Achtergrond

De Wet van Moore

In 1965 deed Gordon E. Moore zijn beroemde voorspelling dat het aantal transistoren per chip elke twee jaar zou verdubbelen. Inmiddels is deze voorspelling beter bekend als de wet van Moore, en met recht. Want al een dikke veertig jaar gaat de wet prima op. De transistoren worden steeds kleiner, waardoor er steeds meer op hetzelfde oppervlak passen. En de uitdaging die het aanhouden van de wet van Moore biedt voor de chips-fabrikanten, zorgt ervoor dat de exponentiële groei steeds weer gehandhaafd blijft. En dat betekent op zijn beurt weer dat de rekenkracht van computers steeds groter wordt. Sinds enkele jaren echter, wordt er sterk getwijfeld aan de mogelijkheid om de wet nog langer te handhaven. De fysieke grenzen komen rap dichterbij, en zijn in het geval van de elektriciteit/warmte eigenlijk al bereikt.

De transistoren zijn namelijk niet 100% efficiënt. Net zoals in alle andere elektrische apparaten, wordt een deel van de elektriciteit door een transistor in warmte omgezet. Nu is dat op zichzelf niet zo'n probleem, maar als je veel warmtebronnen bij elkaar zet wordt het al gauw een klein kacheltje. Met een ventilator valt dan nog wel heel wat warmte weg te nemen, maar er kunnen inmiddels zodanig veel transistoren op een chip geplaatst worden dat een ventilator eigenlijk geen toereikende oplossing meer biedt. Daarnaast wordt ook nog eens zodanig veel energie getrokken door de vele transistoren dat er problemen ontstaan voor de mobiele technologie, vanwege de altijd beperkte energievoorraad. Maar ook milieutechnische redenen baren inmiddels zorgen, zeker in het huidige klimaat[1][2].

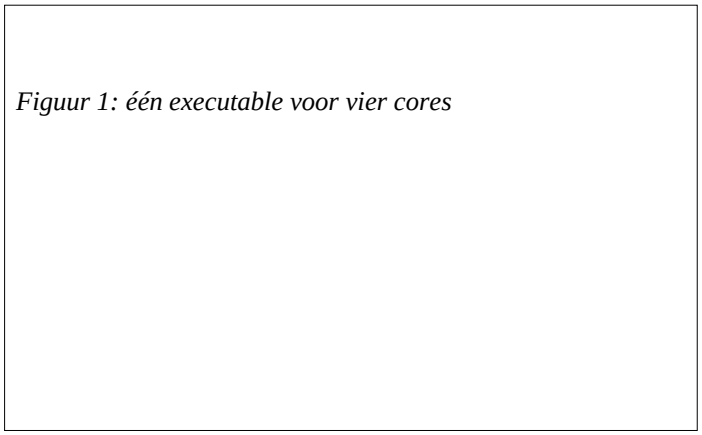
Desalniettemin, de computer industrie is sterk afhankelijk van de wet van Moore. Computers dienen steeds sneller te worden. Steeds meer instructies per tijdseenheid moeten worden uitgevoerd. En die eis werd en wordt met name behaald met behulp van de steeds kleiner wordende transistoren. Nu de voorspelling van Moore steeds moeilijker te halen lijkt, zoekt de industrie juist steeds harder naar alternatieve methoden om de groei in stand te houden. Een van de belangrijkste oplossingen die daarvoor ingezet wordt, is het plaatsen van meerdere processoren per computer. Theoretisch gezien zouden twee processoren twee keer zo veel werk moeten kunnen doen als een enkele processor. Daarmee zou de prestatie van de computer dus net zo veel moeten toenemen als dat er extra processoren worden ingezet. In de praktijk valt dit jammer genoeg nogal tegen. Het efficiënt verdelen van de werkbelasting in een dergelijke processor architectuur, ook wel multi-core, blijkt erg lastig. Hierdoor wordt er niet optimaal gebruik gemaakt van de beschikbare rekenkracht en vallen de prestaties dus tegen.

Multithreaded en Compaan

Het grote probleem bij het verdelen van de werkbelasting zit bij de software. Deze is over het algemeen niet geschreven met meerdere processoren in gedachten maar juist met een enkele processor, en dat zal ook niet gauw gaan veranderen. De mens is namelijk niet goed ingesteld om in termen van meerdere naast elkaar lopende processen te denken. En de mens zal toch, zo mogen we wel verwachten, in de vorm van programmeurs de komende tijd verantwoordelijk blijven voor het schrijven van die software.

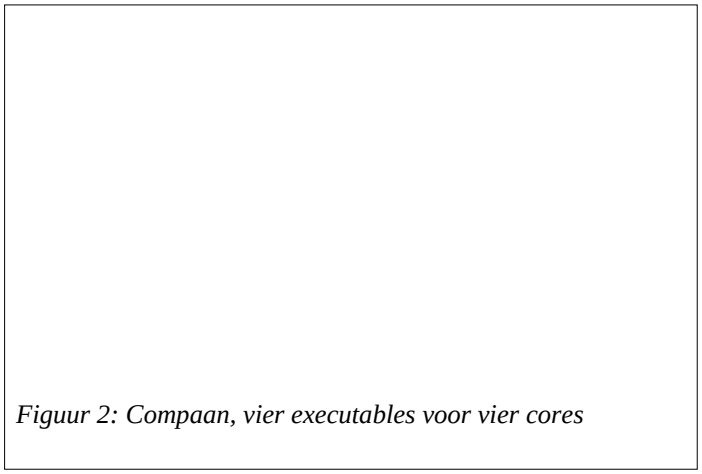
Hoe zorg je er dan toch voor dat een programma, geschreven voor een enkele processor, zo goed mogelijk gebruik gaat maken van de nieuwe processor-architectuur? Het ligt voor de hand om toch weer naar de computer te kijken voor een goede omzettings-strategie, deze heeft namelijk geen enkel probleem met het denken in termen van meerdere processen.

De meeste omzettings-strategieën zoeken in de geschreven code, naar onderdelen die onafhankelijk van de rest uitgevoerd zouden kunnen worden, en geeft bij die stukken aan dat deze in een aparte thread gedraaid kunnen gaan worden. Het resultaat is een enkele executable, met onderdelen daarin die eventueel op een andere processor gedraaid kunnen worden. Of dit ook gebeurd en hoe is dan aan het besturingssysteem.



Figuur 1: één executable voor vier cores

Voor Compaan, dat aan de Universiteit Leiden ontwikkeld wordt, wordt een andere filosofie gevolgd. Compaan verdeeld het programma op in net zo veel onderdelen als er processoren voor handen zijn. En probeert als er verschillende typen processoren beschikbaar zijn daar in de verdeling rekening mee te houden. Zo krijg je een hoeveelheid executables net zo groot als er processoren zijn, welke elk onafhankelijk van elkaar zullen gaan draaien op het type processor dat daar het meest geschikt voor is.



Figuur 2: Compaan, vier executables voor vier cores

Eclipse en Compaan

Het schrijven, aanpassen en compileren van een project wordt bij voorkeur gedaan met behulp van een Integrated Development Environment (IDE), omdat deze een hoge mate van

assistentie verlenen aan de programmeur bij zijn taken. Een project met meerdere executables, zoals er door Compaan geproduceerd wordt, is echter iets waar in de IDE's van deze tijd nog geen rekening mee is gehouden. Een van de meest geliefde IDE's is Eclipse, en ook deze is niet berekend op het concept van meerdere executables per project. Maar er is een lichtpuntje aan de horizon. Eclipse is namelijk een open source product, en is daarmee gratis en vrij aan te passen. Maar het wordt nog beter, Eclipse biedt de mogelijkheid om je eigen plug-ins te definiëren. Met deze plug-ins kan een persoonlijke Eclipse gemaakt worden met de mogelijkheden die de gebruiker graag ziet. Doel van dit bachelor project is om een plug-in te schrijven waarmee Eclipse voortaan wel met multi-core projecten volgens Compaan om kan gaan.

Probleem

Eclipse en CDT

De meeste programmeurs vinden het prettig om met een Integrated Development Environment te werken. Met name omdat ze hiervan assistentie krijgen bij het programmeren en debuggen, en ze dus productiever kunnen zijn. Dat meer productiviteit leidt tot meer geluk bij het management heeft er alleen nog maar meer toe bijgedragen dat IDE's een grote hit zijn.

Een grote speler in de JAVA-wereld is de Eclipse IDE. Het is niet alleen een open source product en dus gratis, maar het biedt ook nog eens kwaliteit. Naast een oersterke basis, geeft het aan de altijd kritische en veeleisende programmeur, de mogelijkheid om het product aan te passen naar zijn wensen door middel van plug-ins. Het gaat hier niet om cosmetische aanpassingen, alhoewel dat ook tot de mogelijkheden behoort, maar om significante aanpassingen van het geheel. Eclipse is op deze manier enorm flexibel en lijkt daarmee in staat om elk probleem op te lossen.

Een aantal jaren geleden is er dan ook begonnen aan een project om Eclipse geschikt te maken voor de inherent complexere talen C en C++. Dit project heet C/C++ Development Tooling, ofwel CDT, en het spreekt voor zich dat het hier gaat om een plug-in. Met deze plug-in streven ze naar eenzelfde kwaliteit en flexibiliteit als de JAVA evenknie.

Eclipse CDT en Compaan?

Eclipse gaat, net zoals de concurrentie, uit van het standaard concept van: één project, één main functie, één executable, één processor. Compaan ziet dat echter graag anders, deze verdeeld één project over X main functies, en X executables. Waarbij X het aantal processoren is. Eclipse is, zoals hierboven uiteengezet, enorm flexibel. Maar kan er ook een dergelijke significante verandering aangebracht worden met behulp van een relatief simpele plug-in?Jammer genoeg is dit niet het enige vraagteken.

Compaan projecten zullen geschreven worden in C/C++. Dit betekent dat om met deze taal om te kunnen gaan de CDT plug-in een vereiste zal zijn om tot een goede oplossing te komen. En hierin zit hem de crux. De CDT is een uitstekend product, maar het is ook een project dat nog sterk in beweging is. De structuur staat nog allerminst vast, en functionaliteit mist. Er moet dan ook rekening mee worden gehouden dat als Eclipse wel in staat is om een opzet met meerdere executables te ondersteunen, het niet noodzakelijkerwijs ook betekent dat CDT dat ook (al) doet.

Oplossing – Concept

Huidige Situatie

De Eclipse CDT maakt gebruik van makefiles om zijn projecten te compileren. Met dat gegeven kan je kiezen uit twee soorten C/C++ projecten, de 'standard makefile project' waarin de gebruiker geacht wordt zelf voor de makefiles te zorgen en die te onderhouden, en daarnaast de 'managed makefile project' waarin Eclipse voor het aanmaken en onderhouden zorgt. Voor dit bachelor project is het de bedoeling dat er voortaan de mogelijkheid is om aan een multi-core toepassing te werken zonder te veel omkijken te hebben naar het compilatieproces. Het ligt dan ook voor de hand om te kiezen voor een uitbreiding of een aanpassing van het huidige 'managed makefile' systeem. Om tot een goede oplossing te komen zal het product van dit systeem aangepast moeten worden, want de makefile structuur die de CDT hanteert is robuust maar jammer genoeg ook onhandig, zeker voor de taak die het nu gesteld wordt.

Iets wat direct op valt aan de huidige structuur is het herdefiniëren van de build-regels in elke makefile, waar het veel logischer zou zijn om deze op een enkele plek te definiëren. Als je dit probeert aan te passen loop je al gauw tegen het probleem aan dat elke makefile in het hele project in de makefile op het project-niveau wordt meegenomen. Het gevolg is foutmeldingen van, ironisch genoeg, het herdefiniëren van regels. De laatste druppel vormt het besef dat er door deze opzet al bij kleinere projecten een gigantische virtuele makefile wordt gecreëerd op het project-niveau, wat de snelheid niet ten goede kan komen. Al met al is de huidige opzet intuïtief en eenvoudig, maar is niet flexibel genoeg om aan de eisen van dit project te kunnen voldoen.

```
# Each subdirectory must supply rules for building sources it contributes
source/%.o: ../source/%.cpp
	@echo 'Building file: $<'
	@echo 'Invoking: Cygwin C++ Compiler'
	g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"
	@echo 'Finished building: $<'
	@echo ' '

source/%.o: ../source/%.cc
	@echo 'Building file: $<'
	@echo 'Invoking: Cygwin C++ Compiler'
	g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"$(@:%.o=%.d)" -MT"$(@:%.o=%.d)" -o"$@" "$<"
	@echo 'Finished building: $<'
	@echo ' '
```

Figuur 3: Twee build-regels in de CDT makefile structuur

Getrapte Makefile Structuur

Het mag duidelijk zijn dat de huidige structuur niet voldoet, en niet makkelijk aangepast kan worden aan de eisen van dit project. Een betere, en flexibelere structuur is de volgende. Eén 'default.mk' wordt gedefinieerd op het project-niveau, deze wordt door elke makefile in het hele project bijgevoegd. In dit bestand worden alle platform onafhankelijke variabelen en regels gezet, wat betekent dat de build- en link-regels eenmalig worden aangemaakt en bij elkaar op één plek staan. Platform afhankelijke variabelen worden op hetzelfde niveau in 'vars.mk' gezet en meegenomen in 'default.mk'. Het voordeel hiervan is dat als een ander

teamlid die op een ander platform werkt, bijvoorbeeld Linux in plaats van Windows, aan het project wil werken, hoeft deze alleen de 'vars.mk' aan te passen of te vervangen om met het project aan de slag te kunnen.

Dan blijft voor de makefiles maar een kleine taak over; het bijhouden van de lijsten van subdirectories en source-files die te zien zijn op hun eigen niveau. Alleen de makefile op het project-niveau krijgt een extra taak mee. Deze moet zorgen voor een nette enkele ingang voor de buitenwereld en daarom worden in dit bestand de 'all'- en 'clean'-regel neergezet, voor het respectievelijk bouwen en schoonmaken van het project. Er mist nu nog een onderdeel, en dat is het stappen van makefile naar makefile. Daarvoor wordt in 'default.mk' een speciaal stukje code geschreven die direct in de onderliggende shell uitgevoerd wordt, welke een 'depth-first' doorloop van de directory-structuur maakt. Hierdoor kan op de juiste plek de build- en link-regels aangeroepen worden met de juiste bestanden. Deze stap-methode vervangt de biuld-regels in de makefiles van de huidige structuur en de bijbehorende kolossale project-makefile, en zorgt daarmee voor een flexibel en lichtgewicht systeem.

```
package: $(CLASS)
	@if [ "x$(SUBDIRS)" != "x" ]; then \
	set $(SUBDIRS);\
	for x do \
		if [ -r $$x ]; then \
		( cd $$x; \
			make package; \
			make archive; \
		) \
		fi; \
	done; \
	fi;
```

Figuur 4: Het Shell-script stap-algoritme

Net als de CDT makefile structuur is ook deze structuur bedacht op een enkele executable per project, en is als zodanig dus ook niet geschikt voor dit bachelor project. In tegenstelling tot de huidige structuur is deze manier van makefiles opbouwen wel flexibel genoeg om aangepast te worden tot een structuur die het Compaan multi-core systeem wel aan kan. Deze getrapte makefile structuur zal dan ook de basis vormen voor de concept oplossing.

Muli-core Makefile Structuur

Het grote verschil dat Compaan heeft met de traditionele opzet, is dat er nu niet een executable per project moet komen maar een executable per subdirectory van een project. Om dit te bewerkstelligen zou je kunnen zeggen dat de getrapte structuur simpelweg een niveau opgeschoven moet worden, en dat is eigenlijk ook zo, maar er zitten natuurlijk wel wat haken en ogen aan. Als we dit niveau, en dan specifiek de directories, direct onder het project voortaan core-folders noemen, dan veranderd er vrij weinig vanaf dit core-folder niveau. De makefiles blijven lijsten van directories en source-files, de 'default.mk' bevat nog steeds de platform-onafhankelijke regels en variabelen en de 'vars.mk' specificeert de variabelen die platform-afhankelijk zijn, of beter core-afhankelijk. Het enige echte verschil is dat de 'all'- en 'clean'-regel niet mee verhuizen naar de core-makefile.

```
COMPILER = g++

LINKER = g++

COMPILEFLAGS = -I $(ROOT)/..

LINKFLAGS =
```

Figuur 5: Een typisch voorbeeld van een vars.mk

```
$(COBJS): %.o: $(DIRPATH)%.c
	$(COMPILER) -c $(COMPILEFLAGS) $(CFLAGS) \
	$(DIRPATH)$(patsubst %.o,%.c,$@) -o $@

executable:
	$(LINKER) $(LINKFLAGS) $(CLASS) $(ARCHIVES) $(HEADER) -o \
	$(EXECUTABLE).exe
```

Figuur 6: Een build-regel en de linkregel uit een default.mk

```
ROOT = ..

MYPATH = source

THIS = source

EXECUTABLE = $(THIS)

SUBDIRS = newsource

CCSRCS = Main.cc

CPPSRCS = Klasse.cpp

HEADERS = Klasse.h

include $(ROOT)/source/default.mk
```

Figuur 7: Een core-makefile, een subfolder-makefile mist de EXECUTABLE variabele

De veranderingen zitten in het project-niveau, ook hier vinden we een 'vars.mk', een 'default.mk' en een makefile, waarbij de makefile de 'all'- en 'clean'-regel behouden heeft om de enkele ingang te bewaren. Alleen de 'default.mk' heeft nu wat veranderingen ondergaan om het stappen van core-folder naar corefolder mogelijk te maken voordat de build of link fase ingezet wordt. Belangrijk om te beseffen bij deze structuur is dat binnen de core-folder steeds de eigen 'default.mk' en 'vars.mk' gebruikt zullen worden, net als dat ook op het project-niveau de eigen 'default.mk' en 'vars.mk' worden gebruikt.

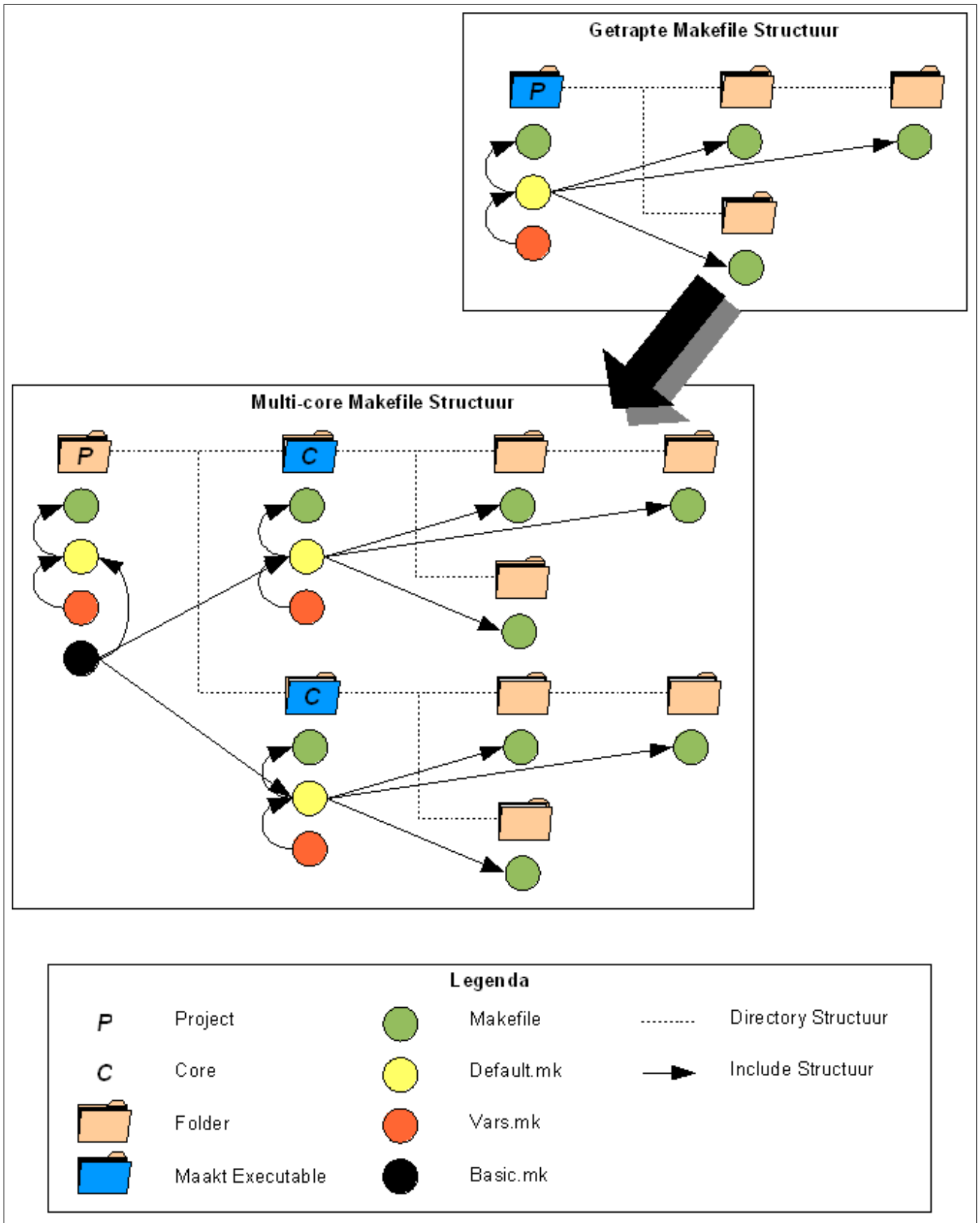
```
all : printstage1 visitcores printstage2 compile

clean : remove all
```

Figuur 8: De all- en clean-regel uit de project-makefile. De all instructie in de clean-regel heeft te maken met een vermoedelijke bug in de CDT.

```
package: $(CLASS)
	@if [ "x$(SUBDIRS)" != "x" ]; then \
	set $(SUBDIRS);\
	for x do \
		if [ -r $$x ]; then \
		( cd $$x; \
			make package; \
			make archive; \
		) \
		fi; \
	done; \
	fi;

visitcores:
	@if [ "x$(SUBDIRS)" != "x" ]; then \
	set $(SUBDIRS);\
	for x do \
		if [ -r $$x ]; then \
		( cd $$x; \
			make package; \
		) \
		fi; \
	done; \
	fi;
```

Figuur 9: Een kleine aanpassing, en het Multi-core stap-algoritme is geboren, uit basic.mk.

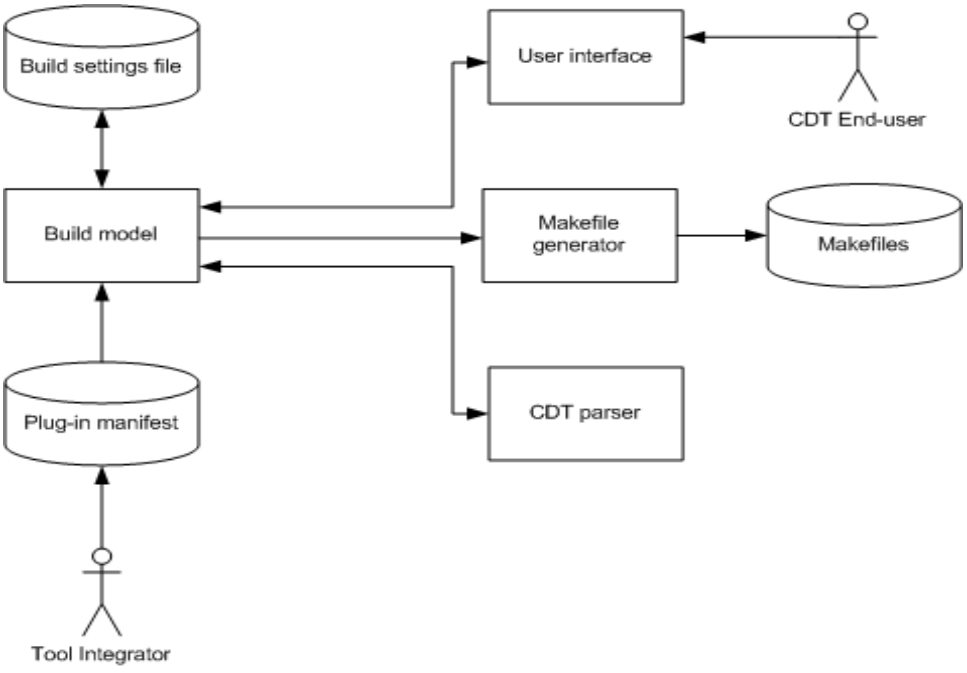
Omdat in elke core-folder en het project de eigen 'default.mk' gebruikt wordt om de eigen 'vars.mk' juist te gebruiken, ontkom je niet aan een bepaalde hoeveelheid duplicatie. Maar om dat aanzienlijk terug te dringen heeft de multi-core makefile structuur nog een aanpassing in petto. Alle regels en variabelen die onveranderlijk zijn, ook met tussenkomst van de variabelen uit 'vars.mk', worden uit de 'default.mk' gehaald en op het project-niveau in 'basic.mk' gezet, welke natuurlijk in elke 'default.mk' wordt meegenomen. Zodoende ontstaat een flexibele structuur, met een minimum aan duplicatie, die per core-folder een executable zal genereren.



Figuur 10: De toepassing van de getrapte makefile structuur in de multi-core plug-in

Oplossing – Uitwerking

Managed Build System

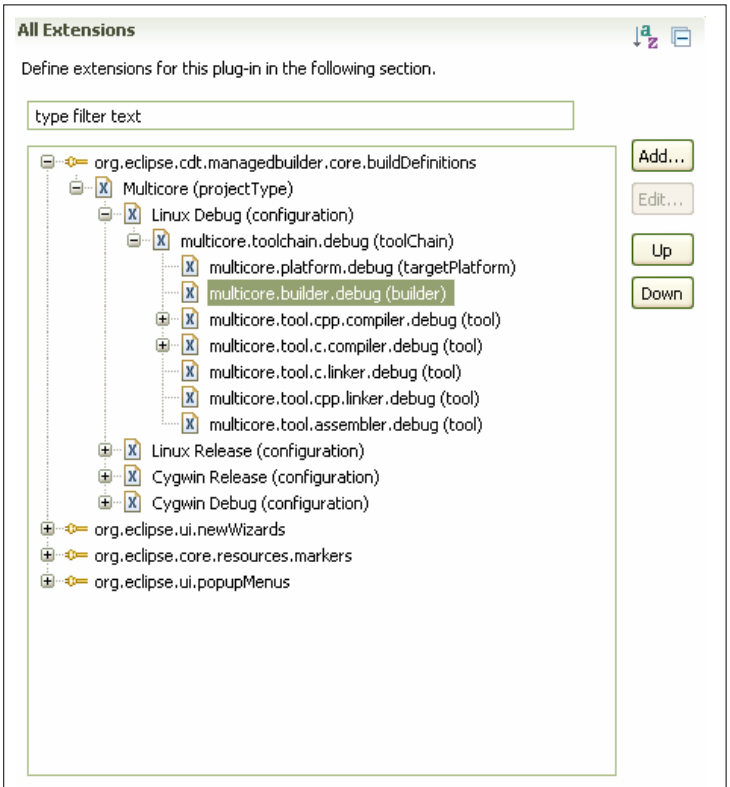


Figuur 11: Managed Build System opbouw [3]

Zoals al eerder genoemd kan er in de Eclipse CDT gekozen worden uit twee typen projecten, 'managed makefile' en 'standard makefile'. Beide typen worden beheerd door de 'Managed Build System", of MBS, welke meerdere zogeheten 'extension points' bevat. Deze 'extension points' bieden plug-in schrijvers de mogelijkheid om op die plaats eigen functionaliteit toe te voegen. Voor dit project is dat dan ook de uitgekozen plek om de multi-core plug-in aan Eclipse aan te sluiten.

Zoals te zien in figuur x, heeft de MBS 4 hoofdcomponenten, waaronder ook een 'Makefile generator'. Intern zit deze 'Makefile generator' gekoppeld aan een builder. Een builder is een standaard systeem binnen Eclipse om onderdelen op de hoogte te stellen van veranderingen, en deze aan het werk te zetten. Zo ook in dit geval een 'Makefile generator'. Maar om een eigen builder te kunnen aanmaken, moest er eerst respectievelijk een eigen projecttype, configuratie, toolchain en alle tools, waaronder de gezochte builder, aangemaakt worden.

Eenmaal beschikkend over een builder, kan er de eigenschap 'buildfileGenerator' worden ingevuld. Dit moet een JAVA-klasse zijn die de interface IManagedBuilderMakefileGenerator implementeerd. Omdat er voor dit project niet zo heel veel van de interne structuur aangepast hoeft te worden, voldoet de multi-core plug-in hieraan door simpelweg de standaard GNUMakefileGenerator, welke door de CDT wordt gebruikt, over te erven, en zodanig veel functionaliteit te hergebruiken.



Figuur 12: De multi-core uitbreiding op de MBS

De implementatie

Figuur 13: Klasse Relatie Diagram van de Multi-core Plug-in

Zoals je in figuur x kunt zien is de Makefile Generatie voor de multi-core plug-in in meerdere objecten onderverdeeld. Dit overigens in tegenstelling tot de CDT makefile generatie die daardoor erg onoverzichtelijk is. In deze paragraaf lopen we alle onderdelen van de Makefile Generatie kort door.

MYMakefileGenerator

MyMakefileGenerator is een direct kind van GNUMakefileGenerator, de klasse die door de CDT gebruikt wordt om makefiles aan te maken. Hierdoor kunnen er veel functionaliteiten hergebruikt worden.

Een van die functionaliteiten is de ResourceProxyVisitor. Deze loopt door de bestandsstructuur heen en bepaald of een bestand een source-file is. Als dat het geval is dan voegt deze de directory waarin het bestand staat, samen met alle directories daarboven toe aan een lijst met interessante directories. Met die lijst kan de MakefileGenerator dan weer effectief aan de slag, om alleen waar nodig makefiles aan te maken.

```
private boolean buildStructure(MultiStatus status) throws CoreException {
	IFolder topDir = createDirectory(_config.getName());
	IPath topBuildDirectory = topDir.getProjectRelativePath();

	// basic.mk
	if (!addBasic(status, topDir)) {
		return false;
	}

	// default.mk
	if (!addDefault(status, _project, topDir)) {
		return false;
	}

	MakefileBuilder mfBuilder = new MakefileBuilder();

	// project makefile
	mfBuilder.addToProject(_project, topDir, getSubdirList(), _monitor);

	for(IContainer folder: getSubdirList()) {
		IFolder mirror =  createDirectory(topBuildDirectory.append(
					folder.getProjectRelativePath()).toString());
		if (folder.getProjectRelativePath().segmentCount() == 1) {
			// core folder dus nog een default.mk
			if(!addDefault(status, folder, mirror)) {
				return false;
			}
			// core makefile
			mfBuilder.addToCore(folder, mirror, getSubdirList(), _monitor);
		} else {
			// standard makefile
			mfBuilder.addToFolder(folder, mirror, getSubdirList(), _monitor);
		}
	}
	return true;
}
```

Figuur 14: buildStructure, het werkpaard van MyMakefileGenerator.

Een andere functionaliteit is de ResourceDeltaVisitor. Deze maakt lijstjes van toegevoegde en verwijderde bestanden. Om zo effectiever de makefiles te kunnen aanpassen. Deze zijn handig voor het geval een 'incremental build' wordt gedaan, in tegenstelling tot een 'full build'. Bij een 'full build' worden zonder omkijken alle bestanden opnieuw gecompileerd, een 'incremental build' doet dat slimmer door alleen de aangepaste bestanden opnieuw te compileren, waar de lijstjes opgebouwd in de ResourceDeltaVisitor goed van pas komen. Dit resulteert in een complexer, maar wel sneller systeem doordat steeds kleine beetjes van het werk worden verricht. Op het moment van schrijven wordt deze functionaliteit nog niet ingezet in de multi-core plug-in.

Het belangrijkste wat MyMakefileGenerator dan nog rest is het opbouwen van de makefilestructuur. Dit wordt gedaan in de methoden generateMakefiles en regenerateMakefiles. Deze tweedeling wordt gebruikt om onderscheid te maken tussen de incremental en full build. Er wordt een 'regenerate' aangeroepen bij een 'full build', en een 'generate' bij een 'incremental build', maar deze laatste verwijst in de multi-core plug-in nog slechts naar zijn broer door.

Om het project netjes te houden wordt de directory-structuur volledig gekopieerd in een daarvoor bestemde folder, de configuratie-folder. Hierbinnen kan vervolgens naar hartelust een makefile structuur opgebouwd worden. Vreemd genoeg is een van de taken van een makefile generator het opbouwen van deze kopie. Daarentegen kan de makefile structuur wel tegelijk met de kopie opgebouwd worden en scheelt dat uiteindelijk een mogelijk kostbare doorloop.

DefaultFile

Voor het opbouwen van de multi-core makefile structuur wordt gebruik gemaakt van een aantal verschillende hulp-klassen. Een daarvan is DefaultFile, die verantwoordelijk is voor de 'default.mk' en 'basic.mk' bestanden. Voor elk type bestand heeft DefaultFile een methode, 'getDefaultMake' en 'getBasicMake'.

'getBasicMake()' kopieert het 'basic.mk' bestand uit de plug-in naar de gewenste plaats in de opgebouwde directory-kopie. 'getDefaultMake()' doet grotendeels hetzelfde, maar bekijkt daarvoor eerst wat voor varsfile in de include regel moet staan. Hiervoor wordt een markering gebruikt op de originele folder die aangeeft welke varsfile voor die betreffende folder gewenst is. Dit systeem is een veelgebruikt handigheidje in Eclipse, welke vooral bekend is van de compiler errors en warnings die op, en in, elk bestand met een klein icoontje aangeven waar het probleem zich voordoet[4]. Doordat de markering op de originele folders wordt gemaakt, blijft de instelling behouden ook al wordt de hele configuratie-map verwijdert.

```
private static String getVars(IContainer original) throws CoreException {
	IMarker[] markers = original.findMarkers("multicore.coremarker", false,
	IResource.DEPTH_ZERO);
	if (markers.length > 0) {
		return (String) markers[0].getAttribute("vars");
	} else {
		return "vars.mk";
	}
}
```

Figuur 15: methode in DefaultFile naar aanleiding van de marker de juiste vars.mk bepaald.

VarsRegister

De markeringen die 'getDefaultMake()' gebruikt verwijzen naar bestanden die 'VarsRegister' aanmaakt met de methode 'createVarsFile()'. Deze worden in een aparte map geplaatst 'VARS_DIR' die aangemaakt wordt door de methode 'createVarsDir()' uit deze klasse.

Dit is een lichte aanpassing van het concept, met name omdat de gebruiker zelf nieuwe vars.mk's moet kunnen definiëren en deze niet verloren mogen gaan bij een schoonmaak

actie. In het project zelf staan ze daarvoor veilig. Het enige merkbare verschil met het concept, is de extra tekst in de include-regel van elke default.mk, maar in essentie blijft het hetzelfde.

De core-folders worden, tenzij anders aangegeven door de gebruiker, altijd gemarkeerd met de standaard “vars.mk” eigenschap. Wat betekent dat in elke default.mk standaard “vars.mk” wordt meegenomen, tenzij anders aangevraagd. Het bestand “vars.mk” wordt daarom ook standaard aangemaakt in de VARS_DIR-folder, en wordt bij verwijdering of hernoeming opnieuw aangemaakt.

MakefileBuilder

De klasse 'MakefileBuilder' produceert de verschillende normale makefiles, en biedt daarvoor drie methoden aan: 'addToCore()', 'addToFolder()' en 'addToProject()'. Hierbij wordt een 'MakefileBlock' variabele aangemaakt en voor het betreffende type makefile klaargezet. Deze variabele wordt doorgegeven aan 'addMakefile()' die de makefile vervolgens aanmaakt.

```
private void addMakefile(MakefileBlocks blocks) throws CoreException {
	StringBuilder result = new StringBuilder();
	result.append(blocks.getStart());
	result.append(blocks.getExecutable());
	result.append(blocks.addMembers());
	result.append(blocks.getRules());
	result.append(blocks.getDefault());

	ByteArrayInputStream content = new ByteArrayInputStream(result.toString().getBytes());
	createMakefile(blocks, content);
}
```

Figuur 16: Het proces voor de aanmaak van een makefile

MakefileBlocks

Deze klasse heeft een sterk samenwerkingsverband met 'MakefileBuilder'. Waar 'MakefileBuilder' de juiste volgorde van de aanroepen maakt, zorgt deze klasse ervoor dat de juiste tekst gegenereerd wordt. Een interne klasse members zorgt er bijvoorbeeld voor dat alle sourcefiles, headerfiles en subfolders worden gesorteerd per soort en extensie.

```
public String getRules() {
	if (_type == PROJECT) {
		StringBuilder result = new StringBuilder();
		result.append("all : printstage1 visitcores printstage2 compile");
		result.append(SPACER);
		result.append("clean : remove all");
		result.append(SPACER);
		return result.toString();
	}
	return "";
}
```

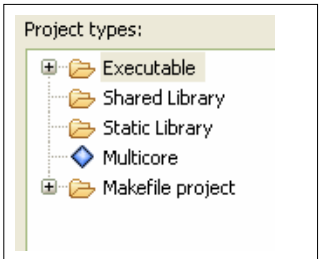
Figuur 17: De toevoeging van de all en clean regel

Case Study

De werking van de multi-core plug-in is mogelijk iets te complex om goed te begrijpen uit de voorgaande tekst. In een poging het geheel inzichtelijker te maken volgen we een klein 'hallo wereld' project.

Project aanmaken

Om te beginnen moet natuurlijk eerst een nieuw project aangemaakt worden. Dit gaat op de bekende wijze via New->Project, of een van de vele andere manieren waarop je een project kan aanmaken. In dit geval kiezen we voor een C++-project.

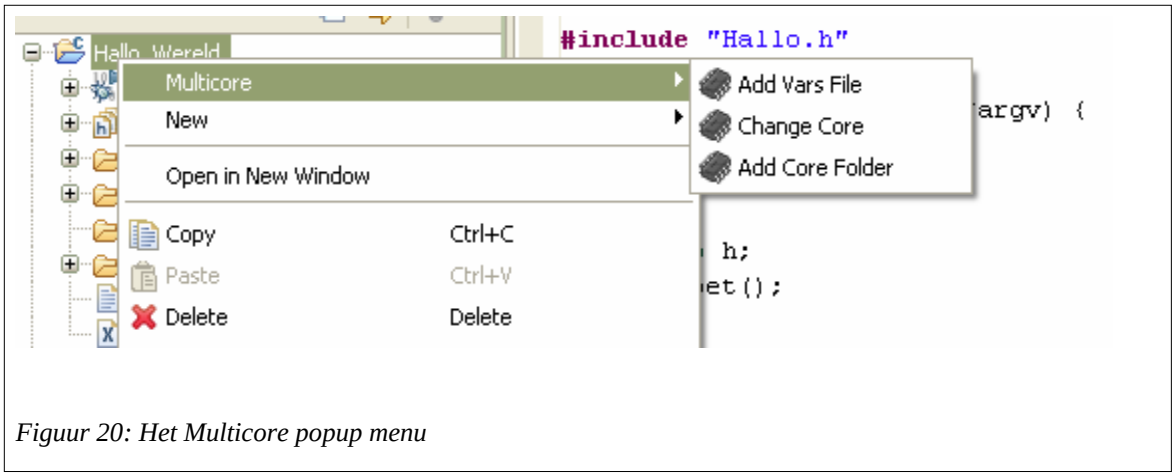


Figuur 18: de projecttypen

In de Project-wizard die hiermee wordt opgestart, is een keuze te maken uit de verschillende projecttypen die gedefinieerd zijn. Waaronder ook het multi-core projecttype. Op de volgende pagina kan eventueel gekozen worden voor een bepaalde set configuraties behorende bij het gekozen projecttype. In dit geval noemen we het project 'hallo_wereld' en kiezen de configuratie 'Cygwin Release'.

Figuur 19: Het nieuwe project

Er wordt nu een nieuw project aangemaakt in de workspace, alsook de bestanden die nodig zijn om van dit project een C/C++-project te maken. Daarnaast wordt er ook een folder 'VARS_DIR' aangemaakt met daarin het bestand 'vars.mk'. De folder is aangebracht om alle verschillende platformspecifieke makefiles in op te slaan, oftewel de varsfiles. Het standaard bestand 'vars.mk' dat daarin meegeleverd wordt, wordt door alle default.mk's meegenomen waarbij geen expliciete andere varsfile aan is toegekend. Het is dan ook wijzigbaar door de gebruiker, maar niet te verwijderen of hernoemen. In een dergelijke situatie zal er door de plug-in direct een verse 'vars.mk' aangemaakt worden.



Figuur 20: Het Multicore popup menu

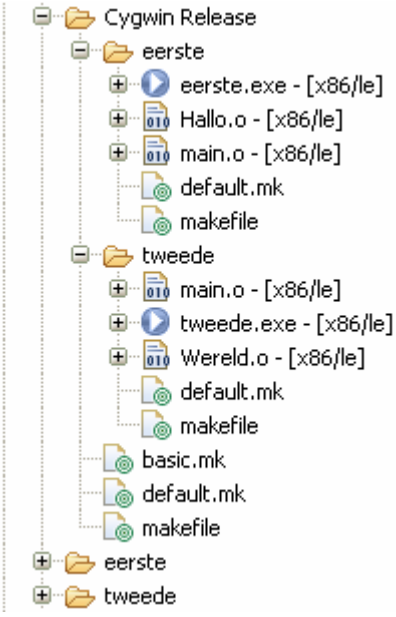
De volgende stap is het aanmaken van twee core-folders. Met een rechterklik op het project komt het pop-up menu te voorschijn. Het item 'multicore' opent een submenu met drie

keuzes: 'Add Vars File', 'Change Core' en 'Add Core Folder' en we kiezen de laatste optie. Dit opent een dialoog waarin de naam van de folder kan worden opgegeven en bijbehorende varsfile kan worden toegekend, het project is al ingevuld, maar kan eventueel gewijzigd worden. De folder geven we de naam 'eerste' en de varsfile laten we leeg. Dit herhalen we voor de tweede folder, toepasselijk 'tweede' genaamd.

Binnen deze folders worden nu elk een klasse aangemaakt, Hallo en Wereld, die elk een functie 'groet()' krijgen, welke in de juiste volgorde uitgevoerd een bekende uitspraak zal geven. Daarnaast wordt er in elke folder ook een losse sourcefile aangemaakt voor de main-functie die de 'groet()' functie van de betreffende klasse aanroept.

Compilatie

Met dit geheel zijn we klaar om te gaan compileren. Een klik op de 'build' knop en een korte wachttijd later, is er een nieuwe folder bijgekomen in het project genaamd 'Cygwin Release'. Dit is de naam van de configuratie die we bij het aanmaken van het project gekozen hebben, en in deze map worden door Eclipse en de plug-in alle verdere makefiles en objectfiles geplaatst om het project zo veel mogelijk overzichtelijk te houden.



Figuur 21: Het resultaat van de build

Als we de nieuwe map openen zien we dat daarin de hele directory structuur, wat betreft de core-folders en wat zich daar eventueel in bevind, van het project gekopieerd is. Daarnaast bevinden zich in de 'Cygwin Release' map de bestanden 'basic.mk, default.mk en 'makefile', in 'eerste' en 'tweede' bevinden zich 'default.mk' en 'makefile'.

```
include $(ROOT)/basic.mk
include $(ROOT)/../VARS_DIR/vars.mk
```

Figuur 22: De include regels van de default.mk's

In de makefiles zien we de plug-in aan het werk, hier vallen in de makefile in 'Cygwin Release' met name de all en clean regel op. In de andere twee makefiles zie je de klassen en main-files terugkomen die daar gedefinieerd zijn. Ook zie je dat elke makefile naar zijn eigen 'default.mk' verwijst, wat op dit moment nog niet zo zinvol lijkt aangezien alle 'default.mk' bestanden hetzelfde zijn. Later zullen we zien dat dit toch daadwerkelijk nut heeft.

```
ROOT = ..

MYPATH = eerste

THIS = eerste

EXECUTABLE = $(THIS)

CPPSRCS = Hallo.cpp main.cpp

HEADERS = Hallo.h

include $(ROOT)/eerste/default.mk
```

Figuur 23: De makefile van 'eerste'

```
ROOT = .

MYPATH =

THIS = Hallo_Wereld

SUBDIRS = eerste tweede

all : printstage1 visitcores printstage2 compile

clean : remove all

include $(ROOT)/default.mk
```

Figuur 24: De project makefile

Naast al deze makefiles zijn ook de objectfiles, 'Hallo.o', 'Wereld.o' en de twee keer een 'main.o' aangemaakt en in hun respectievelijke mappen geplaatst. Ook zien we dat er twee executables beschikbaar zijn, 'eerste' en 'tweede'. Als we deze in de logische volgorde uitvoeren krijgen we de bekende verwelkoming: “hallo wereld”.

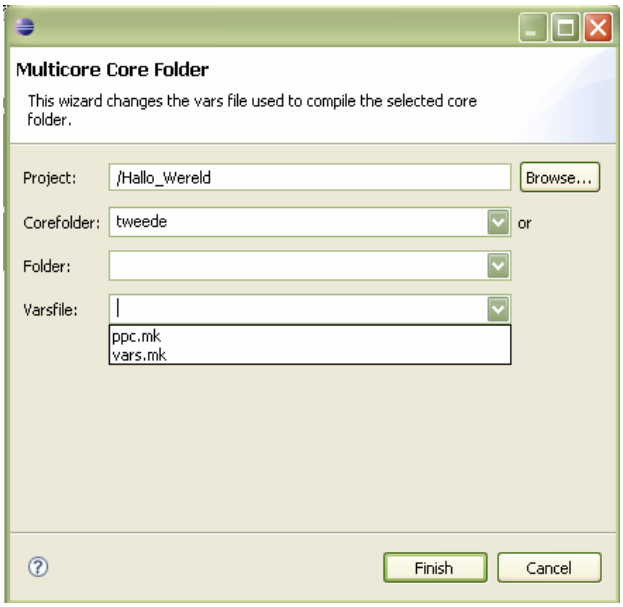
Verschillende Processoren

Stel nu dat we voor de uitvoering van de twee executables die we net gemaakt hebben, twee verschillende processoren, een standaard x86 en een PowerPC, tot onze beschikking hebben. Als elke executable op zijn eigen processor uitgevoerd moet worden dienen er nu ook twee verschillende varsfiles te zijn, een voor elke processortype. Voor de x86 kan de standaard vars.mk gebruikt blijven worden. Voor de powerPC zullen we een nieuwe varsfile aanmaken.

```
COMPILER = g++

LINKER = g++

COMPILEFLAGS = -mcpu=powerpc

LINKFLAGS = -mcpu=powerpc
```

Figuur 25: ppc.mk

Daarvoor doen we weer een rechterklik op het project, en kiezen in het multicore menu voor 'Add Vars File'. We krijgen nu een dialoog te zien waarin we de naam van de varsfile op kunnen en de compiler en linker in kunnen vullen. Deze laatste laten we staan op g++, en de naam wordt 'ppc.mk'. Als we dit bestand openen, uit de 'VARS_DIR', zien we 4 variabelen: COMPILER en LINKER waar g++ aan is toegekend, en COMPILEFLAGS en LINKFLAGS waar we -mcpu=powerpc aan zullen toekennen.



Figuur 26: Het 'Change Core' dialoog

Om deze verse varsfile aan een core-folder te koppelen kiezen we in het multi-core menu voor 'Change Core'. Dit geeft een dialoog waar we kunnen kiezen uit een bestaande core-folder of normale folder om een varsfile aan toe te kennen. Op deze manier kunnen we dus ook een normale map omtoveren tot een core-folder, maar voor nu kiezen we voor de core-folder 'tweede'. In de uitklaplijst bij varsfile kiezen we nu de 'ppc.mk' en drukken op 'Finish'. Uiterlijk is er nu niets aan het project veranderd, maar intern is er de marker op 'tweede' nu aangepast. Als er opnieuw een 'build' gedaan wordt zien we dat in een subtiele verandering in de 'default.mk' voor 'tweede'. Deze neemt nu niet meer de 'vars.mk' mee maar de 'ppc.mk', en de objectfiles en executable van 'tweede' zullen nu dus specifiek voor de PowerPC processor gecompileerd worden terwijl deze voor 'eerste' nog steeds op de x86 zijn toegespitst.

Dit systeem beperkt zich natuurlijk niet tot het wijzigen van een processortype. In de variabelen COMPILEFLAGS en LINKFLAGS kunnen alle flags opgegeven worden die voor de betreffende linker en compiler van toepassing zijn. Als er bijvoorbeeld header bestanden in een aparte directory staan, kan er met de -I optie de directory opgegeven worden waar gezocht moet worden naar deze bestanden.

Conclusies

Het mag duidelijk zijn dat de doelstellingen van dit project behaald zijn, en dat er daarmee geconcludeerd mag worden dat Eclipse prima in staat is om met de hulp van CDT en een simpele plug-in een multi-core project volgens de Compaan methode te ondersteunen. Door de opzet met varsfiles is het zelfs erg makkelijk om verschillende processoren aan elke core-folder te koppelen.

Desalniettemin zijn er natuurlijk ook kanttekeningen te plaatsen. De meeste daarvan hebben

te maken met de CDT die helaas niet altijd even logisch in elkaar steekt. Een goed voorbeeld daarvan is de opbouw van het extension-point 'buildDefinitions'. Nu zou het natuurlijk het makkelijkst geweest zijn als elke tool los hergedefinieerd had kunnen worden met behulp van een plug-in, misschien zelfs de makefile generator zelf[5]. Dit is niet het geval en dat zorgt ervoor dat voor een simpele extensie van de mogelijkheden van de CDT steeds een hele toolchain gekopieerd moet worden.

Zelfs dat was niet zo erg als dat slechts eenmaal had moeten gebeuren per plug-in. Echter de keuze is in de CDT gemaakt om de toolchain onderdeel te maken van een configuratie. Een tegenstelling van wat in de project wizard geïmpliceerd wordt waarin de toolchain gekozen wordt voordat de configuratie bepaald wordt. Aangezien er standaard 5 platformen worden ondersteund met elk een Release en Debug configuratie zorgt dit voor een behoorlijke hoeveelheid zinloos gedupliceerde code.

Voor de debug configuraties is er ook nog een probleem met de aanroep van make. Het hoe en waarom is niet duidelijk, maar de aanroep van 'make all' wordt nooit of mogelijk te vroeg gedaan waardoor het project nooit gebouwd wordt. Op dit moment is dat probleem opgelost door een simpele toevoeging van de 'all' aanroep aan het einde van de 'clean' regel die bij een debug configuratie bij elke build gemaakt wordt. Er moet natuurlijk rekening gehouden worden met de mogelijkheid dat er een bug zit in de multi-core plug-in, maar het ontbreken van enige verwijzing naar 'make' in GNUMakefileGenerator, de inspiratie voor MyMakefileGenerator en zijn hulp-klassen, suggereert dat het probleem in de CDT zit.

Een kleiner probleem zit in de gebruikers interface. De acties die specifiek zijn voor een multi-core project zouden natuurlijk alleen beschikbaar moeten zijn wanneer er daadwerkelijk aan een dergelijk project gewerkt wordt. Binnen Eclipse wordt dat opgelost door een 'nature' aan een project toe te voegen. De nature kan door een plug-in opgevraagd worden, waarna deze besluit functionaliteiten wel of niet aan het menu toe te voegen. Doordat de multi-core plug-in geen eigen project wizard heeft maar zich toevoegt aan de C/C++ project wizard, wordt de initialisatie van het multi-core project op een plaats gedaan die niet door de plug-in bereikbaar of beïnvloedbaar is. Met als gevolg dat er geen multi-core nature aan het project kan worden toegevoegd, en het multi-core menu dus voor alle projecten zichtbaar is.

Het is echter niet zo nuttig om lang stil te staan bij deze problemen. Zoals gezegd is de CDT sterk in ontwikkeling en zullen er op vele punten veranderingen doorgevoerd worden, of ten minste verstandige besluiten genomen worden. Het valt te verwachten dat deze veranderingen vroeger of later ook de multi-core plug-in zullen dwingen tot aanpassing. Het is voor de toekomst dan ook van belang dat de ontwikkelingen in de CDT en met name het Managed Build System bijgehouden worden totdat deze stabiliseren.

Iets wat echter wel op korte termijn verbeterd zou kunnen worden aan de multi-core plug-in is het ontbreken van de 'Incremental Build'. Als de veranderingen in het project steeds tussentijds in de makefiles worden doorgevoerd, zou je pas echt kunnen spreken van een volwaardige plug-in.

Referenties

Scriptie

[1] Inside Intel Core Microarchitecture, setting new standards for energy-efficient performance
Ofri Wechsler, Intel Corporation,
http://download.intel.com/technology/architecture/new_architecture_06.pdf (15-07-2008)

[2] EnergyScale for IBM POWER6 microprocessor-based systems
McCreary, Broyles, Floyd, Geissler, Hartman, Rawson, Rosedahl, Rubio and Ware
http://researchweb.watson.ibm.com/journal/rd/516/mccreary.html (21-08-2008)

[3] Managed Build System extensibility document
Sean Evoy
http://dev.eclipse.org/viewcvs/index.cgi/cdt-home/user/Reference%20Documents/Managed_Build/
Managed_Build_Extensibility.html?root=Tools_Project&view=co (21-08-2008)

[4] Mark My Words
Dejan Glozic and Jeff McAffer
http://www.eclipse.org/articles/Article-Mark%20My%20Words/mark-my-words.html (21-08-2008)

[5] CDT Developers FAQ – Temporary
http://cdt-devel-faq.wikidot.com/#toc29 (21-08-2008)

Project

[5] CDT Eclipsepedia
http://wiki.eclipse.org/index.php/CDT (21-08-2008)

[6] Extending the Eclipse CDT Managed Build System
Chris Recoskie and Leo Treggiari
http://www.ddj.com/cpp/197002115?cid=RSSfeed_DDJ_All (21-08-2008)

[8] GNU Make
http://www.gnu.org/software/make/manual/html_node/index.html (21-08-2008)

[9] GNU Make
http://www.delorie.com/gnu/docs/make/make.html#SEC_Top (21-08-2008)