Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Schriftliche Ausarbeitung für das Modul SiW (Software in der Wissenschaft)

Meson

vorgelegt von

Erik Alaverdyan

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Software-/ System-Entwicklung (SSE)
Matrikelnummer: 7301468

Betreuer: Michael Kuhn

Hamburg, 17.08.2020

Inhaltsverzeichnis

1 Motivation **3**

2 Was ist ein Buildsystem? **4**

3 Meson **5**
3.1 Allgemeines (zu Meson) 5
3.2 Build-Definition schreiben 6
3.3 Erstellen eines Build 7
3.4 Kompilieren 8
3.5 Und ohne Buildsystem? 9
3.6 Die eigene Buildsprache von Meson 10

4 Meson und die Konkurrenz **12**
4.1 Zeitunterschiede 12

5 Zusammenfassung **15**

Literaturverzeichnis **16**

6 Quellen **17**

Abbildungsverzeichnis **18**

1 Motivation

Was ist ein Buildsystem und was bringt es einem?

Darum soll es in dieser Ausarbeitung gehen. Wir werden darauf eingehen, was ein Buildsystem eigentlich ist, was wir damit machen können und wo sich denn Meson in dem Ganzen wiederfindet.
Einige Vorteile eines Buildsystems sind:

- verkürzte Kompilierzeiten,
- Reproduzierbarkeit,
- und vereinfachte Ausführung.

Verkürzte Kompilierzeiten sind zwar bei kleineren Programmen weniger wichtig (wenn überhaupt), aber bei größeren Programmen mit vielen Klassen, Abhängigkeiten, etc. ist es wiederum wichtig. Dies liegt daran, dass man bei kleineren Programmen nur wenig verbessern kann und somit nur eine eben so kleine Menge an Zeit eingespart werden kann. Bei größeren Programmen kann also schon mehr Zeit gespart werden.
Somit haben wir bei kleinen Programmen nicht viel davon, da es so oder so schnell ausgeführt wird. Bei größeren Programmen dauert das Ausführen meist etwas länger und dann etwas Zeit einzusparen ist doch ganz nützlich.

Reproduzierbarkeit ist auch ein wichtiges Thema, da man ohne diese Fehler im Programm nicht so ohne weiteres beseitigen kann. Wenn nun nur manchmal dieser eine Fehler auftritt und manchmal nicht, dann ist das Beheben von diesem Fehler fast unmöglich. Man weiß ja nicht, ob man das Problem gelöst hat oder es doch nur nicht aufgetreten ist, da das Buildsystem es wieder Ganz anders verarbeitet hat.
Also braucht man auch Reproduzierbarkeit um Fehler zu beheben.

Und zu guter letzt, die vereinfachte Ausführung. Man muss mit einem Buildsystem nämlich nicht mehr alle Dateien einzeln und per Hand kompilieren, sondern einfach über einen Knopfdruck einfach alles ausführen lassen.
Man spart dadurch viel aufwand und mühe.

Aber auf das ganze gehen wir noch im folgenden weiter ein..

2 Was ist ein Buildsystem?

Es gibt leider keine klare Definition eines Buildsystems und was es alles können muss, aber hier sind einige der wichtigsten Punkte, die auch von den meisten Buildsystemen vertreten werden:

Ein Buildsystem soll dem Entwickler eine Handvoll Aufgaben abnehmen, die er zwar auch selber machen könnte, aber dies zumal viel zu viel Zeit in Anspruch nehmen würde und auch automatisierbar ist. Dadurch kann der Entwickler sich auch auf die wichtigen Aufgaben konzentrieren z. B. wie man jetzt die Anforderung vom Klienten am besten implementieren könnte..

Eine der Aufgaben eines Buildsystems wäre einen Compiler auszuwählen und dafür zu sorgen, dass die ganzen Abhängigkeiten im Programm berücksichtigt werden.

Ein Buildsystem ist zudem reproduzierbar (bzw. das Endprodukt, das vom Buildsystem erstellt wird, ist reproduzierbar). Das heißt, dass wenn wir 2 mal denselben Quellcode haben und das Buildsystem uns (auf Basis des Quellcodes) eine ausführbare Datei erstellt, dann sollte beide male auch dasselbe in der erstellten Datei stehen!

Wie schon in der Motivation (1 Motivation) erwähnt könnten wir ohne diese Eigenschaft nur schwer Fehler finden und beheben (Dies ist sehr vereinfacht gesagt. Für genaueres kann man auf `https://mesonbuild.com/Reproducible-builds.html#page-description` weiter lesen).

Ein wichtiger Unterschied für das Verständnis von Buildsystemen ist, dass sie keine IDE sind. Eine IDE ist noch viel mehr als nur ein Buildsystem, da sie dem Entwickler Werkzeuge zur Verfügung stellt um z. B. Fehler zu beheben (Debugger) und noch vieles mehr.

Eine Buildsystem hat aber solche Werkzeuge nicht (den Debugger schon, aber in einer anderen Art - Der hilft bei Problemen mit dem Buildsystem und nicht bei Problemen mit dem selbst geschriebenen Quellcode) und sorgt eher dafür, dass der PC das geschriebene Programm auch versteht. Die IDE ist für den Entwickler zuständig und dafür, dass dieser seinen Quellcode angenehm und einfach schreiben kann.

Beliebt ist auch die Eigenschaft von Buildsystemen sich modifizieren zu lassen (z. B. Module). Damit kann der Entwickler das System an sein Programm anpassen und nicht andersrum. Dass das Vorteile mit sich bringt ist glaube ich recht ersichtlich, aber man kann dadurch nicht nur das System an sein Programm anpassen, sondern auch mit ein bisschen Mühe und Interesse auch erweitern, damit auch eher exotischere Aufgaben abgedeckt werden können. [2]

3 Meson

3.1 Allgemeines (zu Meson)

Was ist jetzt aber mit Meson?

Meson ist ein kostenloses Open-Source Buildsystem mit Ninja als Standard Backend. Das heißt Meson ßchreibt"(Siehe Kapitel 3.6 Die eigene Buildsprache von Meson) das ganze Projekt um in einen eigenen Ordner für Ninja und dort kann dann Ninja das ganze Projekt noch weiter verarbeiten. Dazu ist auch noch zu erwähnen, dass Ninja auch ein Buildsystem ist und entwickelt wurde, um die Kompilierzeit zu verringern. Dadurch ist dann Meson entstanden, da man als Mensch nur schwer für Ninja verständlichen Code schreiben kann. Meson ist dabei **nicht** an Ninja gebunden und kann Ninja auch durch andere „Backends“ ersetzten (z. B. vs2019).

Leider ist Meson im Vergleich zu anderen Buildsystemen ziemlich neu (Meson 7 Jahre/ CMake 20 Jahre)und hat somit nur eine kleine Community. Man kann sich aber dennoch auf der Seite von Meson (mesonbuild.com) für eine Mailing-Liste eintragen lassen oder dem IRC (eine Art Chatroom) beitreten. Zudem ist es durch das junge Alter auch möglich, dass Fehler im Quellcode von Meson noch nicht gefunden wurden. Das ist aber nicht so ein großes Problem, da wie schon oben erwähnt es sich um ein Open-Source-Programm handelt. D. h. man kann einfach das Buildsystem um den Fehler ausbessern.

Es werden in Meson 10 Sprachen unterstützt (darunter C, C++, Java, Fortran, ...).

Meson verfolgt das Ziel nach der 90/9/1 Regel Funktionen zu implementieren. Das soll heißen, dass 90 % ohne weiteres funktionieren sollte, 9 % mit Eigenarbeit verbunden ist und 1 % von Meson nicht unterstützt wird.

Die Entwickler von Meson setzten auf die Kommandozeile. Das liegt daran, dass man dem Nutzer so wenig Zeit wie möglich durch ablenkende GUIs oder verschachtelte Menüs stehlen möchte. Dadurch kann man auch ganz einfach in der IDE arbeiten und durch kurzes umschalten zur Kommandozeile ein paar Befehle eingeben und so das Projekt ausführen.[1]

Jetzt wissen wir so einiges über Meson an sich, aber wie Installieren wir das jetzt?

Man muss nur Python3 (python.org) und Ninja (github.com/ninja-build/ninja) installieren. Danach fehlt nur noch Meson (Befehl zum Installieren von Meson über pip: pip3 install meson).[10]

3.2 Build-Definition schreiben

Wie?/ Was?/ Wofür?

Eine Build-Definition ist eine Datei die den Namen „meson.build“ trägt und somit von Meson auch als solche erkannt werden kann. Sie kann über jede Art von Texteditor geschrieben werden und liegt im Source-Ordner (im Projekt Ordner). In ihr wird für Meson definiert, was alles zum Projekt gehört und was man alles braucht um dieses zu kompilieren. Wir gehen das ganze anhand eines Beispieles durch:

```
1 #include<stdio.h>
2
3 int erwiderung() {
4     printf("Hi! Na wie gehts?\n");
5     return 0;
6 }
```

Listing 3.1: Erster Quellcode (Bernd.c)

Wir haben hier in Abbildung 3.1 eine kleine Datei in C geschrieben, die einfach nur „Hi! Na wie gehts?“ auf die Konsole ausgibt und 0 zurückgibt.

```
1 #include<stdio.h>
2 #include "Bernd.c"
3
4 int main(int argc, char **argv) {
5     printf("Hey.\n");
6     erwiderung();
7     return 0;
8 }
```

Listing 3.2: Zweiter Quellcode (Günther.c)

In Abbildung 3.2 haben wir auch eine Datei in C geschrieben und schreiben auch wieder was auf die Konsole („Hey.“). Daraufhin rufen wir aber direkt die Methode aus Bernd.c (Abb. 3.1) und geben 0 zurück. Somit steht dann am Ende:

```
1 Hey.
2 Hi! Na wie gehts?
```

Jetzt haben wir zwar die beiden Dateien Günther.c und Bernd.c, aber an sich können die noch nicht viel anrichten. Deshalb schreiben wir die meson.build Datei um Meson zu sagen was er mit was machen soll.

Man sieht in Abbildung 3.1 wie das Projekt in Zeile 1 Benannt wird und einer Sprache zugewiesen wird (Name = Begrüßung/ Sprache = C).

```
1  project('Begrüßung', 'c')
2  executable('Hey', sources : 'Günther.c')
```

Abbildung 3.1: Build-Definition (meson.build).

In Zeile 2 wird dann eine ausführbare Datei definiert, indem man dem ganzen einen Namen gibt (hier Hey) und auch die Hauptdatei als source angibt (in Java wäre das dann die Datei mit der Main-Methode).

Jetzt weiß zwar Meson was zu tun ist, aber wir müssen ihn das auch tun lassen ... [7][3]

3.3 Erstellen eines Build

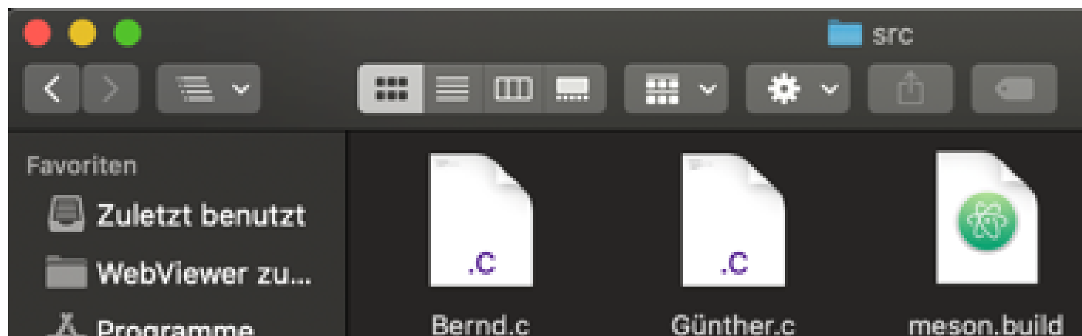

Abbildung 3.2: So sieht das Projekt im Finder bisher aus.

Wir haben jetzt alle 3 Dateien beisammen und können nun über den Befehl eine Build-Directory erstellen:

```
Meson Übung 01 erik$ meson setup src builddir
```

Abbildung 3.3: Befehl zum Build erstellen.

Wenn wir jetzt aus dem Src-Ordner rausgehen, dann haben wir nicht mehr nur den Src-Ordner da, sondern auch noch den builddir Ordner. Der Befehl besteht aus meson setup, um meson anzusprechen und zu zeigen, dass wir das ganze zum ersten Mal machen. Dann haben wir noch src, was der Name des Ordners mit dem Projekt ist. Und noch builddir, was der Name für den neuen Ordner ist (man kann mehrere davon haben mit unterschiedlichen Einstellungen in der meson.build um unterschiedliche Builds zu erhalten -> Tests, Debugging, etc.).[5]

In Abb. 3.5 sieht man das innere der builddir. Dabei ist die build.ninja so ähnlich wie die meson.build bloß für Ninja. Der meson-info Ordner dient als Zusammenfassung

```
The Meson build system
Version: 0.54.1
Source dir: /Users/erik/Desktop/Meson Tests/Meson Übung 01/src
Build dir: /Users/erik/Desktop/Meson Tests/Meson Übung 01/builddir
Build type: native build
Project name: Begrüßung
Project version: undefined
C compiler for the host machine: cc (clang 11.0.0 "Apple clang version 11.0.0 (clan
g-1100.0.33.16)")
C linker for the host machine: cc ld64 530
Host machine cpu family: x86_64
Host machine cpu: x86_64
Build targets in project: 1

Found ninja-1.9.0.git.kitware.dyndep-1.jobserver-1 at /Library/Frameworks/Python.fr
amework/Versions/3.5/bin/ninja
```

Abbildung 3.4: Ausgabe des Befehls aus Abb. 3.3
(zeigt genauere Infos zum Build und dient hier der Vollständigkeit).

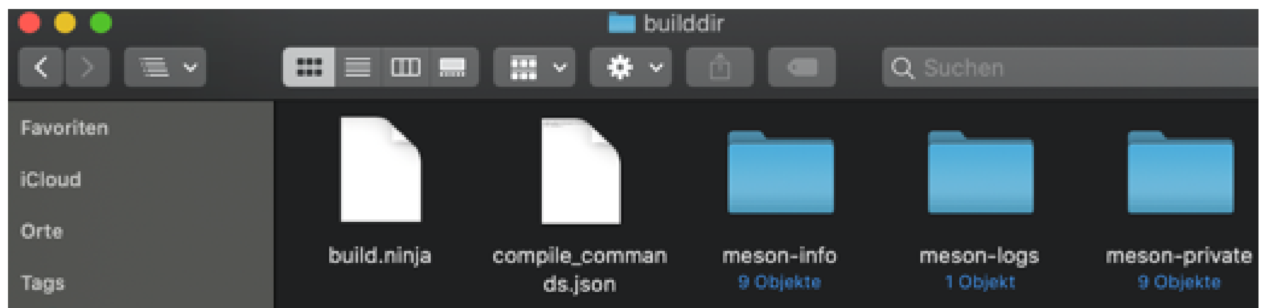

Abbildung 3.5: So sieht das Projekt im Finder nach dem erstellen eines Builds aus.

für die Integrierung in IDEs (so können IDEs einfach diesen Ordner auf Änderungen beobachten). In meson-logs befinden sich wie der Name schon verrät die Logdateien von Meson und in meson-private die für meson wichtigen Dateien bezüglich dem Build. [4]

3.4 Kompilieren

So jetzt haben wir auch die Build-Directory erstellt. Aber um das ganze ausführen zu können fehlt noch ein kleiner aber wichtiger Schritt: Das Kompilieren mit Ninja.

Dafür müssen wir einfach nur folgendes in dieselbe Kommandozeile wie vorhin schreiben:

```
Meson Übung 01 erik$ ninja -C builddir
```

Abbildung 3.6: Befehl zum Kompilieren.

Wir sprechen Ninja direkt über den ninja teil an und navigieren uns in den builddir Ordner über das -C im Befehl.

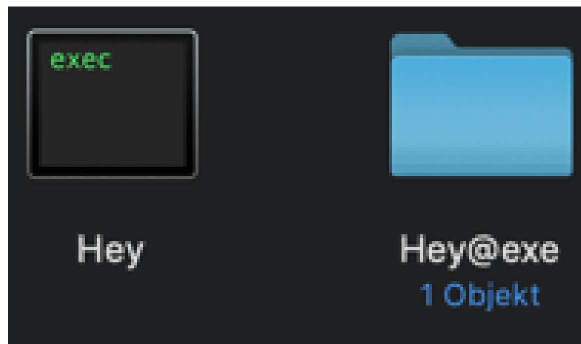

Abbildung 3.7: Entstandene Dateien.

Durch das Kompilieren durch Ninja haben wir jetzt einmal eine ausführbare Datei „Hey“ und den Ordner „Hey@exe“ bekommen. Um dies jetzt auszuführen brauchen wir nur noch den Befehl: [8]

```
[Eriks-MBP:Meson Übung 01 erik$ ./builddir/Hey
Hey.
Hi! Na wie gehts?
```

Abbildung 3.8: Befehl zum Ausführen.

3.5 Und ohne Buildsystem?

Um jetzt zu zeigen, was das Buildsystem uns an Aufgaben abnimmt habe ich das Programm von eben nochmal ohne Buildsystem ausgeführt:

```
[Eriks-MBP:src erik$ gcc -Wall -g -c Günther.c
Eriks-MBP:src erik$ gcc -Wall -g -c Bernd.c
```

Abbildung 3.9: Die einzelnen Dateien Kompilieren.

Wir müssen zunächst die einzelnen Dateien aus dem Programm Stück für Stück per Hand Kompilieren (wird durch das -c ermöglicht). Dabei entstehen je Kompilieren eine Datei mit der neuen Endung „.o“.

Wir müssen aber noch diese beiden verbinden (linken) und das tun wir wie folgt:

Nun haben wir das ganze Kompiliert und können wie schon in Abb. 3.10 das Kompilierte auch ausführen.

Wie man sieht, ist es ohne Buildsystem viel anstrengender und Zeitintensiver etwas zu Kompilieren. Bei diesem Beispiel wurde auch nicht wirklich viel getan. Im Vergleich zu riesigen Programmen wie z. B. die Adobe Programmreihe etc. ist das was wir hier haben nichts. Somit steigt auch der Aufwand um einiges. [11]

```
[Eriks-MBP:src erik$ gcc -o Begrüßung Bernd.o Günther.o -lm
[Eriks-MBP:src erik$ ./Begrüßung
Hey.
Hi! Na wie gehts?
```

Abbildung 3.10: Die einzelnen .o Dateien verbinden/ linken.

3.6 Die eigene Buildsprache von Meson

Meson hat seine eigene „Programmiersprache“ und diese bietet viele Möglichkeiten um das Programm, was man geschrieben hat Meson beschreiben zu können. Ich will hier nur einen groben Überblick über ein paar Befehle von Meson (eine detaillierte Dokumentation findest du unter `https://mesonbuild.com/Manual.html`).

```
1   project('Begrüßung', 'c')
2   bspArray = ['Günther.c']
3   executable('Hey', sources : bspArray)
```

Abbildung 3.11: Arrays verwenden in der Build-Definition.

Hier (Abb. 3.11) kann man sehen wie Arrays in Meson implementiert werden. Man kann auch Arrays addieren, wodurch dann der Inhalt aus beiden Arrays in einen neuen Array gepackt werden und der Variable zugewiesen werden (es werden die alten Arrays nicht erweitert!). [6]

```
4   test('zweiter', exe, priority: 0)
5   test('dritter', exe, priority: -70)
6   test('erster', exe, priority: 2)
```

Abbildung 3.12: Tests anlegen.

Hier (Abb. 3.12) sieht man auch sehr schön, dass man in Meson auch gezielt auf Reihenfolge Testen kann und dass es überhaupt die Möglichkeit gibt Tests in Meson anzulegen. Hier auch nochmal der Output von Abb. 3.12: [9]

```
1/3 erster  OK                0.01s
2/3 zweiter OK                0.01s
3/3 dritter OK                0.01s

Ok:                      3
Expected Fail:           0
Fail:                    0
Unexpected Pass:         0
Skipped:                 0
Timeout:                 0
```

Abbildung 3.13: Output von 3.12.

4 Meson und die Konkurrenz

	Pro:	Contra:
GNU Autotools	Support für Legacy Unix Plattformen	langsam, kompliziert & schwer zu Debuggen
CMake	großer Backend Support	hier und da kompliziert zu nutzen
SCons	Python nutzbar zum schreiben der Build-Scripts/-Definition	langsam und Build-Optionen werden nicht gemerkt
Bazel	kann mit sehr großen Projekten umgehen	schlechter Support für Windows
Meson	schnell, benutzerfreundlich und möglichst unsichtbar für den Nutzer	lleine Community, Backendqualität nicht sehr hoch (außer Ninja) und hat noch unbekannte Bugs.

Abbildung 4.1: Vergleich der unterschiedlichen Buildsysteme („Comparing Meson with other build systems“, https://mesonbuild.com/Comparisons.html, 10.08.2020).

Die Tabelle Abb. 4.1 beinhaltet ein paar Vor- und Nachteile von den unterschiedlichen Buildsystemen.

4.1 Zeitunterschiede

In Abb. 4.2 sieht man wie Meson und CMake mit Ninja zwar nicht die schnellsten sind, aber trotzdem recht schnell. In den folgenden Zeitdiagrammen sieht man auch, wie die Buildsysteme alles in allem am schnellsten sind, wenn sie Ninja als Buildsystem nutzen.

Die Hypothese macht sich immer mehr deutlich wenn man sich Abb. 4.3 und 4.4 anschaut. Die Kompilierzeit ist bei Meson und CMake mit Ninja mit unter am schnellsten und die leere Kompilierzeit ist fast 0.

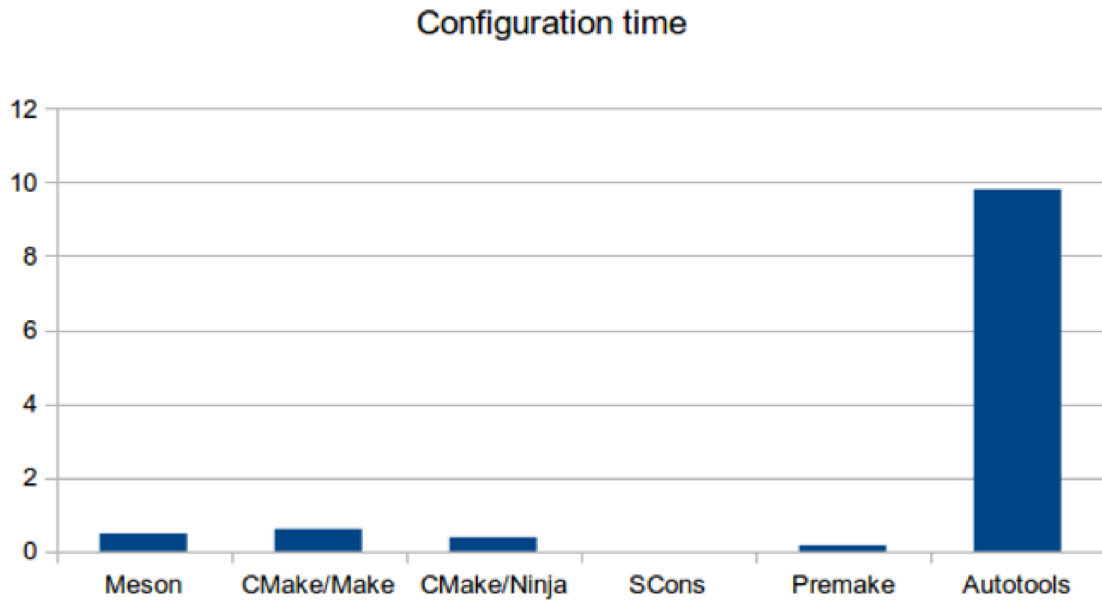

Abbildung 4.2: Vergleicht die unterschiedlichen Konfigurationszeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020).

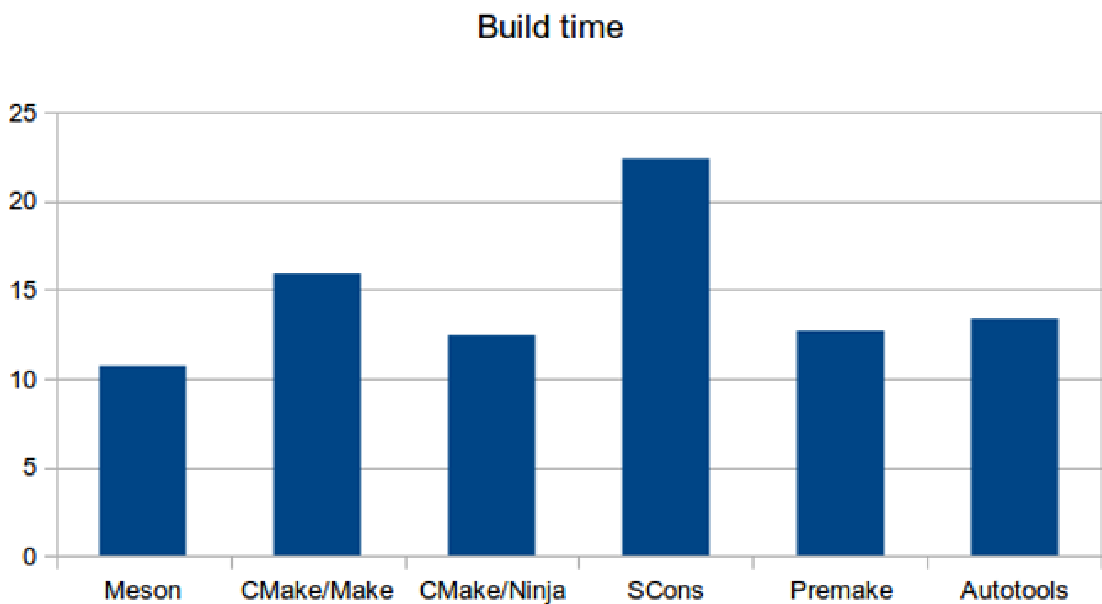

Abbildung 4.3: Vergleicht die unterschiedlichen Kompilierzeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020).

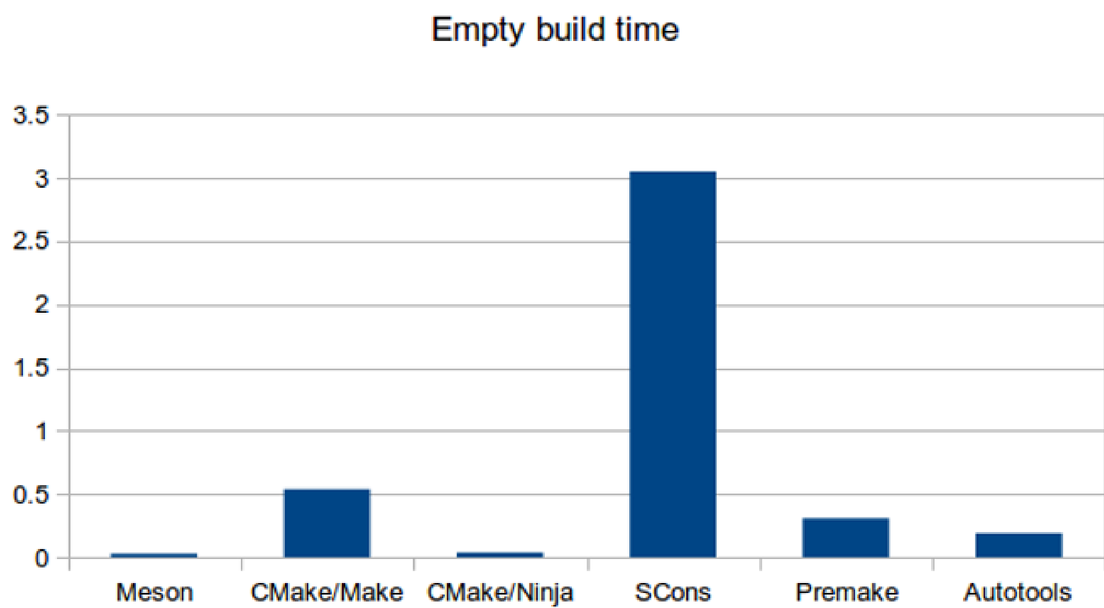

Abbildung 4.4: Vergleicht die unterschiedlichen leeren Kompilierzeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020).

5 Zusammenfassung

Zusammengefasst kann man sagen, dass man wenn es um Schnelligkeit sehr gut auf Meson setzen kann. Zudem ist es auch sehr einfach und schnell zu benutzen. Es ist für einige, die von Buildsystemen nichts wissen ein abstraktes Konstrukt, doch man kann sich recht schnell an das ganze gewöhnen.

Meson hat zudem zwar eine kleine Community wie vorhin schon erwähnt, aber das heißt nicht, dass man bei Fragen/ Problemen keinen Ansprechpartner hat. Im Gegenteil! Es gibt sogar einen Chatroom und eine Mailing Liste.

Die eigene Sprache von Meson ist auch ziemlich nützlich und vereinfacht den ganzen Prozesses des Schreibens einer Builddefinition.

Literaturverzeichnis

[1] EuroPython Conference. Jussi Pakkanen - Meson: compiling the world with Python. 08 2020.

[2] Stuart Lange. What Makes A Build System? 08 2020.

[3] Alex Lugo. What is Meson (and Ninja)? Build system for C/C++, Rust, and Java. 08 2020.

[4] Jussi Pakkanen. IDE integration. 08 2020.

[5] Jussi Pakkanen. Running Meson. 08 2020.

[6] Jussi Pakkanen. Syntax. 08 2020.

[7] Jussi Pakkanen. Tutorial. 08 2020.

[8] Jussi Pakkanen. Tutorial. 08 2020.

[9] Jussi Pakkanen. Unit tests. 08 2020.

[10] Jussi Pakkanen. Using Meson. 08 2020.

[11] Joseph L. Zachary. Answer. 08 2020.

Abbildungsverzeichnis

3.1 Build-Definition (meson.build). 7
3.2 So sieht das Projekt im Finder bisher aus. 7
3.3 Befehl zum Build erstellen. 7
3.4 Ausgabe des Befehls aus Abb. 3.3 (zeigt genauere Infos zum Build und dient hier der Vollständigkeit). 8
3.5 So sieht das Projekt im Finder nach dem erstellen eines Builds aus. . . . 8
3.6 Befehl zum Kompilieren. 8
3.7 Entstandene Dateien. 9
3.8 Befehl zum Ausführen. 9
3.9 Die einzelnen Dateien Kompilieren. 9
3.10 Die einzelnen .o Dateien verbinden/ linken. 10
3.11 Arrays verwenden in der Build-Definition. 10
3.12 Tests anlegen. 10
3.13 Output von 3.12. 11

4.1 Vergleich der unterschiedlichen Buildsysteme („Comparing Meson with other build systems“, https://mesonbuild.com/Comparisons.html, 10.08.2020). 12
4.2 Vergleicht die unterschiedlichen Konfigurationszeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020). . . . 13
4.3 Vergleicht die unterschiedlichen Kompilierzeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020). 13
4.4 Vergleicht die unterschiedlichen leeren Kompilierzeiten („A simple comparison“, https://mesonbuild.com/Simple-comparison.html, 10.08.2020). . . 14