože ta se už provádí, nebo je dokonce hotová. Podle autorů *optimistic make* může být zkrácení doby, po kterou vývojář nečinně čeká, i několikanásobné.

Fungování optimistického *make*

Optimistický *make* se má chovat z pohledu uživatele přesně tak, jako *make* klasický, tedy uživatel dostane aktuální výsledky překladu až po „spuštění“ *make* (v opačném případě by průběžné změny narušily existující soubory).

Toto chování lze zabezpečit dodáním transakčních rysů do celého systému. Jedním z potřebných rysů je izolace probíhající aktualizace od okolního světa. Během automatické aktualizace na pozadí musí být změny v souborech a nově vytvořené soubory skryty. Až když uživatel zadá explicitní žádost o provedení aktualizace, mohou být všechny změny zviditelněny (podobně jako při transakčním commitu).

Zachování konzistence výsledků z hlediska aktuálních vstupů se zajistí sledováním vstupních souborů a závislostí jednotlivých cílů. V případě změny vstupů, dojde k abortu všech závislých výsledků, které čekají na své zviditelnění, a jejich aktualizace se restartuje.

Oproti klasickým transakcím může být použitý mechanismus volnější a dovolit „commit“ nekompletních výsledků, který nemusí být ani atomický – takový „commit“ představuje pak přechod k normálnímu výpočtu, který není nutné skrývat.

Výhody a nevýhody

Technickým problémem je tu zejména zajištění zmíněné izolace – to lze buď úpravami nástrojů, které mají být používány v izolaci, nebo úpravou jádra operačního systému, aby požadované služby poskytoval transparentně. Dále je třeba, aby souborový systém uměl posílat aplikaci oznámení o změně některého souboru, aby bylo možné monitorovat vstupy aktualizace. V případě *optimistic make* napomohl řešení speciální

souborový server, který jednak sloužil k požadovanému upozorňování na změny v souborech a jednak mohl zabezpečit jejich sdílení při distribuovaném výpočtu.

Cenou za zkrácení latence je zbytečné vykonání některých příkazů vydaných *optimistic make*, protože než jsou jejich výsledky dodány uživateli, musí se tyto příkazy vykonat znova v důsledku změn, které v tomto čase ještě nastanou; tedy zkrácení latence je na úkor minimalizace počtu nutných aktualizačních operací. Přesto autoři tvrdí, že optimistický přístup se vyplácí.

2.2.5 *amake*

Pro distribuovaný operační systém Amoeba vznikla koncem 80. let obdoba *make* nazvaná *amake* [2]. Záměrem tvůrců *amake* bylo využít potenciál Amoeby pro paralelní spouštění více úloh [1] a také napravit nedostatky, které pocházely z návrhu *make* – zejména nedostatky v deklaraci obecných pravidel a problém bezstavovosti.

Distribuované zpracování

Použitím systému Amoeba získal *amake* zadarmo podporu distribuovaného zpracování včetně load balancingu a transparentního sdílení souborů na distribuovaných file serverech.

Řešení bezstavovosti

Soubory na Amoebě jsou neměnné (immutable), jednou vytvořený soubor už nelze přepsat. Ke každému souboru, stejně jako k ostatním systémovým objektům v Amoebě, se přistupuje pomocí tzv. capabilities, asociaci mezi jménem souboru a capability, pomocí níž lze k souboru přistupovat, udržuje directory server. Pojmenování souboru v directory serveru není až tak podstatné, protože skutečný přístup k souborům je veden pomocí unikátních capabilities. Protože se nelze v distribuovaném prostředí spolehnout na čas poslední změny souboru, používá *amake* – podobně jako *nmake* – statefile, do nějž ukládá capabilities souborů spravovaných *amake* a který je také využit pro uložení proměnných amakefile. Pokud se změní proměnná spojená s některým souborem nebo capability některého souboru podle statefile nesouhlasí se záznamem v directory serveru, je postižený soubor neaktuální. Tímto je problém bezstavovosti elegantně vyřešen.

Aktualizační strategie

Pro aktualizaci souborů používá *amake* hlavně forward chaining strategii ve stylu „fire wall“: cíle jsou aktualizovány, jakmile jsou k dispozici jejich aktuální prerekvizity, a průchod grafem závislostí probíhá jen do té doby, dokud se nový výsledek liší od posledního zpracování. Při zápisu nového souboru (původní změnit nelze – soubory jsou na Amoebě neměnné) se tedy porovná nově zapisovaný výsledek s původním souborem. Je-li obsah souborů různý, je změněna vazba mezi capability souboru a jeho jménem u directory serveru, aby jméno odkazovalo na nový soubor. Tato strategie umožňuje vynechat některé aktualizační kroky, pokud se ukáže, že mezivýsledky se nezměnily, ačkoliv došlo ke změně vstupů.

Odvozovací pravidla

Odkazy na soubory uvedené v konfiguračním souboru amakefile jsou opatřeny atributy a *amake* dovoluje definovat pravidla pro odvozování dalších atributů z atributů již známých, např. lze definovat pravidlo, že soubor se sufixem *.c* bude mít atribut *TYPE=C-SOURCE*. Na základě odvozených atributů je dalšími pravidly určeno, které akce se vykonají. Pořadí aktualizačních akcí se pochopitelně řídí závislostmi tak, aby závislosti nebyly narušeny a aktualizace byla korektní.

Aktualizační akce se do amakefile nezapisují přímo jako v případě *make*. Přístup *amake* je deklarativní: zadají se soubory a případně jejich atributy a označí se vstupy, ale již se explicitně nezapisuje, co se má se vstupy dělat. Rozhodnutí o potřebných akcích se ponechává na pravidlech. Pravidla mohou pak tvořit opět „knihovny“ sdílené pomocí standardních „hlavičkových“ souborů, podobně jako v případě *cake*.

Aktualizační nástroje

Aktualizační nástroje, které *amake* má používat, se musí prvně nadefinovat, nepoužívají se přímo. To dovoluje *amake* přizpůsobit svým potřebám rozhraní k používaným nástrojům, a vypořádat se snadno např. i s problémem implicitních závislostí. Každý nástroj má za povinnost při svém použití zaznamenat seznam skutečně použitých vstupů (tedy vlastně všech souborových závislostí), které pak *amake* může uchovat ve statefile a brát jejich změny v úvahu při dalším spuštění.

2.2.6 *makepp*

Dalším blízkým příbuzným *make* je *makepp*[13], jehož lze chápat jako následníka *GNU make*. S *GNU make* je *makepp* téměř úplně zpětně kompatibilní, a tedy trpí i jeho nectnostmi. Na druhou stranu poskytuje některé zajímavé funkce, které nemá ani *make*, ani mnoho ostatních build managerů.

Způsob práce

Na rozdíl od *make* sestaví *makepp* prvně graf závislostí, podle nějž pak postupuje od prerekvizit k cílům (používá se tedy forward chaining strategie). Dále *makepp* dává do vztahu cíle a prerekvizity z makefile s reálnými soubory, rozumí zápisu adresářové struktury, a nezachází tedy s alespoň některými objekty v makefile jako s pouhým textem. Díky tomu dovede ztotožnit soubory odkazované různým způsobem, což usnadňuje zpracování i složitější adresářové struktury projektu. Standardně *makepp* nepoužívá rekurzivní spouštění sebe sama na podadresáře, nýbrž pracuje v jediné instanci a zařazuje makefile z podadresářů do zpracování hlavního makefile.

Implicitní závislosti

Do *makepp* je zabudován scanner souborů pro získávání implicitních prerekvizit.[14] Prohledání souborů se buď zadává explicitně v makefile, nebo je spouštěno automaticky na základě analýzy pravidla. Při analýze pravidla se bere v úvahu pouze jeho první

[13] Domovská stránka *makepp*: http://makepp.sourceforge.net/.
[14] Momentálně je podporován jen jazyk C, ale scanner lze snadno rozšiřovat.

příkaz, což je poněkud zvláštní chování, které v některých případech může působit problémy.

Procedurální prvky

Ačkoliv *makepp* nedefinuje žádný jazyk, který by sám přímo interpretoval, dovoluje do svého makefile vkládat kód v jazyce Perl. Vložený perlovský kód je pak prováděn interpreterem Perlu za běhu *makepp*. Tato schopnost dovoluje také snadno rozšiřovat *makepp* o další funkce.

Zpracování více variant vedle sebe

Pozornost zaslouží přímá podpora kompilace více variant výsledného programu, aniž by odvozené soubory vzniklé při překladu jedné varianty přepsaly odvozené soubory jiné dříve přeložené varianty. To šetří čas při paralelní práci na několika variantách projektu, např. při opravách chyb ve finální variantě, které jsou průběžně odlaďovány v ladicí variantě. Řada starších build managerů podobnou funkci překvapivě nijak přímo nepodporuje.

Signatury souborů

Specialitou *makepp* jsou signatury souborů, podle kterých se *makepp* rozhoduje o jejich neaktuálnosti. Místo času poslední modifikace souboru je možné použít jako signaturu souboru jeho hash. Hash může dokonce zahrnout pouze významné části zdrojového kódu a vynechat např. komentáře, na nichž generovaný kód nezávisí. Podobně jako v případě scannerů implicitních prerekvizit, může být *makepp* snadno rozšiřován o moduly počítající signatury jinými způsoby a algoritmy.

Nepříjemnou vlastností hashových signatur je jejich režie; dokonce i v případě minimální (či žádné) aktualizace je třeba signatury přepočítat. Dále je nutné počítat s faktem, že změna v komentáři sice nezmění generovaný kód, ale může vést ke změně údajů o řádkování v ladicích informacích. Přeskočení kompilace souboru, ve kterém byl změněn pouze komentář, může tedy zmást debugger a následně i programátora.

Řešení bezstavovosti

Použití signatur vyžaduje už statefile. Kromě signatur si *makepp* uchovává i seznamy prerekvizit (včetně implicitních nalezených scannerem), informaci o architektuře, na níž překlad proběhl a skutečně použité příkazy, včetně jména adresáře, ve kterém byla aktualizační akce spuštěna. Takto komplexní informace už představuje velice spolehlivé zachycení stavu, které řeší dostatečně problém bezstavovosti.

2.2.7 *Odin*

Pozoruhodným konkurentem *make* byl svého času build manager *Odin*[15], který se bohužel příliš neujal, podobně jako většina náhrad *make*, ačkoliv disponuje řadou

[15] *Odin* byl vyvinut na University of Colorado; distribuci i manuál lze nalézt ve FTP archívu univerzity na adrese ftp://ftp.cs.colorado.edu/pub/distribs/odin/.

zajímavých funkcí a *make* překonává ve většině ohledů – snad s výjimkou snadnosti svého použití začátečníky.

Syntaxe konfigurace

Konfigurační soubor *Odinu*, odinfile, má dobře definovaná syntaktická a lexikální pravidla, jeho objekty jsou typované a rozdělené do tří základních kategorií: soubor, řetězec a seznam objektů.

Syntaxe, již *Odin* používá pro zápis konfigurace, je založena na funkcionálním přístupu – ostatně pravidla v odinfile se nazývají výrazy (odin expressions) a *Odin* řídí svou práci jejich vyhodnocováním. Výrazy obsahují parametry a aplikace operátorů, příkazy se v nich přímo neuvádějí. Přímé použití nástrojů a příkazů stejně není dost dobře možné, neboť použité nástroje jsou od výrazů v odinfile důsledně odstíněné.

Databáze nástrojů a jejich izolované spouštění

Má-li být nějaký nástroj použit *Odinem*, musí být k němu dodán popis, který specifikuje jeho chování, vstupy, výstupy, použité proměnné prostředí apod. Od nástroje se předpokládá jistý protokol chování, který může vynutit použití skriptů, jež nástroj zabalí a jeho chování upraví podle potřeby. Nástroje jsou spouštěny v adresáři, který pro ně *Odin* připraví a ve kterém jsou očekávány jejich výstupy, včetně souborů s chybovými hláškami. Definice nástrojů dovoluje zahrnout i hledání implicitních závislostí.

Spouštění nástrojů v izolovaných lokacích řeší hned několik problémů: není problém se zápisovými právy a cestami pro uložení výsledků vyprodukovaných nástrojem, *Odin* má přehled o vytvořených souborech a může s nimi dále operovat (viz níže) a navíc je snadné nechat nástroje pracovat paralelně či dokonce distribuovaně, aniž by se navzájem ovlivňovaly, dokonce i jejich chybový výstup lze snadno separovat a uchovat pro pozdější použití.

Odinova cache

Soubory vzniklé spuštěním nástrojů přemísťuje *Odin* do cache, nedává je uživateli přímo k dispozici. Cache společně s daty drží i kontext jejich vzniku (čas vzniku, použité proměnné prostředí a přepínače nástroje, seznam implicitních závislostí apod.) a dovoluje skladovat více variant překladu jednoho souboru (např. jeho ladicí a finální verzi). Je pak možné používat soubory z cache podle momentální potřeby.

Výhodou tohoto přístupu je, že se uživatel nemusí starat o skladování více variant překladu, není potřeba úprava souborového systému, aby poskytoval nějaké další služby, a dovede i šetřit některé kroky překladu, podobně jako to dělá *amake*. Neboť cache má přímo k dispozici údaje o stavu svého obsahu, odpadá komplikované zjišťování stavu odvozených souborů při zjišťování závislostí – snad všechny ostatní build managery musí dřív nebo později zkontrolovat závislosti souborů jejich přečtením nebo aspoň zjištěním času jejich modifikace.

Možnou nevýhodou použití cache je relokace vzniklých souborů. Pokud nějaká skupina souborů obsahuje odkazy mezi sebou, mohou se tyto odkazy při přemístění do cache rozpadnout. To se může projevit např. při ladění výsledného programu, jsou-li ladicí informace umístěny v externí databázi a nikoliv vloženy přímo do spustitelného souboru.

Interaktivní režim

Odin nefunguje jen jako jednorázově spouštěný build manager, ale může pracovat i v interaktivním režimu, kdy jsou mu uživatelem zadávány výrazy k okamžitému vyhodnocení. Tyto výrazy mohou podávat i zpětně informaci o dříve proběhlých akcích a jejich výsledcích, které si *Odin* pamatuje díky své cachi.

Poznámky k fungování

Odin se řídí forward chaining strategií podobně jako *amake*. Této strategii napomáhá cache, která spojuje statefile a file server se schopností uchovávat více verzí souboru. To vše, mimo jiné, umožňuje *Odinovi* dokonce snadno překládat projekt přímo z repositáře bez uživatelovy vlastní lokální kopie souborů obsažených v repositáři.

Odin navíc umí během aktualizace sledovat změny spravovaných souborů a aktualizaci v případě potřeby restartovat. Je dokonce možné, aby graf závislostí byl cyklický a cíl aktualizoval sám sebe, dokud aktualizace nedosáhne pevného bodu.[16]

2.2.8 *Jam*

Jedním ze současných konkurentů *make* je *Jam*[17], který se vyvíjí také poměrně dlouhou dobu a své kvality již prakticky prokázal i v případě velkých projektů [19].

Jam byl od svého vzniku zamýšlen jako úplná náhrada *make* a jeho autoři věnovali evidentně dost pozornosti analýze problémů *make* – svůj program založili tedy na odlišných principech a jejich návrh se snaží být přímočarý a jednoduchý.

Zvolený přístup lze asi nejlépe charakterizovat jako procedurální, se snahou oddělit od sebe platformně nezávislou část procesu, která zahrnuje ustanovení závislostí a výběr akcí, od platformně závislé, již tvoří vlastní aktualizační příkazy.

Syntaxe konfigurace

Zápis konfiguračního souboru se řídí jednoduchou a intuitivní gramatikou. Nepoužívá se několik různých druhů syntaxe ani žádný preprocessing či podobné textové manipulace. Konfigurační soubory jsou tedy snadno čitelné a bez zvláštních záludností.

Podobně jako jiné build managery také *Jam* používá pravidla; jeho pravidla jsou však rozdělena na dvě části: na platformně nezávislou (označovanou jako rule) a na příkazovou (označovanou jako action).

Rules vystupují jako procedury (resp. funkce), které mohou mít až 9 argumentů a které jsou psány v poměrně silném jazyce připomínajícím jazyk C. V rules lze definovat lokální proměnné a manipulovat s lokálními i globálními proměnnými pomocí rozumně malé sady jednoduchých operátorů a vestavěných příkazů. K dispozici jsou cykly, podmíněné větvení a další podobné konstrukce. Existence příkazů pro podmíněné větvení je jedním z důvodů, proč *Jam* nepotřebuje preprocesor.

Actions obsahují příkazy shellu, v nichž mohou být použity globální proměnné a parametry předané action, které představují prerekvizity a cíle. Pro action nad

[16] Užitečnost této vlastnosti je trochu sporná, ale může se hodit vývojářům překladačů pro automatický bootstrapping jejich překladače.
[17] Domovská stránka *Jamu*: http://www.perforce.com/jam/jam.html.

libovolným cílem je možné nadefinovat specifické hodnoty globálních proměnných, původní hodnota proměnné je pak po čas provádění action překryta hodnotou jinou. To dovoluje příkazy shellu uvnitř action parametrizovat hodnotami pro každý cíl jinými.

Proměnné fungují jako seznamy (resp. pole) řetězců; oproti *make* a jeho následníkům je třeba zdůraznit podstatný rozdíl: proměnné *Jamu* se typově skutečně chovají jako seznam (s notně rozšířenými možnostmi), zatímco proměnné *make* jsou pouze textové a jejich případná interpretace coby seznamu je dost volná a závislá na jejich obsahu (zejména tedy na přítomnosti různých oddělovačů).

Sdílení pravidel a jejich užívání

Jam pochopitelně umožňuje pravidla sdílet, takže uživatel typicky nepotřebuje žádná pravidla definovat a běžně si vystačí s předdefinovanými pravidly. Za zmínku stojí „inverzní“ způsob sdílení pravidel použitý *Jamem*. Obvykle soubor s použitím pravidel zanořuje soubory s jejich definicemi. V případě *Jamu* je to naopak: při spuštění se hledá soubor Jambase, kde by se měla pravidla definovat – pokud není Jambase nalezen v aktuálním adresáři, užije se jeho verze dodaná v instalaci *Jamu*. Jambase vnořuje z aktuálního adresáře Jamfile, což je soubor určený pro uživatele právě k tomu, aby do něj zapsal aplikace pravidel na soubory svého projektu. Jamfile se případně stará o vnoření dalších souborů, třeba Jamfile z podadresářů (to je však již v režii uživatele). Díky tomuto „inverznímu“ sdílení pravidel se uživatel nemusí starat o vyhledávací cesty k vnořovaným souborům, ani o vnořování potřebných souborů s pravidly z distribuce *Jamu*. Snadno se také vyhne konfliktu s nimi, potřebuje-li jejich modifikaci – stačí k projektu přiložit vlastní verzi Jambase.

Použití předdefinovaných pravidel, i když vlastně představuje volání procedury, se nezasvěcenému uživateli jeví jako deklarace, jíž se jen sděluje, jaké jsou vstupy a jaký má být výstup celého procesu. Nikde není přímo viditelné, jak se výsledku dosáhne. Pravidla určená k použití běžným uživatelem bývají naprogramována tak, aby si většinu potřebných informací a parametrů ze zadaných vstupů sama odvodila a vybrala akce, které je nutné vykonat.

Jak *Jam* pracuje?

Jam funguje trochu jinak než dosud zmíněné build managery. Jeho práce probíhá ve třech fázích: v první fázi načte konfigurační soubory a interpretuje rules, ve druhé fázi vyhledá soubory, se kterými se bude operovat, a ustanoví vazbu mezi výsledkem zpracování rules a soubory (tento proces se nazývá binding) a ve třetí fázi provede potřebné actions.

Nutné je poznamenat, že proměnné jsou po interpretaci rules zmrazeny (tedy jejich hodnota je trvalá ještě před vykonáním jakéhokoliv příkazu), takže do nich nelze např. uložit výsledek nějakého externího programu. Dále *Jam* omezuje hromadnou specifikaci vstupů pravidel pomocí žolíků (wildcards), jak to umožňuje *make*. Toto chování je zdůvodněno tak, že podobné triky jsou závislé na momentálním obsahu adresářů a nepředstavují robustní řešení.

Jam také nepodporuje žádným způsobem své rekurzivní spouštění, ani to nebývá potřeba, neboť práce s adresářovou strukturou je na dosti vysoké úrovni.

Vyhledávání souborů

Speciálními proměnnými se nastavuje jednak adresář, vzhledem k němuž jsou cílové soubory zadávány (zadání cílů se tedy nemusí vztahovat jen k aktuálnímu adresáři), a jednak vyhledávací cesty pro soubory. Protože (i tyto speciální) proměnné lze v rules nastavovat dokonce zvlášť pro každý cíl, je tento mechanismus daleko jemnější než viewpathing v libovolném předchozím build manageru.

Implicitní závislosti

Jam má zabudovaný scanner souborů řízený regulárními výrazy. Implicitní prerekvizity se vyhledávají s pomocí tohoto scanneru, rules (která si uživatel může vytvořit dle svých potřeb) a zabudovaného příkazu, který doplní prerekvizity cíle o prerekvizity zjištěné dynamicky.

Regulární výrazy nedovedou vyřešit některé speciality (jako např. podmíněné vnořování souborů), a tak mohou přinést nadbytečné závislosti. Jiný způsob dodání implicitních prerekvizit však není k dispozici, např. použití externího programu nepřipadá v úvahu kvůli zmíněnému postupu práce *Jamu*.

Nevýhody

Zřejmě největší nevýhodou *Jamu* je jeho bezstavovost, kterou nijak neřeší – stejně jako *make* používá pouze čas modifikace souboru. Nejsou ukládány ani proměnné prostředí, které *Jam* importuje jako proměnné, a které tedy mohou mít přímý vliv na aktualizaci. Neukládání implicitních závislostí také nešetří režii potřebnou na jejich opakované hledání.

Podpora paralelní exekuce akcí je omezená. *Jam* sice dovoluje paralelní vykonávání akcí, ale neposkytuje žádná primitiva, kterými by nebezpečné příkazy bylo možné synchronizovat.

2.2.9 *Boost Jam* a *Boost Build*

Nástroj *Boost Jam* vychází z *Jamu* a dále jej rozvíjí. Základní principy *Jamu* zůstávají – uživatel pomocí procedur deklaruje vstupy projektů a jejich vlastnosti, zpracování konfigurace probíhá stále ve třech fázích. *Boost Jam* se též stále potýká s největším nedostatkem svého předchůdce: stále nepodporuje ukládání stavu. Rozšířen je však významně jazyk pro zápis konfigurace. Nad *Boost Jamem* je vybudován systém *Boost Build*[18], který nové rysy jazyka plně využívá, aby běžnému uživateli poskytnul např. podporu variant překladu nebo vylepšenou přenositelnost konfiguračních souborů.

Varianty překladu

Boost Build definuje tzv. feature jako normalizovaný aspekt konfigurace nezávislý na konkrétním aktualizačním nástroji; příkladem může být feature, která říká, zda je povolen inlining[19]. Každá feature pak může nabývat hodnot, které definují vlastnosti brané v úvahu při aktualizaci. Z definice feature je zřejmé, že features zvyšují

[18] Domovská stránka Boost Build V2: http://www.boost.org/tools/build/v2/.
[19] Inliningem je myšleno rozvíjení těl funkcí, typické zvlášť pro překladače jazyka C++.

přenositelnost konfiguračních souborů. Nastavení hodnot features pak dovoluje produkovat různé varianty produktu – pravidla, která se starají o aktualizaci, se jen zařídí podle těchto hodnot a zavolají aktualizační nástroj s parametry, které zadaným hodnotám odpovídají. Výhodou je, že varianty lze snadno od sebe odvozovat. Systém *Boost Build* věnuje variantám jako jeden z mála build managerů náležitou pozornost.

Změny v jazyce

Jazyk pro implementaci pravidel je v *Boost Jamu* značně posílen, neboť došlo k jeho rozšíření o třídy, lze v něm tedy programovat i velmi komplikované objekty. Ostatně i zmiňované features a podpora variant jsou implementovány v tomto jazyce, nikoliv build managerem samotným. Kromě řady drobných specializovaných rozšíření stojí za zmínku např. omezené typování, nepřímé volání funkcí a semafory pro lepší kontrolu paralelismu.

„Typování“ *Boost Jamu* dovoluje v hlavičkách funkcí stanovit užší možnosti unifikace parametrů s jejich hodnotami předanými při zavolání funkce. Parametry uvnitř funkcí lze chápat jako proměnné s typy odpovídajícími jediné hodnotě, neprázdné sekvenci hodnot nebo obecné (i prázdné) sekvenci. Nepřímé volání funkcí umožňuje dokonce takové konstrukce jako funktory s částečně vyplněnými parametry, které lze předávat jako predikáty dalším funkcím.

Posílení skriptovacího jazyka umožňuje snadno rozšiřovat možnosti *Boost Jamu* a dovoluje mu vypořádat se s řadou problémů, které *Jam* rozumně řešit nedovedl (např. zmíněné features by jazyk *Jamu* zřejmě nedovedl zajistit). Cenou za tuto obecnost je ztráta přímočaré jednoduchosti, která je *Jamu* vlastní.

2.2.10 *Ant*

Ne zrovna typickým build managerem je *Ant*[20], který není koncipován jako náhrada *make*, nýbrž jako build manager speciálně pro vývojáře pracující v jazyce Java[21], v němž je také sám implementován.

Základní principy

Ant není klon *make* a ani jako *make* nefunguje. Úkolem *Antu* není zjišťovat závislosti mezi soubory a spouštět aktualizační nástroje, nýbrž pouze vykonat uživatelem zadané operace, mezi nimiž mohou být závislosti, ze kterých vyplývá pořadí, ve kterém tyto operace musí být vykonány.

Operací může také být přeložení nějakého podprojektu, ale operace samotná se omezuje na spuštění překladače. Zjištění závislostí mezi soubory a aktualizaci již ponechává na překladači samotném. Tento přístup pramení i z fungování překladačů Javy. Mezi typické operace, které *Ant* provádí, patří např. aktualizace zdrojových textů z repositáře, vyvolání překladu, vytvoření instalačního balíku, smazání meziproduktů kompilace apod.

[20] Domovská stránka *Antu*: http://ant.apache.org/.
[21] Existuje i klon *Antu* pro platformu .NET nazvaný *NAnt*, který tu samostatně rozebírán nebude; *NAnt* lze najít na adrese http://nant.sourceforge.net/.

Ant lze v současné době zapojit do většiny integrovaných vývojových prostředí pro Javu a svým celkovým pojetím se vzdaluje od tool-based přístupu, jak je patrné i z následujícího popisu.

Tasks

Místo shellových příkazů či využívání externích programů (i když i to je stále ještě možné) používá *Ant* tzv. úlohy (tasks). Shellové příkazy a externí programy jsou problémem pro přenositelnost, což je v Javě, pro niž je *Ant* určen, nemyslitelné.

Task představuje kompletní náhradu za externí program, která je implementována v podobě javovských tříd; *Ant* tedy může být snadno dynamicky rozšířen o nový task. Standardní sada tasků pokrývá typické příkazy, které se používají i v makefile, zejména manipulaci se soubory, adresáři a archívy, ale zahrnuje i řadu dalších okruhů včetně správy několika rozšířených typů repositářů, jako např. *CVS*, *Perforce* či *ClearCase*. To dovoluje snadno vytvářet skripty, které se postarají o kompletní správu produktu – od aktualizace zdrojového textu z repositáře, překlad, až po vytvoření instalačního balíčku či jeho deployment.

Syntaxe konfigurace

Konfigurační skript *Antu*, buildfile, se zapisuje ve formě XML dokumentu. To vymezuje rámec syntaxe skriptu, šetří uživatele syntaktických podivností, kterými trpí ostatní build managery, a vynucuje strukturování skriptu.

Nevýhodou použití XML je jeho rozvláčnost, která ruční zápis notně znepříjemňuje; na druhou stranu XML je formát, který je možné validovat, snadno automaticky zpracovávat a také snadno generovat. Pro větší projekty se ani nepočítá s tím, že by se buildfile tvořil celý ručně. I v tomto směru se *Ant* vymyká přístupu typickému pro jiné build managery.

2.3 Ostatní build managery

Z velkého množství existujících build managerů si, kromě těch, které byly stručně představeny v předchozím textu, zaslouží aspoň zmínku i některé další. Jednou z pozdějších alternativ k *make*, která nabízela daleko víc než *make*, byl *BiiN SMS* [17]. Zajímavým je bezesporu experiment nazvaný *automatic make* [11], který lze charakterizovat jako „*make* bez makefile“. Program *shape* [13] [14] [15] se pokusil spojit *make* s verzováním souborů pomocí atributového souborového systému nebo databáze, a zastřešit tak funkci build manageru i repositáře. Při hledání deklarativního přístupu popisu konfigurací se experimentovalo třeba i s rozšiřováním *make* o relační algebru [12].

I když je tato práce soustředěná na build managery, je vhodné na tomto místě uvést také výčet některých dalších významných SCM nástrojů a systémů, soustředěných převážně na ostatní úkoly SCM – do tohoto výčtu patří již klasické repositáře *SCCS*, *RCS* a *CVS*, rozsáhlejší systémy jako *Adele*, *Cedar*, *DSEE* nebo *Marvel* a z novějších např. *AllChange*, *ClearCase* nebo *Perforce*.

2.4 Shrnutí požadavků

Následující pasáž shrnuje požadavky na build manager a pozorování vyplývající z předcházejícího výčtu vybraných build managerů a stručného představení některých jejich významných rysů. Uvedené požadavky představují základ pro návrh a implementaci build manageru, což je hlavním cílem této práce.

Přenositelnost

Problém přenositelnosti mezi platformami se týká hlavně přenositelnosti popisu konfigurace, přenositelnost samotného build manageru je spíš druhořadá záležitost. Aby popis konfigurace uživatelova projektu byl přenositelný, musí splňovat následující podmínky.

- Nesmí obsahovat žádné příkazy, neboť ty bývají silně platformně závislé.
- Musí být odstíněn od použitých nástrojů. Nesmí je přímo používat (vyplývá též z předchozího), ale ani nesmí spoléhat na jejich specifické vlastnosti.
- Musí být umožněn jednotný způsob zápisu platformně závislých věcí, např. souborových cest.

Z těchto podmínek je zřejmé, že přímočarý přístup používaný *make* a jeho následníky je zcela nepoužitelný, protože jejich konfigurační soubory jsou založeny na přímém vyvolávání příkazů. Aby byla přenositelnost konfigurace umožněna, je nutné v první řadě veškeré příkazy a nástroje odstínit od jejího popisu, se kterým manipuluje běžný uživatel. Výše uvedené a popsané build managery se snaží s problémem přenositelnosti vypořádat několika způsoby.

Prvním možným řešením je jednotlivé nástroje popsat v nastaveních build manageru. Nevýhodou je potřeba dalšího jazyka nebo nástroje, kterým se nakonfiguruje samotný build manager. Výhodou tohoto odděleného popisu může být zjednodušení jazyka, kterým se popisuje vlastní konfigurace a se kterým se dostane do styku běžný uživatel.

Druhým řešením je po nástrojích vyžadovat určitý protokol chování, který lze zaručit např. pomocnými skripty shellu vytvořenými na míru konkrétním nástrojům. Problematickým se může pak stát přenos na platformy, kde je shell velmi omezený, a přizpůsobení nástrojů tedy komplikovanější. Toto řešení bývá kombinováno např. s řešením prvním, aby se veškerá zátěž při adaptaci nástroje nepřesunula na pomocné skripty.

Třetím řešením je poskytnout pro popis konfigurace dostatečně silný jazyk, který umožní nástroje a příkazy od vlastního popisu konfigurace odstínit jejich přesunutím do znovupoužitelných „knihoven“ s abstraktním, platformně nezávislým rozhraním. Běžný uživatel pak používá pouze tyto knihovny. Nevýhodou je zejména nutnost sdílení zmíněných knihoven a potřeba silnějšího jazyka, který dovede popsat konfiguraci i použité nástroje. Na druhou stranu, výhodu oproti prvnímu řešení představuje existence pouze jediného jazyka, což mimo jiné dovoluje snadněji vytvořit ad hoc modifikace předdefinovaného chování nástrojů, které nemusí vždy plně vyhovovat potřebám uživatele.

Každá z těchto možností vyžaduje pro konkrétní nástroj vytvořit na míru šitou adaptační vrstvu. Tímto problémem netrpí snad jen *automatic make* [11], který jej zcela obchází, neboť místo nástroje adaptuje chování uživatele.

Univerzální pravidla

Univerzální pravidla představují obecný předpis, jak se má vykonat nějaký typický a často opakovaný úkon. Jejich přínosem je zmenšení popisu konfigurace, zvýšení jeho přehlednosti a snížení objemu redundantních informací. Podpora nějaké formy univerzálních pravidel je pro praktickou použitelnost nutná. Stejně tak je nutné, aby bylo možné tato pravidla rozšiřovat, doplňovat a modifikovat.

Tyto požadavky na univerzální pravidla vylučují jejich pevné zabudování do build manageru. Nabízí se přirozené řešení, učinit jazyk pro popis konfigurace natolik silným, aby v něm mohla být snadno definována i tato univerzální pravidla, nikoliv aby sloužil jen k používání předdefinovaných pravidel.

Použití uživatelsky definovatelných univerzálních pravidel přináší problém jejich sdílení. Použití preprocesoru jako (mimo jiné) prostředku pro jejich sdílení se ukázalo být nevýhodné (viz *cake*). Různé importní direktivy, které se chovají podobným způsobem jako preprocesor, trpí problémy s nastavením cest, kde mají být soubory s pravidly hledány (tento problém byl dlouho řešen v *make* a jeho následnících). Rozdělování pravidel do skupin (např. předdefinovaná pravidla, sdílená uživatelská pravidla a explicitní pravidla) s různými prioritami jejich uplatnění je komplikované a může uživatele mást. Jako elegantní řešení se jeví „inverzní“ sdílení používané *Jamem*.

Syntaxe popisu konfigurace

Pro přehlednost popisu konfigurace je potřeba, aby zápis konfigurace mohl být úsporný, ale přesto se vyhýbal zkratkovým symbolům, jejichž význam nelze na první pohled odhadnout. Použitý jazyk by měl mít dobře definovaná a jednoduchá lexikální pravidla; jazyk založený na textových manipulacích s velkou pravděpodobností nebude splňovat požadavek srozumitelného zápisu a zejména hrozí potíže v interpretaci lexikálních pravidel.

Syntaxe jazyka by se měla řídit jednotnou logikou, není vhodné míchat více přístupů (např. tedy zavlékat do jazyka pro popis konfigurace ještě preprocesor). Na druhou stranu nesmí dojít k tomu, aby jedno syntaktické schéma bylo použito pro sémanticky odlišné věci.

Robustnost aktualizačního procesu

Jedním ze základních požadavků na build manager, jak bylo zmíněno v minulé kapitole, je jednak korektnost provedené aktualizace a jednak vykonání aktualizace vedoucí k výsledku, který splňuje kritéria zadaná konfigurací.

Prvním předpokladem pro tento cíl je správné a jednoznačné zadání konfigurace, zejména tedy vstupů celého procesu. Některé build managery dovolují hromadná zadávání vstupních souborů např. pomocí souborové masky. To je sice pohodlné, ale závislé na stavu vývojářových pracovních adresářů. Takové zadání může do výsledku přidat i vstupy, které v něm podle zadané konfigurace být vůbec nemají, nebo naopak se může stát, že některý vstup pro zadanou konfiguraci bude chybět, aniž by na tuto

skutečnost byl vývojář upozorněn. Chybějící možnost hromadného zadání vstupů lze tedy chápat jako záměrný pozitivní rys návrhu, nikoliv jako opomenutí.

Poněkud sporný je význam viewpathingu. V případě, že je tento mechanismus používán pro vyhledávání implicitních závislostí a kopíruje postup vyhledání těchto závislostí aktualizačními nástroji, je to užitečný nástroj. Jeho používání jako náhrady repositáře však může přinášet spíš problémy a ohrožovat robustnost aktualizačního procesu, neboť může opět dojít k tomu, že zpracovány budou jiné vstupy, než které vývojář skutečně chtěl nechat zpracovat.

Problémem je již mnohokrát zmíněný způsob rozhodování o aktuálnosti cílů, který je závislý na kontextu, v němž aktualizace probíhala nebo má probíhat. Pro řešení tohoto problému je nutné uchovávání aspoň nejdůležitější části kontextu, které mají na aktuálnost přímý vliv.

Užitečnou a důležitou vlastností je dohledání implicitních závislostí. Naneštěstí tvar implicitních závislostí je specifický pro různé druhy vstupů. Častým řešením v novějších build managerech je použití scanneru na bázi regulárních výrazů, které je schopné se s typickými případy implicitních závislostí vyrovnat. Alternativním řešením je použití externího nástroje, který seznam závislostí po zadaný vstup dodá; tato alternativa je obecnější než použití vestavěného scanneru, ale také méně efektivní. Nepříjemným vedlejším efektem vyhledávání implicitních závislostí je jeho vysoká cena. Vhodné je tedy použít nějaký mechanismus, který potřebu opakovaného prohledávání omezí. Nutnost explicitního vyžádání aktualizace informací o implicitních závislostech, které bývá často užíváno např. v *make*, však hrozí opomenutím vývojáře a následnými nekorektními výsledky, je tedy nevhodná.

Výkonnost

K vyššímu výkonu aktualizačního procesu může dopomoci paralelní nebo dokonce distribuovaný výpočet. Distribuovaný výpočet je zvlášť bez podpory distribuovaného operačního systému velmi komplikovaná záležitost. Vedle technických problémů se sdílením dat a dílčích výsledků aktualizace tu nastává i potíž se správou kontextu prostředí (např. verzemi softwarového vybavení jednotlivých uzlů sítě). Tyto potíže mohou také ohrozit robustnost aktualizačního procesu: pokud totiž není zaručena shodnost kontextu na všech uzlech, kde probíhá distribuovaná aktualizace, nemusí být výsledek celého procesu korektní vzhledem k zadané konfiguraci.

Na celkovém výkonu aktualizace se podepisuje také režie build manageru. Nejdražší operací bývá vyhledání implicitních závislostí. Cena za zjištění času poslední modifikace souborů, i když není také úplně zanedbatelná, bývá často mylně příliš nadnesená [16]. Cena za zpracování konfiguračních skriptů naopak téměř zanedbatelná bývá (alespoň pokud se jejich syntaxe i uživatel řídí rozumnými pravidly a zásadami).

Je třeba dávat pozor na extenzivní vytváření podprocesů, které s sebou nese další režii. V této souvislosti je vhodné zmínit i rekurzivní spouštění build manageru, které může zvyšovat režii nejen vytvářením nových procesů, ale i opakovaným zjišťováním stavu aktualizace. Rizikem je v tomto případě také narušení korektnosti aktualizace, protože rekurzivně spuštěný podproces nemusí mít k dispozici všechny potřebné informace. Oběma těmito problémy v případě *make* se zabývá opět [16].

3. Návrh řešení

Návrh *Experimentálního Build Manageru*, zkráceně *EBM*, jehož implementace je cílem této práce, čerpá inspiraci zejména z fungování *amake* a *Jamu*, jejichž kvality byly zmíněny v předchozí kapitole. Cílem návrhu je vytvořit build manager, který poskytne:

- robustnější aktualizační proces díky ukládání klíčové části kontextu do statefile,
- možnost zpracování implicitních závislostí vestavěnými i externími prostředky,
- podporu variant,
- paralelizaci aktualizačního procesu.

To vše je nutné zvládnout s přihlédnutím ke snadné přenositelnosti jak vlastního programu, tak i popisu konfigurace, jejíž zápis by měl zůstat dostatečně jednoduchý. Požadavek jednoduchosti se vztahuje též na adaptaci aktualizačních nástrojů.

Tato kapitola se bude věnovat v první části principům, na kterých je EBM založen, a v druhé části důležitým rysům implementace. Detaily implementace a její použití tu rozebírány nebudou. Pro lepší představu, jak pravidla vypadají a používají se, může čtenář nahlédnout do *Přílohy A*, která je věnována demonstračním ukázkám, nebo také do uživatelské příručky na přiloženém disku CD-ROM.

3.1 Základní principy fungování

3.1.1 Graf závislostí

Jádrem všech build managerů a úlohou, na niž se problém aktualizace cílů redukuje, je sestavení grafu závislostí a jeho průchod spojený s vyvoláním aktualizačních operací. Od vlastností grafu a způsobu jeho zpracování se tedy odvíjí zbytek návrhu.

Vrchol v grafu závislostí je v *EBM* nazýván cíl (target). Cíl může a nemusí být spojen se souborem; pokud cíl soubor nereprezentuje, označuje se jako virtuální. Vytváření i zpracování grafu v *EBM* je inspirováno použitím atributů cílů, které bylo představeno u *amake*. Každý cíl v *EBM* nese sadu hodnot atributů, která může být uživatelsky rozšiřována; množina atributů tedy není pevná. Atributy mají význam pro nakládání s cílem, zejména pak pro aktualizační akci, kterou může mít přiřazenou[22] každý (i virtuální) cíl. Některé atributy jsou předdefinovány samotným programem a

[22] Aktualizační akce podobně jako jiné procedury, které mají operovat nad konkrétním cílem, se s cílem asociují opět pomocí atributů, např. pro aktualizační akci je tímto atributem předdefinovaný atribut ***Action***.

ovlivňují algoritmus zpracování grafu pevně předepsaným způsobem. Bezprostředně s podobou grafu souvisejí atributy ***Inputs***, ***Outputs*** a ***WaitFor***.

- ***Inputs*** obsahuje seznam cílů, které tvoří vstupy aktualizační akce spojené s daným cílem. Na svých vstupech je cíl automaticky také závislý.
- ***Outputs*** je atribut analogický k atributu ***Inputs***; obsahuje seznam cílů, které představují výstupy aktualizační akce a které jsou tedy na daném cíli závislé. Cíl samotný je brán jako potenciální vstup i výstup aktualizační akce, ať již je v atributech ***Inputs*** a ***Outputs*** přidán, či nikoliv.
- ***WaitFor*** obsahuje seznam cílů, které mají být aktualizovány před daným cílem, ale nepředstavují vstup jeho akce. Atribut ***WaitFor*** tedy funguje jako vestavěné synchronizační primitivum. Oddělení vstupní a synchronizační závislosti je pro akce důležité, neboť pro ně jsou v první řadě podstatné vstupy a výstupy, nikoliv další závislosti mezi cíli.

Z tohoto popisu atributů, podle nichž se graf formuje, lze nahlédnout, že se v něm vyskytují dva druhy závislostí – první, reprezentovaná atributem ***WaitFor***, představuje časovou závislost akcí, druhá, reprezentovaná atributy ***Inputs*** a ***Outputs***, představuje tok dat mezi akcemi.[23]

Závislost je v grafu reprezentovaná orientovanými hranami. Nechť *A* označuje vrchol grafu různý od vrcholu *B*: jestliže ***Outputs*** vrcholu *A* obsahuje *B* nebo ***Inputs*** vrcholu *B* obsahuje *A*, vede v grafu od *A* do *B* vstupně-výstupní hrana (input/output edge); jestliže ***WaitFor*** vrcholu *B* obsahuje *A*, vede v grafu od *A* do *B* synchronizační hrana (wait-for edge). Smyčky (hrany vedoucí z vrcholu do něj samotného) nebudou do grafu přidány.

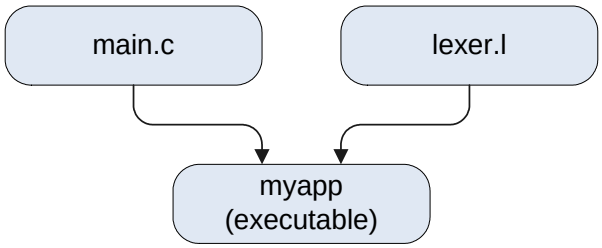
Z hlediska chování grafu je synchronizační hrana slabší než vstupně-výstupní: pokud existuje vstupně-výstupní hrana, není již potřeba paralelní synchronizační hrana, neboť časová závislost, kterou synchronizační hrana představuje, je indukována závislostí toku dat. Místo udržování obyčejného grafu s hranami dvou barev je při implementaci možné též použít multigraf – navazujícímu zpracování grafu tento fakt nijak nevadí.[24]

3.1.2 Odvozovací pravidla

Uživatelovo zadání se skládá typicky jen ze seznamu výstupů a vstupů pro každý zadaný výstup a z nastavení některých atributů, které určují způsob naložení se zadanými vstupy a výstupy. V grafové podobě může uživatelovo zadání vypadat jako graf na obrázku 2. Takové zadání však nepostihuje mezikroky celého procesu, které představují jednotlivé aktualizační akce – zadání bývá pro používané aktualizační nástroje zpravidla neúplné a příliš zjednodušené.

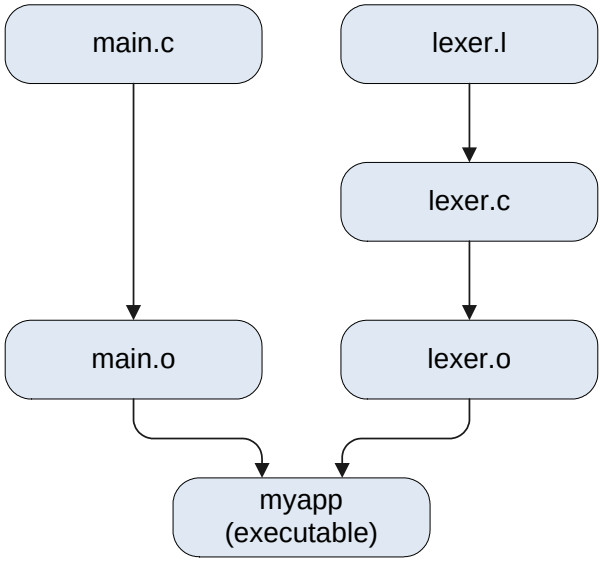
[23] Místo označení graf závislostí (dependency graph) by tedy v *EBM* možná bylo vhodnější označovat tento graf jako graf toku dat (data flow graph).

[24] Implementace také skutečně s multigrafem pracuje, neboť je to snazší a není třeba hlídat duplicitu hran. Lze předpokládat, že nedojde k významnému růstu počtu paralelních hran, a tedy že režie na hlídání a odstraňování duplicit by byla daleko vyšší než jejich ponechání.



Obrázek 2: Primární graf. Schéma ukazuje graf závislostí v podobě, ve které jej zadává uživatel. Cíl *myapp* má zadán i typ, neboť na typu tohoto cíle závisí aplikace pravidel a jeho určení není jednoznačné.

Jelikož data zadaná uživatelem, ať již přímo nebo nepřímo, bývají nedostatečná a neúplná, je třeba doplnit je o mezikroky potřebné pro nasazení nástrojů, které mají být použity k aktualizaci. O splnění tohoto úkolu se starají odvozovací (derivační) pravidla. Kromě toho, že dovodí potřebné aktualizační mezikroky pomocí úpravy grafu závislostí, doplňují také hodnoty atributů cílů, které aktualizační akce potřebují a využívají. Ukázku, jak se může použití odvozovacích pravidel projevit na grafu závislostí, představuje obrázek 3.



Obrázek 3: Sekundární graf. Tento graf vznikne aplikací odvozovacích pravidel na primární graf, který je uveden na obrázku 2. Odvozovací pravidla se mohou řetězit, takže z cíle *lexer.l* bude odvozen prvně cíl *lexer.c* a aplikací stejného pravidla, které zpracovává cíl *main.c*, bude odvozen i *lexer.o*.

Uživatelem zadaný graf, který následně má být zpracován odvozovacími pravidly, bude v dalším textu nazýván primární. Graf, který vznikne zpracováním primárního grafu odvozovacími pravidly bude nazýván sekundární, nebo také odvozený. Vznik sekundárního grafu se řídí následujícími zákonitostmi.

- Cíle primárního grafu se při odvozování zpracovávají v pořadí, které respektuje závislosti – tedy v pořadí respektujícím topologické uspořádání grafu. Z tohoto požadavku vyplývá, že primární graf musí být acyklický.
- Změna atributů zpracovávaných cílů během procházení grafu neovlivňuje tvar primárního grafu. To dovoluje odvozovacím pravidlům modifikovat závislosti pro sestavení sekundárního grafu, kterým se řídí aktualizace, aniž by bylo

narušeno pořadí odvození cílů v primárním grafu definované předchozím odstavcem.

- Odvozovací pravidlo může při zpracování nějakého cíle C vytvořit nové cíle $N_1, N_2, ..., N_n$; tyto nové cíle musí být zpracovány před zpracováním všech cílů $C_1, C_2, ..., C_n$, které jsou závislé na cíli C. Tento požadavek se však týká i dalších cílů, které mohou vzniknout zpracováním cílů $N_1, N_2, ..., N_n$. Navíc všechny nově vzniklé cíle mohou mít při svém vzniku nastaveny závislosti mezi sebou, ale i vzhledem k již existujícím cílům, a tyto závislosti je nutné při dalším odvozování respektovat. Výsledné zpracování nově vzniklých cílů by mělo být ekvivalentní situaci, kdy by byly vytvořeny explicitně uživatelem.

  Snadno lze tohoto chování dosáhnout použitím triku: všechny nově vytvářené cíle se vkládají do primárního grafu, včetně hran představujících jejich závislosti, a na vložených cílech se nechají uměle záviset cíle $C_1, C_2, ..., C_n$. Tyto uměle přidané závislosti zaručí, že cíle $C_1, C_2, ..., C_n$ budou zpracovány až po zpracování nově přidaných cílů přímo i nepřímo odvozených od cíle C. Nutné je ovšem upravit algoritmus pro topologické třídění tak, aby dovoloval přidávání vrcholů a hran grafu i během svého chodu.[25]

- Po odvození všech cílů, včetně nově vzniklých, je sestaven sekundární graf. Ten již bere v úvahu nové hodnoty atributů, může tedy vypadat zcela jinak než původní primární graf. Sekundární graf musí být acyklický, protože i jej bude nutné procházet později podobným způsobem jako primární graf.[26]

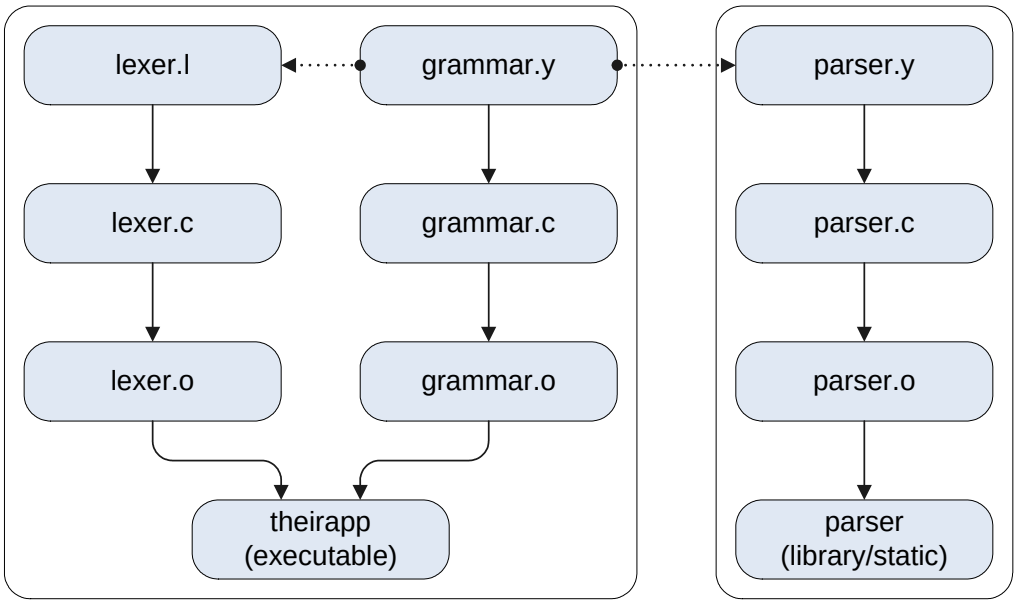
3.1.3 Aktualizace

Pro tuto chvíli bude opominut popis získávání a využití implicitních závislostí, neboť to je značně ovlivněno implementačními detaily. Následující pasáž se proto soustředí pouze na algoritmus, kterým se řídí aktualizace.

Sekundární graf vzniklý výše popsaným způsobem již představuje detailní plán aktualizace, který zachycuje tok dat mezi aktualizačními kroky i pořadí, ve kterém je možné je provádět. *EBM* využívá forward chaining strategii ve stylu „fire wall“ podobně jako *amake*, ale nedotahuje ji do extrému jako optimistický *make*, který aktualizuje sám vše, co aktualizovat lze. Graf totiž může obsahovat spoustu dílčích cílů a uživatel může chtít v daný okamžik aktualizovat jen vybrané konkrétní cíle. *EBM* toto zohledňuje a zpracuje jen tu část grafu, která je potřebná pro aktualizaci zadaných cílů. Prvně je tedy podle sekundárního grafu vytvořen graf s obrácenou orientací hran a jeho průchodem od cílů, které si uživatel přeje aktualizovat, se označí cíle, na nichž je požadovaný výsledek závislý. Zde se také uplatňuje rozdíl mezi synchronizačními a vstupně-výstupními hranami: pokud je některý cíl v „inverzním“ grafu s obrácenou orientací hran dosažitelný pouze synchronizačními hranami, může být z aktualizace vynechán, což snižuje počet aktualizačních kroků, a tedy i výslednou dobu aktualizace.

[25] Tato úprava však může (ba dokonce by měla) ignorovat vznik cyklů, které by obsahovaly již zpracované vrcholy. V sekundárním grafu totiž tyto cykly už vzniknout nemusí.

[26] Hledání vhodného způsobu, jak se vyhnout požadavku na acykličnost sekundárního grafu, a jeho praktické vyzkoušení může být též námětem pro budoucí rozvoj programu. Požadavek acykličnosti totiž představuje jeden z důvodů, proč nejsou implicitní závislostí zařazeny přímo do sekundárního grafu, jak ostatně bude zmíněno v pazáži *Implicitní závislosti*.



Obrázek 4: Sekundární graf se synchronizačními hranami. Schéma ukazuje sekundární graf pro dva nezávislé podprojekty (znázorněny ohraničením příslušejících cílů); graf obsahuje synchronizační hrany (znázorněné tečkovanými šipkami), které vedou z nějakého důvodu i mezi cíli různých podprojektů – např. proto, že uživatel smí spustit jedinou instanci nástroje pro zpracování *.y* souborů, což synchronizační hrana může zajistit. Použití synchronizačních hran však nezanáší datovou závislost, a tak zpracování jednoho podprojektu nevynucuje automaticky zpracování druhého podprojektu.

Průchod acyklickým sekundárním grafem s uzly označenými ke zpracování či vynechání (dle výše popsaného postupu) lze řídit pomocí haldy s využitím následujícího algoritmu.

1. Uzly grafu se vloží do minimové haldy klíčované vstupním stupněm vrcholů grafu – aktualizace vrcholů s nenulovým klíčem je blokována jejich předchůdci v grafu. Dokud není halda prázdná, opakuj kroky 2–4, po vyprázdnění haldy pokračuj bodem 5.
2. Pokud vrchol haldy má nulový klíč, pokračuj bodem 4.
3. Pokud neprobíhá aktualizace žádného vrcholu cíle, přeruš iteraci kroků 2–4 a pokračuj bodem 6. Pokud však probíhá aktualizace některého vrcholu, čekej, dokud neskončí a přejdi zpět k bodu 2.

   Při úspěšném skončení aktualizace cíle *C* vždy sniž každému vrcholu *V* grafu jeho klíč v haldě o počet hran[27], které vedou z *C* do *V*. Nastav vrcholu *V* příznak vynucené aktualizace – tento příznak znamená, že vstupy pro akci vrcholu *V* se změnily a *V* tedy bude potřeba aktualizovat, aniž by bylo třeba vyhodnotit a zohlednit ostatní kritéria aktuálnosti.
4. Odeber vrchol haldy. Jeho aktualizaci spusť tehdy, pokud vrchol patří do části grafu, která má být aktualizována, a zároveň je označen příznakem vynucené aktualizace nebo je vyhodnocen jako neaktuální. Jestliže aktualizace nemá být vykonána, sniž klíče vrcholů na něm závislých (ale nenastavuj jim příznak vynucené aktualizace). Pokračuj bodem 2.

[27] Toto je důležité pro případ multigrafu; v obyčejném grafu snaží snížit klíč o jedničku každému následníkovi vrcholu C.

5. Počkej na výsledek ještě probíhajících aktualizačních operací.
6. Pokud je halda prázdná a všechny proběhnuté aktualizační operace skončily úspěchem, je úspěšně aktualizován i požadovaný výsledek. V opačném případě výsledek nebyl aktualizován úspěšně a halda obsahuje ty cíle, u nichž aktualizace nebyla ani spuštěna z důvodu neúspěchu aktualizace jejich předchůdců v grafu.

Popsaný algoritmus, jak je patrné, umožňuje paralelní vykonávání aktualizačních akcí. Použití haldy popsaným způsobem, který vlastně tvoří prioritní frontu, zaručí respektování závislostí blokováním vrcholů grafu závislých na ještě neaktualizovaných vrcholech a zároveň dovolí maximální paralelizaci.

Fungování tohoto algoritmu lze ještě vylepšit tak, aby byl dosažen efekt „fire wall“ strategie použité *amake*. Aktualizační operace nemusí indikovat pouze úspěch či neúspěch. Může také navrátit hodnotu indikující, že výsledek je totožný s předchozím stavem. V takovém případě se modifikuje chování bodu 3 z algoritmu: při výsledku shodném s minulým stavem není třeba vynucovat aktualizaci závislých cílů. Zbytek algoritmu zůstává beze změny.

3.1.4 Součinnost pravidel

Protože odvozovací pravidla jsou alespoň z části dělaná na míru konkrétním nástrojům, není ve většině případu vhodné jejich přímé spojování s cíli, ať již explicitně uživatelem nebo prostřednictvím nějaké deklarace. *EBM* proto vkládá do celého procesu ještě jednu úroveň pravidel, která mají za úkol detekovat typ cíle (např. podle přípony příslušného souboru). Každý typ cíle může mít přiřazeno implicitní odvozovací pravidlo, které je použito, pokud není nastaveno explicitně pravidlo jiné. Hlavní kroky celého procesu ve vší stručnosti shrnuje následující schéma.

1. Uživatel deklaruje cíle a jejich vzájemný jednoduchý vztah, jak je ukázáno na příkladě na obrázku 2.
2. Cíle, které nemají určený typ, jsou sadou detekčních pravidel otypovány.
3. Je sestaven primární graf a ten je zpracován pomocí odvozovacích pravidel. U cílů, které nemají přiřazeno odvozovací pravidlo, se použije implicitní odvozovací pravidlo pro jejich typ. Odvozovací pravidlo doplní a nastaví cíli atributy a typicky jej také spojí se vhodným aktualizačním pravidlem (akcí).
4. Z výsledků odvození je sestaven sekundární graf. Příkladem takového grafu je obrázek 3.
5. K sekundárnímu grafu je vytvořen graf s hranami s opačnou orientací. Průchodem tohoto „inverzního“ grafu od cílů, které si uživatel přeje aktualizovat, se označí cíle, které nemají být ignorovány – viz obrázek 4.
6. Vrcholy sekundárního grafu se vloží do haldy klíčované svým vstupním stupněm a výše popsaným algoritmem se aktualizují. O spuštění aktualizačních nástrojů se starají aktualizační akce cílů, které mohou využívat atributy nastavené uživatelem i odvozovacími pravidly.

3.2 Návrh implementace

3.2.1 Jazyk pro popis konfigurace

Zvolen byl přístup, kdy je použit jediný jazyk pro popis aktualizačních nástrojů i pro popis konfigurace vytvářený běžným uživatelem. Inspirací pro jazyk *EBM* je jazyk používaný *Jamem*, zejména způsob jeho běžného používání, který víc než procedurální programování připomíná deklarace vlastností a vztahů různých objektů.

Jazyk *EBM* je tedy také procedurální povahy. Rysy klasických procedurálních jazyků jsou v něm však výraznější, než je tomu např. u *Jamu* nebo i *Boost Buildu*. Jedná se zejména o striktnější lexikální i syntaktická pravidla, než je u build managerů zvykem.[28] Nevýhodou striktnějších pravidel je používání většího počtu pomocných symbolů. Výhodou je naopak díky nim menší riziko, že uživatel bude svůj zápis interpretovat jinak než program, který se jím bude řídit. Striktnější syntaxe také usnadnila implementaci *EBM*, neboť umožnila hladké použití běžných nástrojů pro tvorbu překladačů.

Jako každý procedurální jazyk, umožňuje i jazyk *EBM* deklarovat globální a lokální proměnné. Během vývoje *EBM* se experimentovalo s netypovanými proměnnými, ale ukázalo se, že typování je pro uživatele výhodnější, neboť jednak zpřehledňuje zdrojový kód, protože je jasnější, co která proměnná obsahuje, a jednak usnadňuje některé manipulace s hodnotami v proměnných; proměnné *EBM* jsou tedy typované. Na základě experimentů byla zvolena jako dobře použitelná následující kombinace typů: základními dvěma typy jsou řetězec (string) a jméno cíle (target), z těchto základních typů jsou odvozeny ještě typy seznam řetězců (string-list), množina řetězců (string-set) a množina cílů (targets). Pro některé speciální účely je k dispozici ještě typ unknown, který se za běhu přizpůsobí vloženému obsahu. Mezi typy existují automatické typové konverze, takže typování nijak nekomplikuje uživateli zápis.

Na cíle lze pohlížet jako na instance struktury (ve smyslu struktur např. jazyka C), jejíž prvky jsou atributy. Atributy jsou ovšem také typované – např. již známé atributy ***Inputs***, ***Outputs*** a ***WaitFor*** jsou typu targets. Rozšiřování množiny atributů se pak děje jejich deklarací, která určí typ atributu a případně i další jeho vlastnosti. Jiné typy ani typové konstrukce pro jednoduchost jazyk neposkytuje, nicméně toto typování je dostatečně silné, aby umožnilo plnění úkolu, pro který byl jazyk *EBM* navržen.

3.2.2 Pojmenování cílů

Řada build managerů ztotožňuje cíle – uzly grafu závislostí – se jmény souborů, které mají být s těmito cíli spjaty. Tento přístup má nespornou výhodu v uživatelské jednoduchosti a intuitivní přímočarosti. O to víc má ale nevýhod.

- Určitým problémem mohou být virtuální a další speciální cíle, které s žádným souborem spjaté nejsou.

[28] Abstraktní gramatiku jazyka včetně vysvětlení jednotlivých syntaktických i lexikálních konstrukcí lze najít na přiloženém disku CD-ROM.

- Komplikace mohou snadno vzniknout, pokud se užívají pouze jména souborů, bez cest, které by dané soubory jednoznačně určily, a v projektu existuje více souborů stejného jména, jen v různých adresářích.
- Používání adresářových cest v souvislosti s pojmenováním cílů znamená, že cíl může být odkazován pomocí relativních cest různými řetězci. Pokud build manager tuto situaci nějak neošetřuje a manipuluje slepě pouze s řetězci bez znalosti jejich významu, může snadno dojít k úplnému chaosu, v němž se uživateli ztratí informace, se kterým cílem se ve skutečnosti pracuje.
- Použití adresářových cest ve jménech cílů je nešťastné i z důvodů pozdější reorganizace adresářové struktury projektu. Výsledkem může být nutnost rozsáhlých změn a revizí popisu konfigurace.
- V neposlední řadě je problémem platformně závislý zápis adresářových cest. Tento problém je možná dokonce ten nejvážnější, protože se dotýká úzce přenositelnosti popisu konfigurace mezi platformami.

EBM se snaží tyto problémy vyřešit zavedením tří oddělených úrovní jmen.

1. První úroveň tvoří jména uzlů grafu závislostí; nemají bezprostřední souvislost se jmény souborů, proto se také nazývají virtuálními jmény. Jejich účelem je odstínit odkazy na cíle (a jejich atributy) v popisu konfigurace od fyzické adresářové struktury se soubory, se kterými se manipuluje.

   Neboť virtuální jména jsou v celém popisu konfigurace unikátní, je možné se na každý cíl (uzel grafu závislostí) jednoznačně odkázat. Virtuální jména tvoří hierarchii, která se řídí podobnými principy jako adresářové hierarchie souborových systémů; dovoluje tedy snadnou orientaci a strukturování.

2. Druhou úrovní jmen, která má za úkol odstínit popis konfigurace od platformně závislých zápisů adresářových cest, představují logická jména. Logická jména souborů se uvádějí v unifikovaném, platformně nezávislém tvaru a používají se v popisu konfigurace všude, kde se pracuje se jmény souborového systému.
3. Třetí úrovní jmen jsou již jména fyzická nebo také nativní, která představují platformně závislý tvar. Fyzická jména vzniknou překladem logických jmen a používají se až při sestavování příkazů, kde figurují např. jako parametry.

Aby nebyla práce s tolika úrovněmi jmen příliš komplikovaná, *EBM* má několik mechanismů, které řadu úkonů s nimi zautomatizují a usnadní běžné používání tohoto trojvrstvého modelu tak, že rozdíl oproti používání přímých jmen souborů není z hlediska běžného uživatele příliš markantní. Jedním z těchto mechanismů je zavedení aktuálního virtuálního a logického adresáře, jimiž jsou doplněna neúplná virtuální a logická jména při svém zadávání, a spojení změn těchto aktuálních adresářů se skládáním podprojeků.[29]

Vedlejším efektem, který oddělení jmen souborů od jmen cílů provází, je možnost jeden fyzický soubor svázat s několika cíli a ty zpracovávat různými postupy. Tato vlastnost může v některých případech být výhodná, má však i jednu stinnou stránku, která hraje roli ve hledání implicitních závislostí a která bude zmíněna později v podkapitole *3.2.5 Implicitní závislosti*.

[29] Podrobnosti jsou zmíněné v uživatelské příručce; viz zejména funkce *include*. Patrný by tento koncept měl být také z ukázek v *Příloze A*.

3.2.3 Vazba jmen a viewpathing

Virtuální jméno cíle je s cílem pevně spjato po celou dobu jeho existence; uloženo je i ve vestavěném konstantním atributu ***VirtualPath***. Logickou (a tedy ani fyzickou) cestu cíl vůbec nastavenou mít nemusí – v takovém případě se jedná o virtuální cíl.

Nevirtuální cíle mají nastavenou cestu k odpovídajícímu souboru. Tato cesta však není nastavena hned od začátku pevně. Prvním důvodem je, že odvozovací pravidlo může chtít cestu změnit, např. zařídit, aby se soubor vytvořil ve specifickém adresáři – to je velmi důležité pro podporu variant. Druhým důvodem je podpora viewpathingu, který je zase velmi užitečný pro vyhledávání implicitních závislostí. Pevné svázání cíle s fyzickým souborem je tedy odloženo a nastává až po provedení odvozovacího pravidla nad daným cílem, případně jej lze vynutit vestavěnou funkcí i dřív.

Viewpathing je při vazbě jmen podporován následujícím způsobem. Logická cesta k souboru je rozdělena a uložena ve dvou atributech. Prvním je ***Name***, který obsahuje vlastní jméno souboru. Druhým je pak atribut ***Dirs***, který obsahuje celý seznam adresářů, ve kterých se soubor potenciálně může nacházet. Když je vyvolána vazba jmen, je tento seznam adresářů z atributu ***Dirs*** prohledán, aby byl požadovaný soubor nalezen.[30] Do atributu ***LogicalPath*** je pak uložena výsledná logická cesta k souboru a do atributu ***NativePath*** pak její platformně závislá varianta. Jakmile jsou jména navázána, jsou příslušné atributy zmrazeny a nelze je již změnit.

3.2.4 Statefile

Ukládání aspoň části kontextu aktualizační akce je nutné, jak již bylo také několikrát zmíněno. Jelikož v případě *EBM* je kontext pro aktualizační akce soustředěn v atributech cílů, slouží statefile právě k uložení atributů.

Pravidla mohou uložené hodnoty načíst a také využít, typické využití uložených atributů je však automatické. Některé atributy může uživatel označit jako klíčové pro aktualizační proces. V momentě, kdy se rozhoduje o aktuálnosti cíle, porovnají se momentální hodnoty takto označených atributů s jejich uloženými hodnotami, a pokud jsou různé, cíl je ihned vyhodnocen jako neaktuální. Příkladem přirozených kandidátů pro klíčové atributy jsou atributy, ve kterých si odvozovací pravidlo přichystá a uloží významnou část příkazové řádky pro spuštění aktualizačního nástroje.

Statefile je využíván také k ukládání grafu implicitních závislostí, o jehož vytváření pojednává hned následující pasáž.

3.2.5 Implicitní závislosti

Hledání implicitních závislostí komplikuje v *EBM* právě existence virtuálních jmen a možnost asociace jednoho fyzického jména s více virtuálními. Výsledkem nalezení implicitní prerekvizity v podobě souboru je (fyzická) cesta k němu – problém spočívá v určení, se kterými virtuálními jmény má být asociována. Navíc toto rozhodnutí nelze ani učinit dřív, než vazba jmen proběhne na všech cílech, které by mohly připadat v úvahu. Celou úlohu komplikuje i fakt, že implicitní závislosti mohou být cyklické, jenže graf závislostí musí zůstat acyklický. Z těchto důvodu vyhledání implicitních závislostí mezi cíli a doplnění příslušných hran v grafu závislostí není *EBM* nijak zvlášť

[30] Chování *EBM* v případě, že soubor nebude nalezen, je ovlivněno dalším atributem, atributem ***Binding***.

podporováno, ačkoliv je možné v nutných případech pomocí komplikovaného triku a externího programu požadovaného efektu také dosáhnout.

Podpora *EBM* ve vyhledávání implicitních závislostí se zaměřuje na typičtější případ, kdy hrany v grafu závislostí už sice zaručují správné pořadí aktualizace jednotlivých cílů, ale nepokrývají všechny soubory, na nichž jsou cíle závislé. Typickým příkladem jsou jazyky C a C++ se svými moduly a hlavičkovými soubory – s moduly je aktivně manipulováno, jsou přirozenými vstupy a musí být obsaženy v popisu konfigurace a v grafu závislostí, ale hlavičkové soubory, na kterých moduly závisí, mají jen pasivní roli. Implicitní prerekvizity tohoto druhu umožňuje *EBM* vyhledat jak s pomocí externího programu, tak i s využitím vestavěné podpory regulárních výrazů.

Postup vyhledávání

Za pozornost stojí způsob, jakým je do *EBM* prohledávání souborů zabudováno a jak funguje, neboť představuje ve třídě samostatných build managerů zřejmě ojedinělé řešení. Záměrem je dosáhnout minimálního počtu časově náročných vyhledávacích operací.

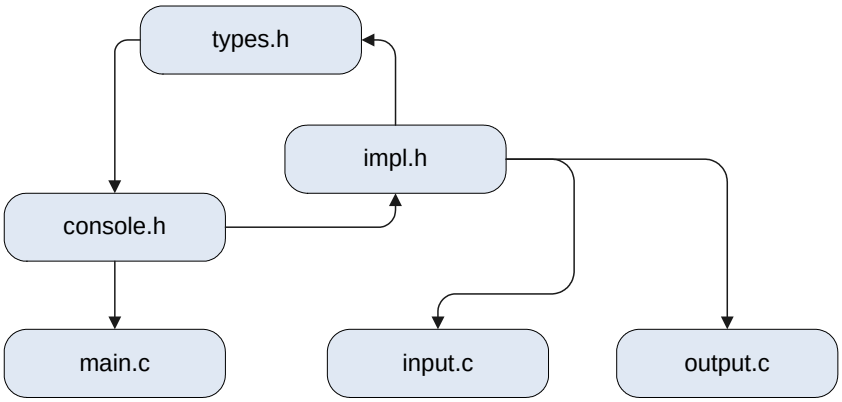
Každý cíl může být asociován s nějakým scannerem. Scanner je funkce v jazyce *EBM*, jejímž vstupem je cíl, který má být prohlédnut na výskyt implicitních prerekvizit, a výstupem seznam nových pomocných cílů, které představují prerekvizity vstupu a jimž scanner musí nastavit atributy pro další zpracování. Tyto nové cíle, které scanner vrací, jsou nejprve podrobeny vazbě jmen souborů; zde je patrný význam popsaného viewpathingu: velmi snadno se zařídí vyhledání souboru ve více možných lokacích. Ty cíle, k nimž se povedlo soubor nalézt, jsou předány scanneru, který je s nimi asociován, k dalšímu zpracování – nemusí být ovšem asociovány nutně s tím samým scannerem, který je vytvořil. Opakováním tohoto postupu je postupně po vrstvách vytvořen celý graf implicitních prerekvizit, který může být i cyklický (na rozdíl od grafu závislostí cílů).

Velkou úsporu prohledávání lze dosáhnout díky pozorování, že řada implicitních prerekvizit bývá mezi cíli sdílená, jak demonstruje na příkladu obrázek 5. Je tedy možné za následujících předpokladů prohledat každý soubor pouze jednou a výsledky sdílet mezi více cíli.

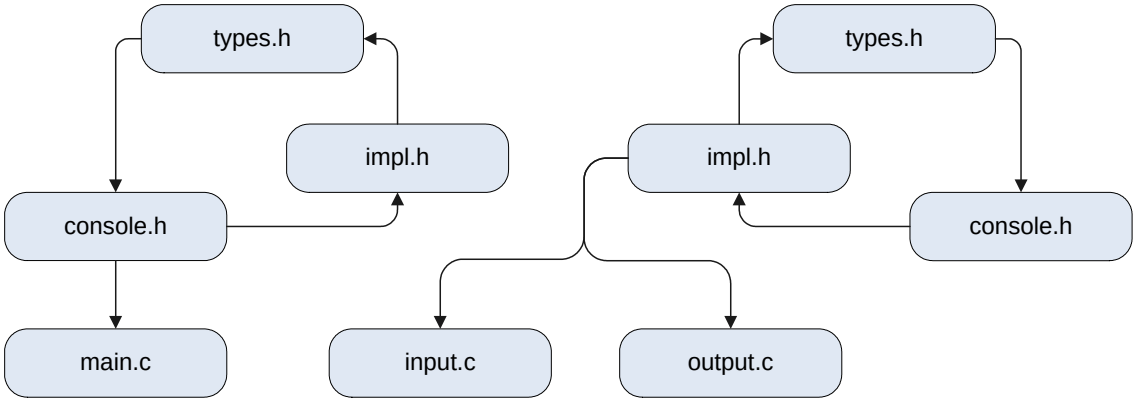
Prvním předpokladem pro využití zmíněného pozorování je, aby scanner vracel pouze bezprostřední prerekvizity prohledávaného souboru, což dovolí vytvářet graf postupně „po vrstvách“. Použití externího programu, který vrátí všechny, i nepřímé, prerekvizity naráz, sice nic zásadního nepokazí, jenže také nedovolí nic sdílet, neboť se ztratí informace o struktuře závislostí.

Druhým předpokladem je pak shoda informací (uchovávaných v *EBM* opět v podobě atributů), které mohou výsledky vyhledávání zásadně ovlivnit. Příkladem z jazyků C a C++ je nastavení adresářů, ve kterých preprocesor vyhledává hlavičkové soubory. Tento předpoklad naštěstí většinou platí a cíle mají nastavení scanneru shodná, tedy mohou nalezené implicitní prerekvizity sdílet. Nicméně *EBM* pamatuje i na případ, kdy předpoklad shody nastavení není, a ošetřuje jej, aby nedošlo k získání nesprávných závislostí. Atributy, které scanner používá k vyhledávání a mohou ovlivnit výsledek prohledávání, se označují při své deklaraci jako atributy klíčové pro scanner. *EBM* pak při konstrukci grafu implicitních prerekvizit nesloučí vrcholy, které sice reprezentují

stejné soubory, ale byly nalezeny různými scannery nebo mají různé hodnoty atributů klíčových pro scanner.[31] Tuto situaci demonstruje obrázek 6.



Obrázek 5: Graf sdílených implicitních závislostí. Graf ukazuje příklad podgrafu sdíleného cíli *main.c*, *input.c* a *output.c*. Sdílet graf lze díky tomu, že tyto cíle používají stejná nastavení klíčových atributů scanneru. Za povšimnutí stojí také fakt, že graf implicitních závislostí může obsahovat cykly, což není možné v grafu závislostí mezi cíli, který je vytvořen odvozovacími pravidly.



Obrázek 6: Graf částečně sdílených implicitních závislostí. Cíl *main.c* používá jiná klíčová nastavení scanneru než cíle *input.c* a *output.c*, nelze tedy graf implicitních závislostí cíle *main.c* sdílet s grafem cílů *input.c* a *output.c*, jejichž klíčová nastavení se naopak shodují, a tedy mohou své grafy sdílet.

Uchovávání získaného grafu

Samotné sdílení částí grafu implicitních prerekvizit velmi šetří nároky na jejich vyhledávání, ale je jen polovinou optimalizace, kterou *EBM* provádí. Druhou polovinu tvoří ukládání grafu do statefile a využívání uložených informací.

Kromě této optimalizace má ukládání grafu do statefile ještě jeden důvod: *EBM* používá statefile také pro uchovávání času poslední modifikace souborů asociovaných s cíli, nespoléhá jen na relativní časové rozdíly mezi cíli, je tedy nutné uchovávat i čas poslední modifikace implicitních prerekvizit.

[31] Implementace je v tomto bodě mírně zjednodušená: aby nebylo nutné uchovávat a porovnávat všechny takové atributy, počítá se z nich pouze hash a pracuje se jen s ním. To není sice stoprocentně spolehlivý přístup, ale pro tento účel postačuje.

Před spuštěním aktualizačního algoritmu je z údajů ve statefile celý graf zrekonstruován a zjištěna aktuálnost příslušných souborů vůči uchovaným údajům. Soubory, které nebyly změněny, už není nutné znovu prohledávat a příslušnou část grafu konstruovat znovu, ostatní soubory jsou z grafu odstraněny. Stav souborů jednou zanesených do grafu už není během aktualizačního procesu znovu zjišťován; z toho vyplývá požadavek, aby implicitní prerekvizity nebyly změněny během aktualizačního procesu, neboť graf je sice během něj aktualizován, ale uzly do něj jednou zanesené již nejsou znovu kontrolovány.[32]

3.2.6 Kritéria pro aktuálnost cílů

Aktualizační algoritmus při vyhodnocování aktuálnosti cíle bere v úvahu několik kritérií:

- výsledek aktualizace prerekvizit představovaný příznakem vynucené aktualizace,
- porovnání momentálních hodnot klíčových atributů s hodnotami uloženými ve statefile,
- změnu implicitních prerekvizit,
- porovnání časů poslední modifikace (časových značek) příslušných souborů s hodnotami zapamatovanými ve statefile.

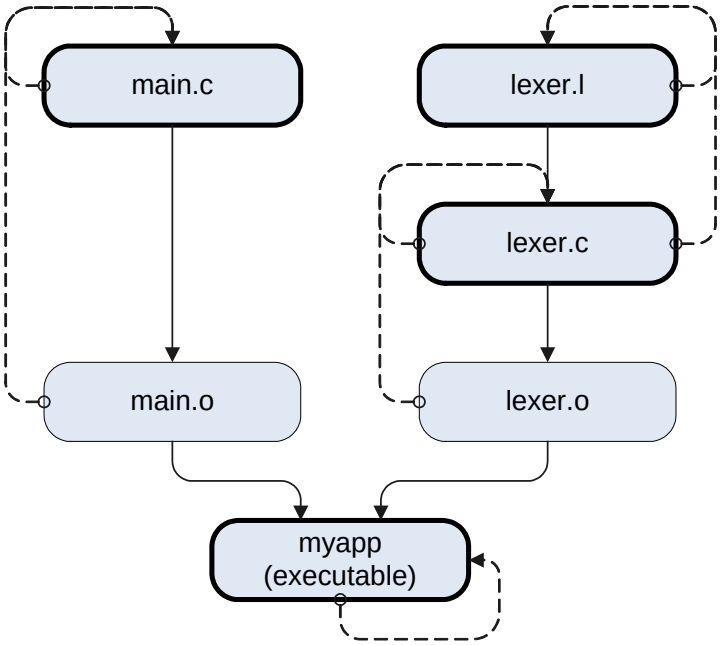
První tři kritéria byla již zmíněna. V případě posledního kritéria, které dosud rozebíráno nebylo, vstupuje do hry drobná komplikace v podobě samotné forward chaining strategie. Většina typických případů totiž vyžaduje, aby se aktuálnost zpracování některého cíle odvíjela také od časových značek výstupu jeho zpracování, tedy od cílů, které mají teprve být vyhodnocovány vzhledem k postupnému „dopřednému“ průchodu grafu. Příkladem může být překlad zdrojového textu do souboru s kódem v relativním tvaru (object file): zde je třeba vzít v úvahu jak časovou značku zdrojového souboru, tak i značku souboru, který vznikne zpracováním zdrojového souboru.

Jedním řešením této situace, které vlastně používá backward chaining strategie v make, je spojení aktualizační akce s jejím výsledkem místo se zdrojem. Tento přístup však není vhodný pro strategii používanou *EBM*, zvlášť s ohledem na atributy cílů, a také se nedá použít v případě, že akce neprodukuje pouze jeden cíl, nýbrž hned několik cílů naráz.

EBM tedy tento problém řeší jinak a velmi jednoduše – ponechává rozhodnutí na pravidlech a zajistí jim jen potřebnou podporu, aby svá rozhodnutí mohla realizovat. Potřebnou podporu v tomto případě představuje atribut ***DependsOn***, který obsahuje seznam cílů, na jejichž časovém údaji je daný cíl závislý. Zavedení tohoto atributu, který je vyhodnocován dynamicky podle potřeby, dokonce řeší i některé méně typické případy časové závislosti[33] a vhodně doplňuje atributy ***Inputs***, ***Outputs*** a ***WaitFor***.

[32] Toto chování odstraňuje režii spojenou se zjištěním stavu souborů v grafu po každé aktualizaci. Pokud implicitní prerekvizity hrají pasivní roli, není uvedený předpoklad omezující.

[33] Tento atribut se vykládá striktně – cíl dokonce nemusí být časově závislý sám na sobě, pokud není v ***DependsOn*** obsažen, čímž lze také dosáhnout některých zajímavých efektů.



Obrázek 7: Graf závislosti na časových značkách cílů. Cíle se zvýrazněným orámováním mají přirozeným způsobem přiřazenou nějakou aktualizační akci, např. *main.c* je zpracováván překladačem jazyka C, jehož je zdrojovým souborem. Čárkované šipky představují závislost na časových značkách.

3.2.7 Inicializace a sdílení pravidel

Všechny build managery řeší problém, jak zabezpečit sdílení pravidel, ale také jak pravidlům předat informace o prostředí, ve kterém jsou vykonávána (např. o platformě, dostupných nástrojích apod.). O sdílení pravidel bylo již pojednáno v závěrečném shrnutí předchozí kapitoly. Druhému problému, jímž je inicializace pravidel vnějšími informacemi, se věnuje následující text.

Klasickým způsobem, jak této inicializace dosáhnout, je použít proměnné prostředí nebo argumenty na příkazové řádce, které se nějak promítnou do činnosti pravidel. Nevýhodou obou těchto postupů je, že představují kontext konfigurace, který se vymyká konfiguraci samotné a který je nutný zdokumentovat (a zajistit) mimo ni. Na kvalitě případné dokumentace pak závisí i snadná reprodukcibilita výsledku – stačí opomenout např. nastavení některé proměnné prostředí a výsledek může být zcela jiný.

EBM se proto snaží problém inicializace pravidel řešit jinak. V první řadě nemá žádné volby, které by mohly přímo ovlivnit konfigurační skripty z příkazové řádky, ani neimportuje proměnné prostředí jako proměnné skriptů, jak to dělá řada jiných build managerů.[34] Ke konfiguračním skriptům je místo toho přiložen inicializační soubor, který je považován za součást popisu konfigurace a který umožňuje nastavit některé parametry důležité pro program samotný a také nastavit inicializační argumenty, na něž se může skript pomocí vestavěné funkce dotázat.

Ve skutečnosti jsou inicializační soubory dva; jeden (typicky pojmenovaný buildfile) je součástí konfigurace a měl by být přiložen ke konfiguračnímu skriptu.

[34] Pravidla však mohou použít vestavěnou funkci, aby zjistila hodnoty proměnných prostředí, neboť to může pomoci v některých případech např. detekci platformy; skripty *EBM* tedy nejsou o přístup k proměnným prostředí ochuzeny.

Druhý (pojmenovaný defaults) je uložen v instalaci programu samotného – ten je načten jako první a jeho nastavení jsou doplněna či předefinována nastaveními buildfile. Důvodem tohoto postupu je existence některých typických inicializačních argumentů.

Jedním z takových argumentů, který pravidla bude velmi ovlivňovat, je identifikace platformy, na které program právě běží. Nemá smysl tuto hodnotu uvádět v buildfile, protože při přenosu konfigurace na jiný počítač by bylo nutné ji měnit (anebo mít více variant buildfile, což se může také hodit, ale v tomto případě je to zbytečné). Naopak je ideální ji uvést v souboru defaults a tím nastavit parametry konkrétní instalace *EBM*. Podobným způsobem lze skriptům sdělit např. seznam nástrojů, které jsou na konkrétním počítači k dispozici. Aniž je třeba zasahovat do konfigurace projektu, dozvědí se skripty vše potřebné o systému, na kterém jsou vykonávány.

Lze si povšimnout, že inicializační skripty fungují podobně jako mechanismus „inverzního“ sdílení skriptů použitý *Jamem*. *EBM* však neužívá tohoto „inverzního“ sdílení pro skripty, nýbrž jen pro inicializační soubory. Pro skripty to již není potřeba. Jednou z věcí, kterou inicializační soubory obsahují, je totiž jméno skriptu, který má být proveden, a seznam adresářů pro nalezení skriptů s opakovaně využitelnými pravidly, které mohou být tímto primárním skriptem importovány. Není snad nutné zdůrazňovat, že vyhledávací cesty jsou uváděny ve tvaru logických jmen, že mohou být relativní a že je přesně definováno, vůči kterému adresáři budou relativní cesty vztaženy – použité řešení si vystačí pouze s uvedením relativních cest a není ani závislé na aktuálním pracovním adresáři, ze kterého se *EBM* spouští. Tím je zaručeno nejen snadné přenášení konfigurace bez nutnosti změn, byť jen změn vyhledávacích cest ke sdíleným projektovým pravidlům, nýbrž je také zaručeno použití těch správných pravidel nehledě na proměnné okolnosti, jakými je např. aktuální adresář při spouštění build manageru.

3.2.7 Podpora variant

Podpora variant výsledku vychází ze čtyř pozorování:

1. výběr konfigurace pro konkrétní variantu by měl být jednoduchý,
2. vlastnosti varianty by měly být stabilní a vycházet z vybrané konfigurace,
3. je výhodné mít možnost varianty navzájem od sebe odvozovat i nastavovat jejich společné rysy dohromady,
4. je nutné varianty od sebe dobře separovat, aby zpracování více variant vedle sebe nebylo ovlivněno nežádoucím způsobem.

Výběr a stabilita variant

S ohledem na první pozorování má v *EBM* každá sada vlastností, definující nějakou variantu, jednoduché a jednoznačné symbolické pojmenování. Výběr varianty se děje zadáním tohoto pojmenování jako argumentu programu, případně lze implicitní variantu nastavit v inicializačním souboru.

Všechny vlastnosti varianty musí být obsaženy v konfiguračním skriptu, *EBM* nepodporuje nastavování vlastností variant např. argumenty z příkazového řádku z těch samých důvodů, které byly zmíněny v předchozí pasáži věnované inicializaci skriptů. *EBM* také kontroluje, aby vybraná varianta skutečně byla konfiguračními skripty

definovaná a povolená. Důsledkem tohoto opatření je lepší reprodukcibilita výsledku a větší odolnost vůči chybám uživatele.

Odvozování variant

Ve většině případů se od sebe varianty projektu liší jen v některých drobnostech – např. na různých platformách se užívají různé podpůrné knihovny, je vyžadováno přidání některých specifických souborů do projektu atd. Většina rysů jednotlivých konfigurací je však shodná. Nabízí se přirozené odvozování variant od sebe, kdy odvozená varianta má všechny rysy původní varianty, ale přidá k nim vlastní rysy, nebo některé zděděné rysy změní. Vzniká tak přirozená hierarchie variant. Občas je však také potřeba nastavit některé vlastnosti celé skupině variant, bez ohledu na jejich příbuznost.

Jak již bylo zmíněno, varianty v *EBM* mají jednoduchá symbolická jména. Jelikož *EBM* podporuje hierarchie variant, jsou i tato jména hierarchická; příkladem jmen variant mohou být např. ***win32.release.static*** nebo ***linux.debug.shared***.

Uživatel při spouštění *EBM* výše popsaným způsobem vybere variantu, která má být aktualizována. Konfigurační skripty mají k dispozici konstrukt, který povolí nebo zakáže zpracování některé části skriptu vzhledem k vybrané variantě. Ten je možné i vnořovat, čímž se dosáhne hiearchického efektu, nebo aplikovat na skupiny variant bez ohledu na hierarchii. Skripty také deklarují, které varianty jsou legální, a tak např. varianta ***win32.debug.release*** nemusí být povolena, protože by nedávala dobrý smysl.

Separace variant

Je nutné zajistit, aby uživatel mohl mít rozpracováno vedle sebe několik variant a jejich aktualizace se navzájem neovlivňovaly, např. aby si nepřepisovaly odvozené soubory.

EBM tuto práci ponechává v podstatě celou na pravidlech. Vzhledem k možnostem odvozovacích pravidel není problém zcela změnit adresáře, kam mají být odvozené soubory umístěny, a dokonce dle typů souborů zajistit, aby pomocné soubory byly umístěny do adresářů mimo výsledek v podobě spustitelných souborů a sdílených knihoven.

Jedinou podporou separace variant, kterou *EBM* má zabudovanou a neponechává ji na pravidlech, je separace stavových informací, která je ovšem vzhledem k jejich roli nutná. Každá varianta tedy má svůj vlastní statefile.

4. Implementace

Obsahem této kapitoly jsou poznámky k implementaci samotné a také zhodnocení jejích výsledků a zkušeností s ní.

4.1 Implementace a prostředí

Program byl implementován v jazyce C++, který byl pro tento účel vybrán díky své rozšířenosti, výkonnosti, podpoře ze strany rozšířených nástrojů pro syntaktickou a lexikální analýzu, podpoře objektově orientovaného programování, relativně snadné přenositelnosti a v neposlední řadě v jednoduchosti nestandardních řešení, která by mohla být vynucena okolnostmi. Lexikální analýza byla zajištěna programem *flex* a syntaktická programem *bison*.[35]

EBM byl realizován jako konzolová aplikace a s přihlédnutím na požadavek své maximální přenositelnosti. Na cílovém systému není tedy vyžadována přítomnost dynamického linkeru ani podpora multithreadingu. Kromě standardní knihovny jazyka C nepoužívá žádné další knihovny; místo použití standardní knihovny jazyka C++ byla vytvořena vlastní implementace potřebné části její funkcionality, neboť např. hash tabulka není v této knihovně dosud standardizována.

Naprostá většina zdrojového kódu je platformně nezávislá. Platformně závislé části jsou odděleny od zbytku programu jednoduchým rozhraním a mohou být snadno zaměněny při přenosu na jinou platformu. Platformně závislými úkoly, které program musí řešit, jsou: spuštění procesu, čekání na ukončení procesu a zjištění jeho návratového kódu, dotazy na prostředí (týká se to proměnných prostředí, zjištění adresáře pro dočasné soubory, zjištění aktuálního adresáře apod.), některé adresářové a souborové služby, které nejsou uspokojivě pokryty standardní knihovnou jazyka C, a konverze logických a fyzických cest.

Platformně závislé funkce byly implementovány pro platformu Windows (tj. pro Win32 API) a pro unixovské systémy. Pro úspěšné přeložení programu je nutné použít překladač jazyka C++, který dovede dostatečně dobře překládat šablony. Nástroje *bison* a *flex* není nutné mít k dispozici, stačí využít již vygenerovaného kódu; ten byl vygenerován verzemi *flex 2.5.4* a *bison 1.35*. Vstupní soubory pro tyto nástroje lze zpracovat i některými staršími verzemi, ale byl zaznamenán problém u příliš starých verzí, které nerozpoznaly některé použité direktivy. Speciálně je třeba dát pozor na překlad vstupů pro *flex*: na unixovských platformách je nutné nejprve ze vstupu odstranit znaky CR (carriage return), které jinak unixovská varianta *flexe* interpretuje jako část výrazu a vygeneruje kód pro zcela jiný automat, než se předpokládá.

[35] Domovskou stránkou *flexe* je http://www.gnu.org/software/flex/flex.html, domovskou stránkou *bisona* http://www.gnu.org/software/bison/bison.html.

V případě některých starších variant *bisonu* může nastat podobný problém a *bison* odmítne soubor se znaky CR přeložit.

Překlad programu byl úspěšně ozkoušen na platformách a překladačích uvedených v následující tabulce.

Hardwarová architektura	Překladač	Operační systém
IA32	*gcc 4.0.1* *gcc 3.4.4*	Linux RedHat
	gcc 3.3.6 *gcc 3.3.5*	Linux Gentoo
	gcc 3.3.5	Linux Debian
	Microsoft Visual C++ 7.1	Windows 2000
AMD64	*gcc 3.3.6* *gcc 3.3.5*	Linux Gentoo
UltraSPARC II	*gcc 3.3.5*	Linux Gentoo
	gcc 3.4.2 *gcc 3.3.5*	SunOS 5.10
MIPS R12000	*gcc 3.2.2*	IRIX 6.5.20

4.2 Výsledky implementace

4.2.1 Použitelnost programu

Jelikož program samotný bez pravidel není použitelný a nelze jeho činnost demonstrovat, bylo jedním z cílů také vytvoření takových ukázkových pravidel, která jeho funkčnost a použitelnost postačujícím způsobem demonstrují. Záměrem autora bylo ukázat použitelnost programu tím, že bude schopen přeložit sám sebe, což zároveň předpokládá vytvoření pravidel pro použité překladače (*gcc* a *Microsoft Visual C++*), linker a další nástroje (*flex*, *bison* a *Microsoft Resource Compiler*). Vytvořená ukázková pravidla nakonec zvládají dokonce 12 typů vstupních souborů a 7 typů různých nástrojů, z nichž většina je připravena pro použití na platformách Windows a Unix. Vzhledem k jejich demonstračním účelům jsou tato pravidla zjednodušená a nepokrývají všechny možnosti, které podporované nástroje nabízejí. Přesto jsou schopna zabezpečit jejich základní užívání, a demonstrovat tak funkčnost programu i jeho principů a myšlenek.

4.2.2 Praktické zkušenosti

Jelikož program samotný bylo nutné nejprve také nějak překládat, aby mohl přeložit pak i sám sebe, nabízí se jeho srovnání přinejmenším s build managery, pomocí nichž byl

prvně přeložen. Na platformě Windows byl překládán v integrovaném prostředí *Visual Studio .NET 2003*, pro překlad na unixovských systémech byl použit *GNU make*.[36]

Konfigurační soubory *EBM* jsou přizpůsobeny pro kompilaci na platformě Windows i na unixovských systémech a podporují pro každou z těchto dvou variant ještě ladicí a finální podvarianty – celkem tedy mohou vzniknout čtyři různé varianty.

Makefile pro *GNU make* je přiložen ke zdrojovým textům; jedná se o naprosto jednoduchý makefile, jehož jediným účelem je přeložit *EBM* pro účely ladění. Překlad finální verze už obstarává získaný výsledek pomocí vlastního popisu své konfigurace. Vzhledem ke *GNU make* si *EBM* vedl velice dobře. Navzdory tomu, že konfigurační skripty *EBM* včetně všech předdefinovaných pravidel, jejichž zpracování je nutné též započítat, jsou rozsáhlé, nebyla režie jejich zpracování vůbec patrná. Doba běhu programu v případě *EBM* i *GNU make* byla shodná a odvíjela se čistě od nároků spouštěných nástrojů a to jak v případě povolení běhu jediného podprocesu, tak i v případě povolení paralelního běhu více podprocesů.[37]

Situace v případě prostředí *Visual Studia* byla odlišná a dobře ukazuje zmiňované výhody integrovaných prostředí. Na rozdíl od *Visual Studia* nemůže *EBM* činit řadu předpokladů, a je tak více omezen. *Visual Studio* v rámci jednoho projektu porovnává volby pro překlad všech souborů a dle nich je rozděluje do skupin se shodnými nastaveními. Tyto skupiny pak zpracovává jediným spuštěním překladače, čímž snižuje režii na jeho opětovné spouštění a na některé dílčí operace. Zároveň se počítá se sériovým během překladače, což dovoluje sdílet některé pomocné soubory mezi více běhy překladače – jedná se zejména o soubory s ladicími informacemi a o databázi předkompilovaných hlavičkových souborů. *EBM* musí naproti tomu ladicí informace nechat uložit pro každý vstup do jeho vlastního souboru, což může být pomalejší a zvyšuje nároky na diskový prostor, a nemůže používat předkompilované hlavičkové soubory, neboť při paralelním běhu více překladačů by docházelo ke konfliktům nad příslušnými pomocnými soubory a překlad by neuspěl. Vzhledem k těmto rozdílům při zpracovávání vstupu může být *EBM* rychlejší pouze na multiprocesorovém systému, kde dovede na rozdíl od *Visual Studia* těžit ze skutečně paralelního běhu více instancí překladače.

4.2.3 Srovnání

Srovnání vlastností *EBM* s vlastnostmi některých z popisovaných build managerů nabízí v přehledné formě níže uvedená tabulka. Pro srovnání byly vybrány programy *GNU make*, který představuje zřejmě jeden z nejpoužívanějších build managerů, *makepp* jako zástupce pokročilejší větve programů vycházejících z make, a dále systém *Boost Build*, který představuje alternativní přístup k přístupu programů odvozených od make.

[36] Nutno poznamenat, že zde uvedené srovnání není v žádném směru exaktní, ani se o to nepokouší. Aby prezentované závěry měly dostatečnou váhu, bylo by nutné provést testování ve velkém měřítku a na dostatečně rozsáhlém projektu.

[37] Experiment ukázal, že na uniprocesorovém stroji nemá povolení paralelního běhu více podprocesů praktický přínos; na multiprocesoru je zvýšení výkonu již patrné.

GNU make	***makepp***	***Boost Build***	***EBM***
jazyk			
Založený na textové manipulaci a shellových příkazech. Příkazy lze částečně odstínit zabudovanými univerzálními pravidly.	Kromě téhož, co nabízí *make*, dovoluje zápis procedur v jazyce Perl.	Vlastní objektově orientovaný jazyk; pravidla odstíněná od příkazů.	Vlastní procedurálně orientovaný jazyk; pravidla odstíněná od příkazů.
univerzální pravidla			
Založená na pattern matchingu jmen cílů.	Viz *make*.	Implementovaná pomocí obecných procedur.	Viz *Boost Build*.
implicitní závislosti			
Přímo nepodporovány, lze pouze externími nástroji, které upravují vstupní soubory programu.	Vyhledávány moduly se scannery různých jazyků. Program je rozšiřitelný o další moduly.	Vyhledávány zabudovaným programovatelným scannerem na bázi regulárních výrazů.	Lze vyhledat pomocí zabudovaných funkcí i externích programů.
viewpathing			
Globální nastavení proměnnou, direktivami lze nastavit cesty pro skupiny souborů „žolíkovým“ vzorem.	Není k dispozici.	Nastavitelný pro každý cíl zvlášť.	Viz *Boost Build*.
stavovost			
Stav se neukládá.	Ukládají se detailní informace, zejména aktualizační příkazy a signatury souborů.	Viz *make*.	Ukládají se atributy specifikované programátorem, časové značky souborů a graf implicitních závislostí.
detekce neaktuálnosti			
Na základě relativního rozdílu v časech modifikace souborů.	Porovnáním signatury souboru (čas modifikace nebo hash) s hodnotou ve statefile.	Na základě relativního rozdílu v časech modifikace souborů.	Porovnáním času modifikace souboru s hodnotou ve statefile, závisí také na hodnotě uživatelských atributů.
podpora variant			
Žádná.	Dovoluje více variant vedle sebe; parametry variant se určují při spouštění programu.	Dovoluje více variant vedle sebe a jemnou konfiguraci jejich atributů; parametry variant se určují při spouštění programu.	Dovoluje více variant vedle sebe; parametry variant jsou určeny skriptem, při spouštění programu se pouze vybírá z povolených variant.
podpora paralelismu			
Je možné spustit více podprocesů, není podpora synchronizace.	Viz *make*.	Je možné spustit více podprocesů. Synchronizaci mohou zajistit semafory.	Je možné spustit více podprocesů. Synchronizaci zajišťují synchronizační závislosti. Program separuje výstupy podprocesů na konzoli.

5. Závěr

Cílem práce bylo navrhnout a také implementovat build manager, který nebude mít nedostatky typické pro většinu existujících build managerů. Hlavními požadavky zadání byla schopnost paralelní (případně i distribuované) činnosti, výrazného zjednodušení jazyka pro popis konfigurací, ale se zachováním užitečných vlastností stávajících jazyků, a také možnost používat externí programy pro deklaraci závislostí. Cílová implementace měla být nezávislá na operačním systému i hardwarové platformě.

Důležitým bodem práce, na němž je založen návrh řešení, je průzkum existujících implementací a jimi používaných metod a algoritmů. Významná část textu práce je tedy věnována popisu zajímavých rysů několika vybraných implementací, a představuje tak jednak východisko pro popis vlastního návrhu a jednak cenný materiál pro srovnávání jednotlivých přístupů. Spojením vybraných rysů popsaných systémů vznikl zajímavý a neobvyklý návrh, který slibuje naplnění výše zmíněných cílů práce v uspokojivé míře.

Tento návrh byl implementován do podoby plně funkčního programu, který je provozuschopný na širokém spektru platforem. K ověření použitelnosti programu byla vytvořena sada demonstračních konfiguračních skriptů, s jejichž pomocí dovede program sám sebe na všech podporovaných platformách přeložit. Detaily implementace nejsou s ohledem na její rozsáhlost v práci probírány, text práce se věnuje jen principům navrženého řešení a jejich dopadům na implementaci. Implementace samotná, včetně podrobnější dokumentace, je však pro případné zájemce k dispozici na přiloženém disku CD-ROM.

Výsledek práce lze chápat jako experiment, který se snaží najít uspokojivé řešení problémů nastíněných zadáním a ověřit zvolený přístup implementací. Zvolené řešení se dovede s těmito problémy uspokojivě vypořádat, ale podobně jako jiné přístupy má také svá slabá místa a není schopné se vypořádat s některými speciálními případy – náměty na jejich řešení mohou být náplní budoucího vývoje. Prostor je stále otevřen zejména pro řešení distribuovaného zpracování, jehož podpora je daleko obtížnější než podpora zpracování paralelního, a proto není navrženou implementací vůbec řešena.

Možnosti budoucího rozvoje

Pro další rozvoj zvoleného přístupu a programu samotného je klíčové zejména získání dalších zkušeností a jeho praktické vyzkoušení ve větším měřítku. Lze očekávat, že na základě těchto informací se ukáže jako výhodné některé aspekty zvoleného přístupu pozměnit a přizpůsobit tomu i návrh implementace.

Implementace sice podporuje paralelní zpracování, nicméně bylo upuštěno od podpory distribuovaného zpracování vzhledem k veliké náročnosti tohoto úkolu. Dá se předpokládat, že bez podpory distribuovaného operačního systému je řešení tohoto úkolu mimo možnosti jednoduchých nástrojů, jakými jsou samostatné build managery. Pro budoucí vývoj však může být zajímavé ozkoušet chování *EBM* v distribuovaném prostředí a zobecnit jeho návrh tak, aby distribuovaná prostředí lépe podporoval, nebo

aspoň vytipovat zásady pro návrh konfiguračních skriptů, které by měly operovat v distribuovaném prostředí.

Podrobnější průzkum si zaslouží zejména zapojení dalších možností získávání implicitních prerekvizit do návrhu, který je založen na atributovém grafu závislostí. Stávající řešení sice zřejmě je dostačující, a dokonce v některých ohledech lepší než v řadě jiných build managerů, ale je kompromisem mezi návrhem a jeho implementací a nelze jej rozhodně považovat za ideální. Otevřenou otázkou zůstává, jaký by byl praktický rozdíl mezi použitím stávající implementace a nasazením konzervativnějšího přístupu, který by hůře optimalizoval počet některých operací, ale kladl méně předpokladů na své použití. Je dost dobře možné, že by se použití konzervativnějšího přístupu vyplatilo vzhledem ke zvýšení robustnosti algoritmu při snesitelném zvýšení režie – důležité pro takový závěr by bylo ovšem otestování této hypotézy na dostatečně velkém projektu.

Při porovnání chování a výsledků *EBM* a *Visual Studia* se přirozeně naskýtá otázka, jak posílit implementaci samostatného build manageru tak, aby se více svým výkonem a možnostmi přiblížila integrovanému prostředí, aniž by ale ztratila své výhody. V tuto chvíli lze konstatovat, že aspoň zápis konfigurace v *EBM* se svou logikou v mnoha věcech již podobá logice nastavování konfigurace v integrovaných prostředích.

6. Použitá literatura

[1] Baalbergen E. H. (1988): Design and Implementation of Parallel Make. *Computing Systems*, vol. 1, 135–158.

[2] Baalbergen E. H., Verstoep K., Tannenbaum A. S. (1989): On the design of the Amoeba Configuration Manager. *ACM SIGSOFT Software Engineering Notes*, vol. 17, 15–20.

[3] Bubenik R., Zwaenepol W. (1992): Optimistic Make. *IEEE Transactions on Computers*, vol. 41, no. 2, 207–217.

[4] Conradi R., Westfechtel B. (1998): Version Models for Software Configuration Management. *ACM Computing Surveys*, vol. 30, no. 2, 232–282.

[5] Dart S. (1991): Concepts in Configuration Management Systems. *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1–18.

[6] Estublier J. (2000): Software Configuration Management: A Roadmap. *Proceedings of the International Conference of Software Engineering*, 279–289.

[7] Feiler P. H. (1991): Configuration Management Models in Commercial Environments. *Technical report 95-03*, Technical University of Braunschweig.

[8] Feldman S. I. (1979): Make – A Program for Maintaining Computer Programs. *Software Practice and Experience*, vol. 9, no. 3, 255–265.

[9] Fowler G. (1985): The Fourth Generation Make. *Proceedings of the USENIX Summer Conference*.

[10] Fowler G. (1990): A Case for make. *Practice and Experience*, vol. 20, no. S1, 35–46.

[11] Holyer I., Pehlivan H. (2000): An Automatic Make Facility. *Technical Report CSTR-00-001*, Department of Computer Science, University of Bristol.

[12] Lamb D. A. (1990): Relations in Software Manufacture. *Queen's University Department of Computing and Information Science Technical Report 1990*.

[13] Mahler A., Lampen A. (1988): A toolkit for software configuration management. *Proceedings of the Spring 1988 EUUG Conference*, 185–202.

[14] Mahler A., Lampen A. (1988): shape – A Software Configuration Management Tool. *Proceedings of the 1st International Workshop on Software Version and Configuration Control*, 228–243.

[15] Mahler A., Lampen A. (1989): An Integrated Toolset for Engineering Software Configurations. *SIGPLAN Software Engineering Notes*, vol. 13, no. 5, 191–200.

[16] Miller Peter (1997): Recursive Make Considered Harmful. *AUUGN Journal of AUUG Inc.*, vol. 19, no. 1, 14–25.

[17] Schwanke R. W., Cohen E. S., Gluecker R., Hasling W. M., Soni D. A., Wagner M. E. (1989): Configuration Management in BiiN™ SMS. *Proceedings of the 11th International Conference on Software Engineering*, 383–393.

[18] Somogyi Z. (1987): Cake: a fifth generation version of make. *Australian Unix system User Group Newsletter*, vol. 7, no. 6, pages 22–31.

[19] Wingerd L., Seiwald C. (1997): Constructing a Large Product with Jam. *7th International Workshop on Software Configuration Management*.

Přílohy

Příloha A: Ukázky skriptů

Aby si čtenář mohl učinit aspoň přibližnou představu o používání jazyka *EBM* i o tom, jak může vypadat nějaká složitější syntaktická konstrukce, aniž by se musel dívat na přiložený CD-ROM, je v této příloze uvedeno několik příkladů.

Hello, World!

První sadu ukázek tvoří zkrácená verze malého demoprojektu. Tento projekt je v rozšířené variantě k dispozici na přiloženém disku CD-ROM. Projekt obsahuje knihovnu a aplikaci, která tuto knihovnu používá. Adresářová struktura projektu je následující.

🗁 `/`
Kořenový adresář projektu. Obsahuje buildfile; ten kromě platformy apod. určuje také umístění konfiguračního skriptu, který bude zpracován jako první.

🗁 `/.statefiles/`
Do tohoto adresáře budou umístěny statefiles pojmenované podle jednotlivých konfigurací.

🗁 `/build/`
V podadresářích se jmény ve tvaru *konfigurace/podprojekt* budou vytvořeny pomocné soubory, např. *.obj* či *.o* soubory. O toto se postarají už předdefinovaná pravidla. Tento adresář lze také smazat pro odstranění souborů, které nejsou potřeba pro spuštění výsledných programů.

🗁 `/result/`
Do podadresářů se jmény jednotlivých konfigurací bude umístěn výsledek překladu, tedy binární programy (a případně další soubory nezbytné k jejich běhu).

🗁 `/src/`
Zdrojové soubory. Zde je umístěn také soubor *build.ebm*.

🗁 `/src/apps/hello/`
Adresář se zdrojovými soubory programu *hello*; obsahuje konfigurační skript *hello.ebm*.

🗁 `/src/include/`
Sdílené hlavičkové soubory projektů.

🗁 /src/libs/console/
Zdrojové soubory pro knihovnu console; obsahuje konfigurační skript *console.ebm*.

Konfigurační skripty

```
########################################################################
# soubor: build.ebm
#
# Hlavní skript; importuje potřebné nástroje, postará se o společná
# nastavení a zanoří konfigurační skripty podprojektů.
#
# Jména cílů jednotlivých podprojektů:
# "/apps/hello"     Hello, World! (spustitelný program)
# "/libs/console"   utility (statická nebo sdílená knihovna)

# import běžných pravidel
import "defaults.ebm" ;

# nastavení požadovaných nástrojů
Tools.Required = "CC" "linking" ;

# import nástrojů - v případě, že požadované nástroje nejsou
# k dispozici, skript nebude za tímto příkazem pokračovat
import "toolkits.ebm" ;

# deklarace povolených konfigurací (povoleny jsou tímto zápisem např.
# win32.debug.static, ale nikoliv např. win32.debug.release)
configurations win32 linux solaris : debug release : static shared ;

# nastavení defaults všem projektům - zde potřebujeme nastavit cesty
# pro hlavičkové soubory pro překladač jazyka C
Projects.Defaults@CC/IncludeDirs = '$(RootDir)/src/include' ;

# "release" konfigurace si vystačí s implicitními hodnotami, "debug"
# konfigurace potřebuje nastavit některé atributy jinak
config debug {

on $(Projects.Defaults) {
    CC/Debugging    = "on" ;
    CC/Optimize     = "off" ;
    CC/Define       += "_DEBUG" ;
} # on

} # config

# pro "static" konfigurace musíme nastavit, že si přejeme vytvoření
# statických knihoven (implicitní nastavení tvoří dynamické knihovny)
config static {

Projects.Library/Type = "static" ;

} # config
```

```
# zanoříme soubory s jednotlivými podprojekty; příkaz include nám
# dovolí též "namontovat" jednotlivé cíle do hierarchie virtuálních
# jmen - to je význam druhého parametru (jinak detaily viz uživatelská
# příručka)
include "apps/hello/hello.ebm"      : "/apps/hello" ;
include "libs/console/console.ebm"  : "/libs/console" ;

# a ještě připojíme příkazem all hlavní cíl - aplikaci hello -
# k populárnímu cíli all (resp. tedy /all), který bývá defaultním
# cílem
all "/apps/hello" ;


######################################################################
# soubor: hello.ebm

# vytvoří projekt spustitelného programu, přidá do něj soubor hello.c
# a učiní projekt závislým na cíli /libs/console, který může být
# cokoliv - pravidla už se zařídí
Projects.executable "../hello" : "hello.c" : "/libs/console" ;


######################################################################
# soubor: console.ebm

# do pomocné proměnné se uloží jméno cíle knihovny, aby se s ním lépe
# manipulovalo
local target console = "../console" ;

# vytvoříme projekt knihovny podobně jako jsme vytvořili projekt
# aplikace hello
Projects.library $(console) : "colors.c" "input.c" "print.c" ;

# deklarujeme cíle, které představují externí knihovny vyhledávané
# linkerem (to by nebylo ani nutné, ale takto jsou aspoň viditelně
# jejich deklarace a na společném místě)
local target curses     = Files.library/search "/libs/curses" ;
local target ncurses    = Files.library/search "/libs/ncurses" ;

# nastavení jednotlivých konfigurací

config linux {

# přidáme závislost projektu console na knihovně ncurses
Projects.addInputs $(console) : $(ncurses) ;

} # config


config solaris {

# Solaris ncurses nemá, použít můžeme proto pouze curses
Projects.addInputs $(console) : $(curses) ;

} # config


config solaris linux {
```

```
# ať již se užije curses nebo ncurses, necháme pro konfigurace solaris
# i linux nadefinovat další makro
on $(console) {
    CC/Define += "USE_CURSES" ;
}

} # config


config win32 {

config shared {

# pro dynamickou Win32 knihovnu přidáme ještě další vstupní soubory,
# resource script a definiční soubor – nemuseli jsme ani deklarovat
# záměr použít resource compiler: pokud není k dispozici, budou
# pravidla tyto soubory ignorovat, protože k nim nebude existovat
# vhodné odvození
Projects.addFileInputs $(console) : "console.def" "resources.rc" ;

} # config

} # config
```

Ukázka pravidel

Předchozí ukázky demonstrovaly použití jazyka *EBM* běžným uživatelem. Běžné použití využívá jen malou část jazyka – omezuje se vlastně jen na volání pravidel, maximálně na nastavování některých atributů (např. atributů s definicemi maker).

Následující ukázka je tedy podrobně komentovaným úryvkem předdefinovaných demonstračních pravidel. Pravidla jsou rozdělena typicky do dvou částí – první část je tzv. prototyp, který definuje zvyklosti používání pravidel pro celou třídu nástrojů (např. pro překladače jazyka C), druhá část pak je realizace prototypu pro konkrétní nástroj. Díky tomu lze vybírat z několika konkrétních implementací nástroje a navzájem je zaměňovat.

Ukázka tedy zahrnuje dva soubory – prvním souborem je prototyp pro nástroj *bison* a druhým pak realizace tohoto prototypu. Protože *bison* je velmi jednoduchý nástroj a i jeho používání na všech platformách je shodné, prototyp by nebyl ani potřeba – pro demonstrační účely však je vytvořen i pro tento nástroj.

```
#######################################################################
# soubor: proto/bison.ebm
#
# Obecný prototyp pro nástroj bison.

# import pravidel, která řídí relokaci souborů do různých adresářů
import "output.ebm" ;

# definice atributů pro prototyp bisona; zde je velké zjednodušení:
# atribut bude obsahovat přímo potřebné přepínače – u přenositelných
# prototypů je nutné ovšem deklarovat obecnější vlastnosti a pravidla
# nechat tyto vlastnosti překlopit do příslušných přepínačů
attributes {
    string-list Bison/Options ;
}
```

```
namespace Bison {

# následující dvě funkce jsou používány jako detektor typů souborů
function detectType-C ( target tgt ) : string {
    if Paths.getSuffix $(tgt@Name) == ".y" {
        return "true" ;
    }
} # function

function detectType-C++ ( target tgt ) : string {
    if ${ Paths.getSuffix $(tgt@Name) } common ${ ".ypp" ".y++" } {
        return "true" ;
    }
} # function

# pomocná funkce pro společné odvození .y i .ypp (resp. y++) souborů
function /*<private>*/ commonDerive (
    target src      :       # zdrojový soubor
    string type     :       # jeho typ
    string suffix           # sufix výsledku
) {
    # vytvoření odvozeného cíle - zdrojového souboru pro překladač
    local target result = Targets.createBlank
        ${ Paths.replaceSuffix $(src) : $(suffix) }
        : "created" ;

    # nastavíme typ a další atributy novému cíli
    on $(result) {
        Type = $(type) ;
        Project = $(src@Project) ;
        Name = Paths.replaceSuffix $(src@Name) : $(suffix) ;
        DependsOn += $(result) ;
    }

    # bisonímu vstupu pak opravíme atributy
    on $(src) {
        Outputs += $(result) ;
        DependsOn += $(result) ;
        Bison/Options ?= $(src@Project@Bison/Options) ;
    }

    # vygenerovaný soubor necháme vytvořit v adresáři pro pomocné
    # soubory - užijeme k tomu opět dalších předdefinovaných pravidel
    # (pro vysvětlení: "intermediate" je kategorií souboru, od níž
    # se odvíjí adresář - adresáře nejsou v pravidlech uvedeny pevně)
    Output.apply $(result) : "intermediate" ;
} # function

# vlastní odvozovací pravidla - ta jsou již jednoduchá
function derive-C ( target src ) {
    commonDerive $(src) : "source/C" : ".c" ;
}

function derive-C++ ( target src ) {
    commonDerive $(src) : "source/C++" : ".cpp" ;
}

} # namespace
```

```
# registrace typů souborů
type "source/bison"      : $(Bison.detectType-C) ;
type "source/bison++"    : $(Bison.detectType-C++) ;


# tohle už je jen doplněk konvencí demonstračních pravidel; pokud by
# uživatel chtěl některý soubor explicitně deklarovat jako bisoní
# gramatiku, může k tomu použít tyto funkce místo přímého nastavování
# atributů (použití deklarující funkce je ostatně i bezpečnější)
namespace Files {

function source/bison (
    string-list paths :
    string-list flags
) : targets {
    local targets result = Targets.createByFile $(paths) : $(flags) ;
    result@Type = "source/bison" ;
    return $(result) ;
} # function

function source/bison++ (
    string-list paths :
    string-list flags
) : targets {
    local targets result = Targets.createByFile $(paths) : $(flags) ;
    result@Type = "source/bison++" ;
    return $(result) ;
} # function


} # namespace


###############################################################################
# soubor: toolkit/bison.ebm
#
# Implementace prototypu nástroje bison.

# import základních pravidel
import "defaults.ebm" ;
# import prototypu
import "proto/bison.ebm" ;

# kontrola, zda se již některá implementace prototypu
# nezaregistrovala, to bychom nepokračovali
if "bison" in $(Tools.Available) {
    return ;
}

# vypíší se použité a dostupné nástroje
System.stdout   "Using toolkit 'bison'"
                "Tools available: 'bison'";

# instalace toolkitu
Toolkits.Available += "bison" ;
# instalace dostupných nástrojů
Tools.Available += "bison" ;
Tools.Installed += "bison" ;
```

```
namespace BisonTool {

# prvně zjistíme příkaz, kterým se bison pouští (default je "bison")
global string-list Command = Tools.queryCommand "bison" : "bison" ;


# odvozovací pravidla: zavolají prototypovou implementaci a doplní
# svá specifická nastavení
function derive-C ( target src ) {
    Bison.derive-C $(src) ;

    on $(src) {
        Action ?= $(compile) ;
        Action/Options = $(src@Bison/Options) ;
    }
} # function

function derive-C++ ( target src ) {
    Bison.derive-C++ $(src) ;

    on $(src) {
        Action ?= $(compile) ;
        Action/Options = $(src@Bison/Options) ;
    }
} # function


# akce se budou lišit ve verzích pro Unix a pro Windows
if $(System.OS/Name) == "Windows" {

function compile ( target src ) : string {
    local target tgt = $(src@Outputs) ;

    # vytvoříme adresář, kam má být umístěn výsledek
    FileSystem.createDirs ${ Paths.getDir $(tgt@LogicalPath) } ;

    # zařadíme do fronty příkazy akce, které se mají vykonat;
    # parametrem je příkazová řádka
    Action.queueCommand $(Command)
        $(src@Action/Options)   # options
        "-o" $(tgt@NativePath)  # výstupní soubor
        $(src@NativePath) ;     # zdrojový soubor bisonu

    # ukončíme akci potvrzením platnosti zadaných příkazů; ty
    # budou nyní vykonány
    return "update" ;
} # function

} # if
else {

# situace pro unixovské systémy bude analogická, jen tu ještě
# zařídíme odstranění CR znaků ze zdrojového souboru, protože
# některým verzím bisona dělají potíže
```

```
function compile ( target src ) : string {
    local target tgt = $(src@Outputs) ;
    # získáme logickou cestu k vygenerovanému souboru
    local string tgtPath = Paths.getDir $(tgt@LogicalPath) ;
    # vytvoříme potřebný adresář
    FileSystem.createDirs $(tgtPath) ;
    # a připavíme si jméno pro pomocný mezisoubor
    tgtPath = Paths.toNative '$(tgtPath)/$(src@Name)' ;

    # abychom neměli následující výraz příliš dlouhý, spojíme
    # si řetězce do pomocné proměnné – následující výraz každý
    # řetězec v seznamu v atributu Action/Options cíle src uzavře
    # do apostrofů, aby jej shell neinterpretoval
    local string options = ${
            string opt in $(src@Action/Options)
            ; '\'$(opt)\''
        } ;

    # zde stojí za povšimnutí několik věcí:
    # * necháme provést shellovský skript
    # * na shellu se ponechá zřetězení příkazů
    # * je tu použito přerušení literálu tildami – a přesto je řetězec
    #   pěkně zarovnaný
    Action.queueScript
        'tr -d "\\015" < \'$(src@NativePath)\' > \'$(tgtPath)\''
        '\'$(Command)\' $(options) \'-o$(tgt@NativePath)\' ~
        ~ \'$(tgtPath)\'' ;

    return "update" ;
} # function

} # else

} # namespace

# zaregistrujeme odvozovací pravidla
derivation "source/bison"   : $(BisonTool.derive-C) ;
derivation "source/bison++" : $(BisonTool.derive-C++) ;
```

Příloha B: CD-ROM

Nedílnou součástí práce je příloha v podobě disku CD-ROM. Na přiloženém disku je vedle veškerých zdrojových textů implementace k dispozici i příručka pro uživatele. Instrukce pro práci s diskem jsou obsaženy v souboru *readme.txt*, který je v kořenovém adresáři.